Embedded BIOS™ 4.3

The Full-Featured BIOS for Embedded Systems, Handheld, Mobile, and Consumer Electronics*

OEM Adaptation Guide with BIOS Interrupt Reference

"The most configurable BIOS available"

*for Source Code Adaptation Kit Users
and Online Adaptation Kit Users

Includes BIOStart™

This material is provided as a product component for the EMBEDDED BIOS Adaptation Package. It is licensed material and cannot be redistributed without written permission from General Software.

To get started immediately, turn to the installation instructions in Chapter 2 of this manual.

General Software, Inc.
Box 2571
Redmond, Washington 98073

Tel: (425) 454-5755
FAX: (425) 454-5744
Web: http://www.gensw.com
Email: support@gensw.com

Copyright (C) 1990-2000 General Software, Inc. All rights reserved.

IMPORTANT NOTICES

General Software, the GS logo, EMBEDDED BIOS, Embedded DOS, Embedded LAN, CE Ready, the CE Ready logo, CodeProbe, The Snooper, EtherProbe, and "The Soul of Your Next Machine" are trademarks or registered trademarks of General Software, Inc.

Please complete and return your **Product Registration** card immediately. This will help us to notify you of updates, and make you eligible to receive technical support and access to on-line services.

Important Licensing Information

IMPORTANT -- READ CAREFULLY BEFORE OPENING THIS PACKAGE. BY OPENING THIS SEALED PACKAGE, INSTALLING OR OTHERWISE USING THE SOFTWARE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE AGREEMENT, DO NOT OPEN THE PACKAGE OR USE THIS SOFTWARE AND PROMPTLY RETURN THE UNOPENED SOFTWARE AND ANY ACCOMPANYING MATERIALS TO GENERAL SOFTWARE FOR A REFUND.

This License Agreement is a legal agreement between you and General Software, Inc. ("General") for the General software product identified above, which includes the computer software and any associated media and printed materials or electronic documentation (collectively, the "Software ").

GRANT OF LICENSE. In consideration of the license fees paid to General, and subject to the terms and conditions set forth herein, General hereby grants you a nonexclusive, nontransferable license (the "Adaptation License") to use the Software and the accompanying documentation solely for your internal use to evaluate and adapt the Software for use in products manufactured by you. Other than for the purpose of evaluating and adapting the Software for use with your products, you may not reproduce or install copies of the Software. Prior to distributing the Software in such products or installing the Software in the products for distribution, you must enter into a separate license agreement with General for such installation and distribution and pay the applicable royalties.

SOURCE CODE. Subject to the terms and conditions set forth herein, General grants you a nonexclusive, nontransferable license to use portions of the source code for the Software for your internal use only, for the sole purpose of adapting the Software to your products. You will keep the source code under restricted access in a safe and secure place and shall take reasonable precautions and actions to protect the source code from unauthorized use or disclosure. Such precautions and actions shall be at least as stringent as those you take to protect your own source code or other confidential information. You will not disclose the source code to any person or entity without the prior written consent of General, except that you may disclose the source code to your employees on a need-to-know basis, provided such employees are contractually obligated to maintain the confidentiality of the source code. The foregoing obligations will survive any termination of this Adaptation License.

PROPRIETARY RIGHTS. This Software is licensed to you, not sold, and is protected by copyright laws and international treaty provisions. All title, copyrights and other proprietary rights in and to the Software, the accompanying printed materials, and any copies thereof are owned by General. No title to the Software or any copy thereof or any associated proprietary rights are transferred to you by this license.

DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS. You may not copy, use or distribute all or any portion of the Software or the accompanying printed material except as expressly permitted herein. You may not reverse engineer, decompile, or disassemble the Software, nor may you modify the Software except as necessary to adapt the Software for use with your products. You may not sell, assign, rent, lease or lend the Software.

TERMINATION. Without prejudice to any other rights or remedies, General may terminate this Adaptation License if you breach any of the terms and conditions of this License Agreement. In such event, you must destroy and/or erase all copies of the Software and all of its component parts.

SUPPORT. For a period of thirty (30) days from the date you acquired this Software, General will provide, subject to availability, up to 5 hours of telephone support to assist you with questions regarding the installation and adaptation of the Software. If additional support is desired, a variety of support plans are available from General by entering into a separate support agreement. You acknowledge that except as may be expressly set forth in a separate written support agreement entered into by the parties, GENERAL MAKES NO WARRANTIES OR REPRESENTATIONS OF ANY KIND WITH RESPECT TO THE RESULTS OR AVAILABILITY OF SUCH SUPPORT.

DISCLAIMER OF WARRANTY. You acknowledge that the Software and any accompanying materials are being furnished solely for evaluation and adaptation purposes. Therefore, **THE SOFTWARE AND THE ACCOMPANYING MATERIALS ARE BEING PROVIDED "AS IS" AND GENERAL MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THE SOFTWARE, EITHER EXPRESS OR IMPLIED, AND THERE IS EXPRESSLY EXCLUDED ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.**

INDEMNIFICATION. Without limiting the generality of any of the foregoing, you acknowledge that the Software is not designed or intended to be used in connection with any product, the operation, use or malfunction of which could result in death or serious bodily harm, and you agree to take full responsibility for the use of and results achieved from the incorporation of the Software into your products. You agree to defend, indemnify and hold harmless General, and its officers, directors, employees and agents, from and against any and all claims, actions, proceedings, liabilities, costs and expenses (including without limitation reasonable attorneys' fees) arising out of (a) any representation, warranty, act or omission made by you in connection with your products or the Software, or (b) the use of or inability to use your products or the Software incorporated therein or distributed therewith.

LIMITATION OF LIABILITY. IN NO EVENT SHALL GENERAL BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THIS ADAPTATION LICENSE OR THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF GENERAL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL GENERAL'S LIABILITY UNDER THIS ADAPTATION LICENSE OR RELATED TO THE SOFTWARE (WHETHER IN TORT, CONTRACTS OR OTHERWISE) EXCEED THE AMOUNTS PAID TO GENERAL FOR THIS ADAPTATION LICENSE.

FEDERAL GOVERNMENT ACQUISITION. By accepting delivery of this Software, the Government hereby agrees that this Software qualifies as "commercial computer software" as that term is used in the acquisition regulation applicable hereto. To the maximum extent possible under federal law, the Government will be bound by the commercial terms and conditions contained in this license. The following additional statement applies only to procurements governed by DFARS Subpart 227.4 (1988): Restricted Rights - Use, duplication and disclosure by the Government is subject to restriction as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 (1988).

MISCELLANEOUS. This License Agreement constitutes the entire agreement between you and General regarding the Software. This License Agreement is governed by the laws of the State of Washington and the United States of America, without reference to its choice of law rules. The provisions of the 1980 U.N. Convention on Contracts for the International Sale of Goods shall not apply.

TABLE OF CONTENTS

INTRODUCTION 1

INTRODUCING EMBEDDED BIOS 1
CONFIGURABILITY 1
EMBEDDED FEATURES 2
DESKTOP PC FEATURES 3
SOFTWARE COMPATIBILITY 4
APPLICATIONS FOR EMBEDDED BIOS 4
LOWERED SYSTEM COST WITH EMBEDDED DOS-ROM 5
CHOOSING EMBEDDED DOS-ROM OR EMBEDDED DOS 6-XL 5
RELATED READING 6
ABOUT THE EMBEDDED BIOS ADAPTATION KIT 6
FOR CUSTOMERS WITH VERSION 4.0 OF EMBEDDED BIOS 7
FOR CUSTOMERS WITH VERSIONS OF EMBEDDED BIOS EARLIER THAN 4.0 8

PART I 11

BASIC STEPS FOR BIOS BUILDING 11

INSTALLATION 15

BACKING-UP YOUR RELEASE DISKS 15
INSTALLING THE CORE EMBEDDED BIOS SOFTWARE 15
INSTALLING ADDITIONAL SUPPORT MODULES 16
ORGANIZATION OF THE SOFTWARE 16
PROJECTS SUBDIRECTORY 16
SYSTEM SUBDIRECTORY 17
SYSTEM32 SUBDIRECTORY 17
INC SUBDIRECTORY 18
CHIPSETS SUBDIRECTORY 18
CPUS SUBDIRECTORY 19
BOARDS SUBDIRECTORY 19
TOOLS SUBDIRECTORY 19
UTIL SUBDIRECTORY 20
COW SUBDIRECTORY 20
RESOURCE SUBDIRECTORY 20
WHAT'S NEXT? 20

KEY EMBEDDED BIOS CONCEPTS 23

3.1 ARCHITECTURAL OVERVIEW 23
3.1.1 MEMORY MODEL 24
3.1.1.1 The Interrupt Vector Table 24
3.1.1.2 The BIOS Data Area 24
3.1.1.3 Free Low RAM 24
3.1.1.4 The Extended BIOS Data Area 25
3.1.1.5 Expanded Memory 25
3.1.1.6 Video ROM Extensions 25
3.1.1.7 Other ROM Extensions 25
3.1.1.8 The System ROM 26
3.1.1.9 Extended Memory 26
3.1.1.10 CMOS Memory 26
3.1.2 INTERRUPT MODEL 26
3.1.2.1 BIOS Service Interrupts 28
3.1.2.1.1 INT 10h, Video Services 28
3.1.2.1.2 INT 11h, Equipment List Service 29
3.1.2.1.3 INT 12h, Low Memory Size Service 29
3.1.2.1.4 INT 13h, Disk Services 30
3.1.2.1.5 INT 14h, Serial Port Services 32
3.1.2.1.6 INT 15h, General System Services 33
3.1.2.1.7 INT 16h, Keyboard Services 34
3.1.2.1.8 INT 17h, Parallel Port Services 35
3.1.2.1.9 INT 18h, Boot Fault Routine 35
3.1.2.1.10 INT 19h, Bootstrap Routine 35
3.1.2.1.11 INT 1ah, Time/Date Services 36
3.1.2.2 Table Pointers 37
3.1.2.2.1 INT 1dh, Video Parameter Table (VPT) 37
3.1.2.2.2 INT 1eh, Floppy Diskette Parameter Table (DPT) 37
3.1.2.2.3 INT 1fh, Video Graphics Character Table (VGCT) 38
3.1.2.2.4 INT 41h/46h, Fixed Disk Paramter Tables (FDPTs) 39
3.1.2.3 BIOS Upcalls 39
3.1.2.3.1 INT 15h Device Management 39
3.1.2.3.1.1 INT 15h Function 4fh 40
3.1.2.3.1.2 INT 15h Function 90h 40
3.1.2.3.1.3 INT 15h Function 91h 40
3.1.2.3.1.4 INT 15h Function 85h 41
3.1.2.3.2 INT 1bh Control-Break Signal 41
3.1.2.3.3 INT 1ch User Timer Interrupt 41
3.1.2.3.4 INT 4ah Real Time Software Interrupt 41
3.1.2.4 CPU Traps/Faults 42
3.1.2.5 Hardware Interrupts 43
3.3 SETUP SCREENS 44
3.4 API SERVICE MODULES 44
3.5 DEVICE SERVICE MODULES 45
3.6 OTHER MODULES 46
3.7 CPU PERSONALITY MODULES 46
3.8 CHIPSET PERSONALITY MODULES 48
3.9 BOARD PERSONALITY MODULES 49
3.10 BIOS CONFIGURATION 51
3.10.1 PROJECT FILES 51

3.10.2 BINARY CONFIGURATION PATCH AREA 52
3.10.3 SYSTEM CONFIGURATION TABLE 52
3.10.4 KEYBOARD SCANCODE TRANSLATION TABLE 53
3.11 CONSOLE I/O REDIRECTION 53
3.11.1 VIDEO (INT 10H) REDIRECTION 53
3.11.2 KEYBOARD (INT 16H) REDIRECTION 54
3.12 INTEGRATED BIOS DEBUGGER 54
3.13 MANUFACTURING MODE 55
3.14 ROM DISK 56
3.15 WATCHDOG TIMER 56
3.16 POWER MANAGEMENT AND APM 57
3.17 CACHE MANAGEMENT 57
3.18 PROTECTED MODE SUPPORT 58

SETTING UP YOUR DEVELOPMENT TOOLS 59

4.1 CONFIGURING FOR BORLAND OR MICROSOFT TOOLS 59
4.1.1 OBTAINING BORLAND 32-BIT TOOLS 59
4.1.2 OBTAINING MICROSOFT 32-BIT TOOLS 60
4.1.3 BUILD CONTROL ENVIRONMENT VARIABLES 60
4.2 STANDARD SYSTEM/TOOLSET ENVIRONMENT VARIABLES 62
4.3 USING OTHER COMPILERS, ASSEMBLERS, AND LINKERS 62
4.4 GSMAKE, THE PROGRAM MAINTENANCE UTILITY 62
4.4.1 STARTING GSMAKE 63
4.4.2 COMMAND LINE OPTIONS 63
4.4.3 TYPES OF MAKEFILE STATEMENTS 64
4.4.4 INTRINSIC GSMAKE COMMANDS 65
4.5 GSMERGE, THE MERGE UTILITY 66
4.5.1 OVERVIEW OF GSMERGE OPERATION 66
4.5.2 IDF FILE SYNTAX 66
4.5.3 IDF KEYWORDS 67
4.5.3.1 IMAGEDEF Keyword 67
4.5.3.2 AT Keyword 67
4.5.3.3 ALIGN Keyword 68
4.5.3.4 RESERVE and RESERVETO Keywords 68
4.5.3.5 FREE and FREETO Keywords 68
4.5.3.6 SET and SETTO Keywords 69
4.5.3.7 FROM Keyword 69
4.5.3.8 TO Keyword 69
4.5.3.9 INCLUDEMAP Keyword 69
4.5.3.10 LOCATEPE Keyword 70
4.5.3.11 LOCATERES Keyword 70
4.5.3.12 PLACEDIR32 Keyword 70
4.5.3.13 LOAD Keyword 71
4.5.3.14 COMPRESSTO Keyword 71
4.5.3.15 LOADPE Keyword 71
4.5.3.16 CONVERTBMP Keyword 72
4.5.3.17 LOADRES Keyword 72
4.5.3.18 INCLUDE Keyword 72

4.5.3.19 SETADDRESS Keyword 72
4.5.4 Example IDF File 74
4.6 DISKIMAG, THE DISK IMAGE GENERATOR 75
4.7 BIOSLOC, THE ROM BIOS EXTENSION LOCATOR 76
4.8 BIOSSUM, THE ROM BIOS EXTENSION CHECKSUM UTILITY 77
4.9 BIOSMAP, THE EMBEDDED BIOS MAP FILE ANALYZER 78
4.10 PERF, THE FILE SYSTEM PERFORMANCE ANALYZER 78
4.10.1 STARTING PERF 78
4.10.2 COMMAND LINE OPTIONS 79
4.10.3 MULTIPLE PASSES 80
4.10.4 MULTIPLE REPETITIONS PER PASS 80
4.10.5 SOME EXAMPLES 81

BUILDING EMBEDDED BIOS 83

5.1 BUILDING THE SYSTEM BIOS 83
5.1.1 CONFIGURING BUILD OPTIONS AND PARAMETERS 83
5.1.2 SELECTING THE CPU PERSONALITY MODULE 83
5.1.3 SELECTING THE CHIPSET PERSONALITY MODULE 84
5.1.4 SELECTING THE BOARD PERSONALITY MODULE 85
5.1.5 TYPE GSMAKE IN DOS OR IN A DOS BOX UNDER WINDOWS 85
5.1.6 INSPECTING THE BINARY 16-BIT SYSTEM BIOS FILE 85
5.1.7 PROGRAMMING A BOOT ROM WITH MYPROJ.ABS 87
5.1.8 THE 32-BIT BIOS BUILD, AND COMPOSITE BIOS FILES 87
5.1.9 BOOTING THE SYSTEM 88
5.2 BUILDING AUXILLIARY COMPONENTS 88

CONFIGURING THE BIOS WITH BIOSTART 91

6.1 OVERVIEW OF BIOSTART 91
6.2 INSTALLING THE ADAPTATION KIT WITH BIOSTART 93
6.3 CREATING AND EDITING A PROJECT 93
6.4 CUSTOMIZING A PROJECT 95
6.5 PRINTING PROJECT CUSTOMIZATION SETTINGS 97
6.6 SAVING THE PROJECT AND SETTINGS 97
6.7 BUILDING THE PROJECT 98
6.8 PATCHING BINARY SYSTEM BIOS FILES 98
6.9 UPGRADING BIOSTART 99

BIOS BUILD OPTIONS 101

7.1 OPTIONS FOUND IN OPTIONS.INC 102
7.1.1 BIOS_MAJOR_VERSION CONSTANT 102
7.1.2 BIOS_MINOR_VERSION CONSTANT 102
7.1.3 OPTION_BIOS_KBSIZE OPTION 102
7.1.4 OPTION_SUPPORT_PCODE OPTION 103
7.1.5 OPTION_SUPPORT_SETUP OPTION 104

7.1.6 OPTION_SUPPORT_CONFIGBOX OPTION ... 105
7.1.7 OPTION_SUPPORT_POSTCODES OPTION ... 105
7.1.8 OPTION_SUPPORT_POSTCODES_COM OPTION ... 106
7.1.9 OPTION_SUPPORT_MFGCODES OPTION ... 106
7.1.10 OPTION_SUPPORT_POSTMSGS OPTION ... 107
7.1.11 OPTION_SUPPORT_POWERON_DELAY OPTION ... 107
7.1.12 OPTION_SUPPORT_DEBUGGER OPTION ... 108
7.1.13 OPTION_SUPPORT_SHADOW OPTION ... 109
7.1.14 OPTION_SUPPORT_CACHE OPTION ... 110
7.1.15 OPTION_SUPPORT_8250 OPTION ... 110
7.1.16 OPTION_SUPPORT_8254 OPTION ... 111
7.1.17 OPTION_SUPPORT_8255 OPTION ... 112
7.1.18 OPTION_SUPPORT_PORT_B OPTION ... 112
7.1.19 OPTION_SUPPORT_8259 OPTION ... 113
7.1.20 OPTION_SUPPORT_8259_2 OPTION ... 114
7.1.21 OPTION_SUPPORT_8237 OPTION ... 115
7.1.22 OPTION_SUPPORT_8237_2 OPTION ... 115
7.1.23 OPTION_SUPPORT_8042 OPTION ... 116
7.1.24 OPTION_SUPPORT_CMOS OPTION ... 117
7.1.25 OPTION_SUPPORT_NPX OPTION ... 121
7.1.26 OPTION_SUPPORT_V25 OPTION ... 121
7.1.27 OPTION_SUPPORT_XT_NMI OPTION ... 122
7.1.28 OPTION_SUPPORT_VIDEO OPTION ... 122
7.1.29 OPTION_SUPPORT_KEYBOARD OPTION ... 124
7.1.30 OPTION_SUPPORT_TESTBASEMEM OPTION ... 126
7.1.31 OPTION_SUPPORT_PAGEREG OPTION ... 126
7.1.32 OPTION_SUPPORT_XTEXPANSION OPTION ... 126
7.1.33 OPTION_SUPPORT_SCT OPTION ... 127
7.1.34 OPTION_SUPPORT_PROTECT_MODE OPTION ... 127
7.1.35 OPTION_SUPPORT_SERIAL OPTION ... 130
7.1.36 OPTION_SUPPORT_PARALLEL OPTION ... 132
7.1.37 OPTION_SUPPORT_ROM_EXTENSIONS OPTION ... 132
7.1.38 OPTION_SUPPORT_VIDEO_BOARDS OPTION ... 133
7.1.39 OPTION_SUPPORT_SOUND OPTION ... 134
7.1.40 OPTION_SUPPORT_DEVICECALLS OPTION ... 136
7.1.41 OPTION_SUPPORT_TIMEBIOS OPTION ... 136
7.1.42 OPTION_SUPPORT_APM OPTION ... 137
7.1.43 OPTION_SUPPORT_POWERMAN OPTION ... 138
7.1.44 OPTION_SUPPORT_PCI OPTION ... 139
7.1.45 OPTION_SUPPORT_PCI_POSTMSGS OPTION ... 140
7.1.46 OPTION_SUPPORT_MCA OPTION ... 140
7.1.47 OPTION_SUPPORT_PS2MOUSE OPTION ... 140
7.1.48 OPTION_SUPPORT_WATCHDOG OPTION ... 141
7.1.49 OPTION_SUPPORT_SOFT_ERR OPTION ... 142
7.1.50 OPTION_SUPPORT_MINI_DOS OPTION ... 143
7.1.51 OPTION_SUPPORT_EXHMEMTEST OPTION ... 143
7.1.52 OPTION_SUPPORT_KNOWN_ENTRYPOINTS OPTION ... 145
7.1.53 OPTION_SUPPORT_IBM_COMPAT OPTION ... 145
7.1.54 OPTION_SUPPORT_MFGMODE OPTION ... 145
7.1.55 OPTION_SUPPORT_PARITY OPTION ... 147

7.1.56 OPTION_SUPPORT_PASSWORD OPTION 147
7.1.57 OPTION_SUPPORT_DEMO OPTION 148
7.1.58 OPTION_SUPPORT_DEMO_MSG OPTION....148
7.1.59 OPTION_SUPPORT_ATA OPTION....149
7.1.60 OPTION_SUPPORT_CON_REDIRECTOR OPTION....149
7.1.61 OPTION_SUPPORT_MCL OPTION 150
7.1.62 OPTION_SUPPORT_DISKIO OPTION....150
7.1.63 OPTION_SUPPORT_WINCE OPTION 151
7.1.64 OPTION_SUPPORT_BOOT_FAR OPTION 151
7.1.65 OPTION_SUPPORT_BIOS32 OPTION 152
7.1.66 OPTION_SUPPORT_SPLASHSCR OPTION....152
7.1.67 OPTION_SUPPORT_EXTRES OPTION....153
7.1.68 OPTION_SUPPORT_INT13_EXTENSIONS OPTION....153
7.1.69 OPTION_SETUP_CUSTOM OPTION....153
7.1.70 OPTION_SETUP_DEMO OPTION....154
7.1.71 OPTION_SETUP_PASSWORD OPTION....154
7.1.72 OPTION_SETUP_DIAGNOSTICS OPTION 155
7.1.73 OPTION_SETUP_DEBUGGER OPTION....155
7.1.74 OPTION_SETUP_IDE OPTION....156
7.1.75 OPTION_SETUP_SHADOW OPTION 156
7.1.76 OPTION_SETUP_PWR_FEATURES OPTION 156
7.1.77 OPTION_SETUP_PWR_TIMEOUTS OPTION 157
7.1.78 OPTION_SETUP_MFGMODE OPTION....158
7.1.79 OPTION_SETUP_RAMDISK OPTION 158
7.1.80 OPTION_SETUP_RFDDISK OPTION 158
7.1.81 OPTION_SETUP_SHAD_C000 OPTION....159
7.1.82 OPTION_SETUP_SHAD_C400 OPTION....159
7.1.83 OPTION_SETUP_SHAD_C800 OPTION....160
7.1.84 OPTION_SETUP_SHAD_CC00 OPTION 160
7.1.85 OPTION_SETUP_SHAD_D000 OPTION....161
7.1.86 OPTION_SETUP_SHAD_D400 OPTION....161
7.1.87 OPTION_SETUP_SHAD_D800 OPTION....162
7.1.88 OPTION_SETUP_SHAD_DC00 OPTION....162
7.1.89 OPTION_SETUP_SHAD_E000 OPTION 163
7.1.90 OPTION_SETUP_SHAD_E400 OPTION 163
7.1.91 OPTION_SETUP_SHAD_E800 OPTION 164
7.1.92 OPTION_SETUP_SHAD_EC00 OPTION 164
7.1.93 OPTION_SETUP_SHAD_F000 OPTION 165
7.1.94 OPTION_REFRESH_8237 OPTION....165
7.1.95 OPTION_REFRESH_CHIPSET OPTION....166
7.1.96 OPTION_REFRESH_CPU OPTION 167
7.1.97 OPTION_REFRESH_BOARD OPTION 168
7.1.98 OPTION_REFRESH_CHARGE OPTION....168
7.1.99 OPTION_DMA_8237 OPTION 169
7.1.100 OPTION_DMA_CPU OPTION 169
7.1.101 OPTION_DMA_BOARD OPTION 170
7.1.102 OPTION_INT_8259 OPTION....170
7.1.103 OPTION_INT_CPU OPTION 171
7.1.104 OPTION_INT_BOARD OPTION....171
7.1.105 OPTION_TIMER_8254 OPTION....172

7.1.106 OPTION_TIMER_CPU OPTION 172
7.1.107 OPTION_TIMER_BOARD OPTION 172
7.1.108 OPTION_SOUND_8254_8255 OPTION 173
7.1.109 OPTION_SOUND_CPU OPTION 173
7.1.110 OPTION_SOUND_BOARD OPTION 174
7.1.111 OPTION_WATCHDOG_CHIPSET OPTION 174
7.1.112 OPTION_WATCHDOG_CPU OPTION 174
7.1.113 OPTION_WATCHDOG_BOARD OPTION 175
7.1.114 OPTION_WATCHDOG_TIMER_KICK OPTION 175
7.1.115 OPTION_CACHE_CPU OPTION 176
7.1.116 OPTION_CACHE_CHIPSET OPTION 176
7.1.117 OPTION_CACHE_BOARD OPTION 177
7.1.118 OPTION_SPEED_CPU OPTION 177
7.1.119 OPTION_SPEED_CHIPSET OPTION 177
7.1.120 OPTION_SPEED_BOARD OPTION 178
7.1.121 OPTION_A20_8042 OPTION 178
7.1.122 OPTION_A20_CHIPSET OPTION 179
7.1.123 OPTION_A20_CPU OPTION 179
7.1.124 OPTION_A20_BOARD OPTION 180
7.1.125 OPTION_A20_PORT92 OPTION 181
7.1.126 OPTION_A20_FAILMEM OPTION 181
7.1.127 OPTION_REBOOT_JUMP OPTION 182
7.1.128 OPTION_REBOOT_PORT92 OPTION 182
7.1.129 OPTION_REBOOT_8042 OPTION 183
7.1.130 OPTION_REBOOT_CHIPSET OPTION 183
7.1.131 OPTION_REBOOT_BOARD OPTION 183
7.1.132 OPTION_TOREAL_PORT92 OPTION 184
7.1.133 OPTION_TOREAL_8042 OPTION 184
7.1.134 OPTION_TOREAL_CPU OPTION 185
7.1.135 OPTION_POWERMAN_CPU OPTION 185
7.1.136 OPTION_POWERMAN_CHIPSET OPTION 186
7.1.137 OPTION_POWERMAN_BOARD OPTION 186
7.1.138 OPTION_SERIAL_8250 OPTION 187
7.1.139 OPTION_SERIAL_CPU OPTION 187
7.1.140 OPTION_SERIAL_WAIT_DSR OPTION 188
7.1.141 OPTION_SERIAL_WAIT_DSRCTS OPTION 188
7.1.142 OPTION_SERIAL_FIFO OPTION 189
7.1.143 OPTION_SERIAL_HALT OPTION 190
7.1.144 OPTION_SERIAL_9600_BAUD OPTION 190
7.1.145 OPTION_PARALLEL_EXTERNAL OPTION 191
7.1.146 OPTION_PARALLEL_CPU OPTION 191
7.1.147 OPTION_KEYBOARD_PCAT OPTION 192
7.1.148 OPTION_KEYBOARD_CUSTOMER OPTION 192
7.1.149 OPTION_KEYBOARD_MATRIX OPTION 193
7.1.150 OPTION_KEYBOARD_PCXT OPTION 193
7.1.151 OPTION_KEYBOARD_CHIPSET OPTION 193
7.1.152 OPTION_8042_TESTP22P23 OPTION 194
7.1.153 OPTION_8042_READPWRSTAT OPTION 194
7.1.154 OPTION_8042_CHECKBAT OPTION 195
7.1.155 OPTION_8042_PS2 OPTION 195

7.1.156 OPTION_8042_WAIT_BEFORE_BAT OPTION 195
7.1.157 OPTION_VIDEO_6845 OPTION 196
7.1.158 OPTION_VIDEO_HD61830 OPTION 197
7.1.159 OPTION_VIDEO_HDMLCD OPTION 198
7.1.160 OPTION_VIDEO_AMDELAN OPTION 199
7.1.161 OPTION_VIDEO_CUSTOMER OPTION 200
7.1.162 OPTION_VIDEO_DUPLICATE OPTION 201
7.1.163 OPTION_VIDEO_VIDEOMEM OPTION 202
7.1.164 OPTION_VIDEO_STDFONT OPTION 202
7.1.165 OPTION_VIDEO_MODE_REDIR OPTION 203
7.1.166 OPTION_CRITICAL_BOARD OPTION 203
7.1.167 OPTION_CRITICAL_BEEP OPTION 204
7.1.168 OPTION_CRITICAL_FLOPPY_LIGHT OPTION 204
7.1.169 OPTION_CRITICAL_MFGMODE OPTION 205
7.1.170 OPTION_CMOS_MOUSE OPTION 205
7.1.171 OPTION_CMOS_TEST1MB OPTION 205
7.1.172 OPTION_CMOS_TESTCLICK OPTION 206
7.1.173 OPTION_CMOS_PARITY OPTION 206
7.1.174 OPTION_CMOS_DELETE OPTION 207
7.1.175 OPTION_CMOS_HEXLOWER OPTION 207
7.1.176 OPTION_CMOS_F1ERROR OPTION 208
7.1.177 OPTION_CMOS_NUMLOCK OPTION 208
7.1.178 OPTION_CMOS_TYPEMATIC OPTION 208
7.1.179 OPTION_CMOS_WEITEK OPTION 209
7.1.180 OPTION_CMOS_FLOPPYSEEK OPTION 209
7.1.181 OPTION_CMOS_EXTCACHE OPTION 210
7.1.182 OPTION_CMOS_INTCACHE OPTION 210
7.1.183 OPTION_CMOS_FASTA20 OPTION 211
7.1.184 OPTION_CMOS_HDSEEK OPTION 211
7.1.185 OPTION_CMOS_CONFIGBOX OPTION 211
7.1.186 OPTION_CMOS_EXHMEMTEST OPTION 212
7.1.187 OPTION_CMOS_PASSWORD OPTION 212
7.1.188 OPTION_CMOS_KEYBOARD OPTION 213
7.1.189 OPTION_CMOS_SHADOW_ENABLE OPTION 213
7.1.190 OPTION_CMOS_SHADOW_C000 OPTION 214
7.1.191 OPTION_CMOS_SHADOW_C400 OPTION 214
7.1.192 OPTION_CMOS_SHADOW_C800 OPTION 215
7.1.193 OPTION_CMOS_SHADOW_CC00 OPTION 215
7.1.194 OPTION_CMOS_SHADOW_D000 OPTION 216
7.1.195 OPTION_CMOS_SHADOW_D400 OPTION 216
7.1.196 OPTION_CMOS_SHADOW_D800 OPTION 217
7.1.197 OPTION_CMOS_SHADOW_DC00 OPTION 217
7.1.198 OPTION_CMOS_SHADOW_E000 OPTION 218
7.1.199 OPTION_CMOS_SHADOW_E400 OPTION 218
7.1.200 OPTION_CMOS_SHADOW_E800 OPTION 218
7.1.201 OPTION_CMOS_SHADOW_EC00 OPTION 219
7.1.202 OPTION_CMOS_SHADOW_F000 OPTION 219
7.1.203 OPTION_CMOS_SPEED OPTION 220
7.1.204 OPTION_CMOS_REFRESH OPTION 220
7.1.205 OPTION_CMOS_POWER OPTION 221

7.1.206 OPTION_CMOS_ATA OPTION 221
7.1.207 OPTION_CMOS_RFD OPTION 222
7.1.208 OPTION_CMOS_LOAD_WINCE OPTION 222
7.1.209 OPTION_HARDERR_A20 OPTION 222
7.1.210 OPTION_HARDERR_DISSHADOW OPTION 223
7.1.211 OPTION_HARDERR_KBDCTRL OPTION 223
7.1.212 OPTION_HARDERR_CMOS OPTION 224
7.1.213 OPTION_HARDERR_PCI OPTION 224
7.1.214 OPTION_HARDERR_TIMER OPTION 225
7.1.215 OPTION_HARDERR_REFRESH OPTION 225
7.1.216 OPTION_HARDERR_MEMCFG OPTION 226
7.1.217 OPTION_HARDERR_BASEMEM OPTION 226
7.1.218 OPTION_HARDERR_DMA OPTION 227
7.1.219 OPTION_HARDERR_INT OPTION 227
7.1.220 OPTION_HARDERR_FIRMWARE OPTION 227
7.1.221 OPTION_HARDERR_KBD OPTION 228
7.1.222 OPTION_HARDERR_VIDEO OPTION 228
7.1.223 OPTION_HARDERR_PSWD OPTION 229
7.1.224 OPTION_HARDERR_LOWMEM OPTION 229
7.1.225 OPTION_HARDERR_PROTMODE OPTION 230
7.1.226 OPTION_SOFTERR_SETUP OPTION 230
7.1.227 OPTION_SOFTERR_LPT OPTION 231
7.1.228 OPTION_SOFTERR_MEMMIS OPTION 231
7.1.229 OPTION_SOFTERR_BOARD OPTION 231
7.1.230 OPTION_SOFTERR_CHIPSET OPTION 232
7.1.231 OPTION_SOFTERR_CPU OPTION 232
7.1.232 OPTION_QUERY_ENTERSETUP OPTION 232
7.1.233 OPTION_QUERY_FORMATRFD OPTION 233
7.1.234 OPTION_QUERY_VERIFYRFD OPTION 233
7.1.235 OPTION_QUERY_FORMATRAM OPTION 233
7.1.236 OPTION_QUERY_DEBUG OPTION 234
7.1.237 OPTION_MFGMODE_TIMEOUT OPTION 234
7.1.238 OPTION_MFGMODE_FIFO OPTION 235
7.1.239 OPTION_MEMTEST_LOW_POST OPTION 235
7.1.240 OPTION_MEMTEST_HIGH_POST OPTION 235
7.1.241 OPTION_MEMTEST_WAIT OPTION 236
7.1.242 OPTION_MEMTEST_CLEAR OPTION 236
7.1.243 OPTION_MEMTEST_CLICK OPTION 237
7.1.244 OPTION_MEMTEST_QUICK OPTION 237
7.1.245 OPTION_RFD_TESTFREE OPTION 237
7.1.246 OPTION_RFD_FAT_SNOOP OPTION 238
7.1.247 OPTION_DEBUG_HOTKEY OPTION 238
7.1.248 OPTION_DEBUG_FLASH OPTION 239
7.1.249 OPTION_DEBUG_WATCHINT OPTION 239
7.1.250 OPTION_DEBUG_NMI OPTION 240
7.1.251 OPTION_DEBUG_PCMCIA OPTION 240
7.1.252 OPTION_DEBUG_ASSEMBLY OPTION 241
7.1.253 OPTION_DEBUG_EDOSROM OPTION 241
7.1.254 OPTION_DEBUG_CHIPSET OPTION 241
7.1.255 OPTION_FLOPPY_SEEK OPTION 242

7.1.256 OPTION_FLOPPY_DMA OPTION 242
7.1.257 OPTION_FLOPPY_82077 OPTION 243
7.1.258 OPTION_FLOPPY_WATCHIO OPTION 244
7.1.259 OPTION_FLOPPY_FAST_POLL OPTION 244
7.1.260 OPTION_FLOPPY_POLL_ERRORS OPTION 244
7.1.261 OPTION_FLOPPY_144_ONLY OPTION 245
7.1.262 OPTION_IDE_RESET OPTION 245
7.1.263 OPTION_IDE_SEEK OPTION 246
7.1.264 OPTION_IDE_DISABLE_INTS OPTION 246
7.1.265 OPTION_IDE_SLOWDOWN OPTION 247
7.1.266 OPTION_IDE_POLLED OPTION 247
7.1.267 OPTION_IDE_AUTODETECT OPTION 247
7.1.268 OPTION_IDE_LBA OPTION 248
7.1.269 OPTION_IDE_CHS OPTION 249
7.1.270 OPTION_IDE_LBACMD OPTION 249
7.1.271 OPTION_IDE_BYTE_IO OPTION 250
7.1.272 OPTION_IDE_QUICK_DETECT OPTION 250
7.1.273 OPTION_BOOT_BEEP OPTION 250
7.1.274 OPTION_ BOOT_QUICK OPTION 251
7.1.275 OPTION_ BOOT_PRESERVE_WARM OPTION 251
7.1.276 OPTION_ BOOT_WARM_DELAY OPTION 252
7.1.277 OPTION_ CON_REDIR_WAIT OPTION 252
7.1.278 OPTION_ CON_REDIR_DISABLE OPTION 252
7.1.279 OPTION_ CON_REDIR_CANCEL OPTION 253
7.1.280 OPTION_ CON_REDIR_AUTO OPTION 253
7.1.281 OPTION_RTC_CMOS OPTION 254
7.1.282 OPTION_RTC_72421 OPTION 254
7.2 PARAMETERS FOUND IN CONFIG.INC 254
7.2.1 BIOS_DATE PARAMETER 255
7.2.2 BIOS_NAME CONSTANT 255
7.2.3 BIOS_RESERVED CONSTANT 255
7.2.4 CPU_TYPE PARAMETER 256
7.2.5 CPU_MHZ PARAMETER 256
7.2.6 CONFIG_BOARD_VERSION PARAMETER 257
7.2.7 CONFIG_POWER_ON_DELAY PARAMETER 257
7.2.8 CONFIG_CPU_DATA_BYTES PARAMETER 258
7.2.9 CONFIG_CS_DATA_BYTES PARAMETER 258
7.2.10 CONFIG_BOARD_DATA_BYTES PARAMETER 259
7.2.11 CONFIG_MAX_CMOS_LOCATIONS PARAMETER 259
7.2.12 CONFIG_START_BOARD_CMOS PARAMETER 259
7.2.13 CONFIG_START_CMOS_CACHE PARAMETER 260
7.2.14 CONFIG_CMOS_INDEX PARAMETER 261
7.2.15 CONFIG_CMOS_DATA PARAMETER 261
7.2.16 CONFIG_DEFAULT_RTC PARAMETER 262
7.2.17 CONFIG_CMOS_BOOT_0 PARAMETER 262
7.2.18 CONFIG_CMOS_BOOT_1 PARAMETER 263
7.2.19 CONFIG_CMOS_BOOT_2 PARAMETER 263
7.2.20 CONFIG_CMOS_BOOT_3 PARAMETER 264
7.2.21 CONFIG_CMOS_BOOT_4 PARAMETER 265
7.2.22 CONFIG_CMOS_BOOT_5 PARAMETER 266

7.2.23 CONFIG_CMOS_FLOPPY_0 PARAMETER....266
7.2.24 CONFIG_CMOS_FLOPPY_1 PARAMETER....267
7.2.25 CONFIG_CMOS_FLOPPY_2 PARAMETER....268
7.2.26 CONFIG_CMOS_FLOPPY_3 PARAMETER....268
7.2.27 CONFIG_CMOS_IDE_0 PARAMETER....269
7.2.28 CONFIG_CMOS_IDE_1 PARAMETER....271
7.2.29 CONFIG_CMOS_IDE_2 PARAMETER....271
7.2.30 CONFIG_CMOS_IDE_3 PARAMETER....272
7.2.31 CONFIG_CMOS_IDE0_CYL PARAMETER....273
7.2.32 CONFIG_CMOS_IDE0_HDS PARAMETER....273
7.2.33 CONFIG_CMOS_IDE0_SPT PARAMETER....274
7.2.34 CONFIG_CMOS_IDE1_CYL PARAMETER....274
7.2.35 CONFIG_CMOS_IDE1_HDS PARAMETER....275
7.2.36 CONFIG_CMOS_IDE1_SPT PARAMETER....275
7.2.37 CONFIG_CMOS_IDE2_CYL PARAMETER....276
7.2.38 CONFIG_CMOS_IDE2_HDS PARAMETER....276
7.2.39 CONFIG_CMOS_IDE2_SPT PARAMETER....277
7.2.40 CONFIG_CMOS_IDE3_CYL PARAMETER....277
7.2.41 CONFIG_CMOS_IDE3_HDS PARAMETER....278
7.2.42 CONFIG_CMOS_IDE3_SPT PARAMETER....278
7.2.43 CONFIG_CMOS_ASSIGN_A PARAMETER....279
7.2.44 CONFIG_CMOS_ASSIGN_B PARAMETER....279
7.2.45 CONFIG_CMOS_ASSIGN_C PARAMETER....280
7.2.46 CONFIG_CMOS_ASSIGN_D PARAMETER....280
7.2.47 CONFIG_CMOS_ASSIGN_E PARAMETER....281
7.2.48 CONFIG_CMOS_ASSIGN_F PARAMETER....281
7.2.49 CONFIG_CMOS_ASSIGN_G PARAMETER....282
7.2.50 CONFIG_CMOS_ASSIGN_H PARAMETER....283
7.2.51 CONFIG_CMOS_ASSIGN_I PARAMETER....283
7.2.52 CONFIG_CMOS_ASSIGN_J PARAMETER....284
7.2.53 CONFIG_CMOS_ASSIGN_K PARAMETER....284
7.2.54 CONFIG_CMOS_TYPEMATIC_DELAY PARAMETER....285
7.2.55 CONFIG_CMOS_TYPEMATIC_RATE PARAMETER....285
7.2.56 CONFIG_CMOS_FLOPPY_RETRY PARAMETER....286
7.2.57 CONFIG_CMOS_EQUIP PARAMETER....287
7.2.58 CONFIG_BOOT_ATTEMPT PARAMETER....287
7.2.59 CONFIG_WAIT_8042 PARAMETER....287
7.2.60 CONFIG_WAIT_8042_INIT PARAMETER....288
7.2.61 CONFIG_SETTLE_8042 PARAMETER....288
7.2.62 CONFIG_WAIT_COUNT PARAMETER....288
7.2.63 CONFIG_WAIT_LPT PARAMETER....289
7.2.64 CONFIG_WAIT_IDE_INIT PARAMETER....289
7.2.65 CONFIG_WAIT_IDE_IO PARAMETER....290
7.2.66 CONFIG_WAIT_PROGRESS_COM PARAMETER....290
7.2.67 CONFIG_SERIAL_TIMEOUT PARAMETER....290
7.2.68 CONFIG_PARALLEL_TIMEOUT PARAMETER....291
7.2.69 CONFIG_POST_PROGRESS_PORT PARAMETER....291
7.2.70 CONFIG_POST_PROGRESS_COM PARAMETER....291
7.2.71 CONFIG_POST_PROGRESS_BAUD PARAMETER....292
7.2.72 CONFIG_MFG_PROGRESS_PORT PARAMETER....292

7.2.73 CONFIG_MAX_LOW_MEMORY PARAMETER 293
7.2.74 CONFIG_TESTBASE_SIZE PARAMETER 293
7.2.75 CONFIG_MAX_EXT_MEMORY PARAMETER 294
7.2.76 CONFIG_EXTRA_SEGMENT PARAMETER 294
7.2.77 CONFIG_FSINIT_SEGMENT PARAMETER 294
7.2.78 CONFIG_DEFAULT_EQUIP_BYTE PARAMETER 295
7.2.79 CONFIG_VIDEO_ROM_SCAN PARAMETER 295
7.2.80 CONFIG_LOW_ROM_SCAN PARAMETER 295
7.2.81 CONFIG_HIGH_ROM_SCAN PARAMETER 296
7.2.82 CONFIG_ROM_SCAN_INTERVAL PARAMETER 297
7.2.83 CONFIG_MINI_DOS_SCAN PARAMETER 297
7.2.84 CONFIG_PCI_ROM_SHADOW_START PARAMETER 298
7.2.85 CONFIG_VIDEO_SEG_GRAPHIC PARAMETER 298
7.2.86 CONFIG_VIDEO_SEG_MONO PARAMETER 299
7.2.87 CONFIG_VIDEO_SEG_COLOR PARAMETER 299
7.2.88 CONFIG_BEEP_LENGTH PARAMETER 299
7.2.89 CONFIG_BEEP_CYCLE PARAMETER 300
7.2.90 CONFIG_BEEP_8254_TONE PARAMETER 300
7.2.91 CONFIG_PCMCIA_IOBASE PARAMETER 301
7.2.92 CONFIG_RFDDISK_KBBLKSIZE PARAMETER 301
7.2.93 CONFIG_FLASH_DATASEG PARAMETER 302
7.2.94 CONFIG_FLASH_CODESEG PARAMETER 302
7.2.95 CONFIG_PAGED_MEM_SEG PARAMETER 303
7.2.96 CONFIG_VPP_TIMEOUT_IN_TICKS PARAMETER 303
7.2.97 CONFIG_PCI_ROM_MAP PARAMETER 304
7.2.98 CONFIG_PCI_MEM_AVAIL PARAMETER 304
7.2.99 CONFIG_PCI_IO_PORT_BASE PARAMETER 305
7.2.100 CONFIG_PCI_IO_ALLOC PARAMETER 306
7.2.101 CONFIG_PCI_IO_TMP_TBL_SEG PARAMETER 306
7.2.102 CONFIG_PCI_BM_OFFSET PARAMETER 306
7.2.103 CONFIG_PCI_MMIO_AVAIL PARAMETER 307
7.2.104 CONFIG_PCI_LATENCY PARAMETER 307
7.2.105 CONFIG_PCI_IRQ_BITMAP PARAMETER 307
7.2.106 CONFIG_PS2_MOUSE_IRQ PARAMETER 308
7.2.107 CONFIG_PS2_MOUSE_LOOP PARAMETER 308
7.2.108 CONFIG_IDE_PORT_BASE PARAMETER 308
7.2.109 CONFIG_IDE_PORT_ALT_STATUS PARAMETER 309
7.2.110 CONFIG_IDE_PORT_CTRL PARAMETER 309
7.2.111 LPT1_BASE PARAMETER 309
7.2.112 LPT2_BASE PARAMETER 310
7.2.113 LPT3_BASE PARAMETER 310
7.2.114 COM1_BASE PARAMETER 310
7.2.115 COM2_BASE PARAMETER 311
7.2.116 COM3_BASE PARAMETER 311
7.2.117 COM4_BASE PARAMETER 311
7.2.118 COM1_INIT PARAMETER 312
7.2.119 COM2_INIT PARAMETER 312
7.2.120 COM3_INIT PARAMETER 312
7.2.121 COM4_INIT PARAMETER 313
7.2.122 MFG_COM_BASE PARAMETER 313

7.2.123 MFG_INT_VECT PARAMETER....314
7.2.124 CONFIG_MFG_BAUD PARAMETER....314
7.2.125 MFG_EOI_PORT PARAMETER....315
7.2.126 MFG_EOI_CMD PARAMETER....315
7.2.127 CONFIG_MFG_BUFSIZE PARAMETER....316
7.2.128 CONFIG_MFG_CBSIZE PARAMETER....316
7.2.129 CONFIG_MFG_TIMEOUT PARAMETER....317
7.2.130 CONFIG_CON_REDIR_STD PARAMETER....317
7.2.131 CONFIG_CON_REDIR_DEBUG PARAMETER....318
7.2.132 CONFIG_CON_REDIR_SETUP PARAMETER....318
7.2.133 BIOS_HDWR PARAMETER....319
7.2.134 BIOS_HDWR_SUB PARAMETER....320
7.2.135 DEBUG_CMDBUF_LEN PARAMETER....320
7.2.136 DEBUG_MAX_BREAKPOINTS PARAMETER....321
7.2.137 DEBUG_MAX_BKPT_CMD_LEN PARAMETER....321
7.2.138 CONFIG_WINCE_ENTRY PARAMETER....321
7.2.139 CONFIG_WINCE_VIDEO PARAMETER....322
7.2.140 CONFIG_WINCE_PORT PARAMETER....322
7.2.141 CONFIG_WINCE_BAUD PARAMETER....323
7.2.142 CONFIG_WINCE_PCI PARAMETER....323
7.2.143 CONFIG_CFGBOX_MONO_ATTRIB PARAMETER....323
7.2.144 CONFIG_CFGBOX_COLOR_ATTRIB PARAMETER....324
7.2.145 CONFIG_DELAY_ADD PARAMETER....324
7.2.146 CONFIG_DELAY_MULTIPLY PARAMETER....324
7.2.147 CONFIG_DELAY_IO PARAMETER....325
7.2.148 CONFIG_SPLASH_VMODE PARAMETER....325
7.2.149 CONFIG_SPLASH_WBYTES PARAMETER....326
7.2.150 CONFIG_SPLASH_HEIGHT PARAMETER....326
7.2.151 CONFIG_SPLASH_COLORS PARAMETER....327
7.2.152 CONFIG_SPLASH_SEG PARAMETER....327
7.2.153 CONFIG_SPLASH_BOOTS PARAMETER....328
7.2.154 SPLASH_TABLE TABLE....329
7.2.155 POWER_DEVID (POWER MANAGEMENT) TABLE....330
7.2.156 MEDIA_REGION (MEDIA MANAGEMENT) TABLE....332
7.2.157 FILE_SYSTEM (INT 13H DRIVE MANAGEMENT) TABLE....334
7.2.158 LOAD_IMAGE (WINDOWS CE BOOTABILITY) TABLE....336
7.2.159 PCI_ROM CONFIGURATION TABLE....337
7.2.160 RELOCATE_FEATURE CONFIGURATION TABLE....338

STEP-BY-STEP BIOS ADAPTATION....341

8.1 THE PROJECT CONCEPT....341
8.2 SELECTING THE BEST STARTING POINT....344
8.3 DETERMINING WHAT NEEDS TO BE CUSTOMIZED....345
8.3.1 PROJECT FILE SYMBOL OVERRIDES....345
8.3.2 GENERAL PURPOSE PACKAGE PIN ASSIGNMENTS....345
8.3.3 DEFINING POWER CONTROL....345
8.3.4 WATCHDOG TIMER....346
8.3.5 PCI INTERRUPT MAPPING....346

8.3.6 SUPER I/O PROGRAMMING ... 346
8.4 BUILDING THE BIOS ... 347
8.5 GETTING THROUGH POST ... 348
8.5.1 USING THE SPEAKER TO REPORT POST FAILURES ... 348
8.5.2 USING POST CODES TO REPORT POST FAILURES ... 348
8.5.3 USING A SERIAL PORT TO REPORT POST FAILURES ... 348
8.5.4 USING THE GRAPHICAL POST TEST ICONS ... 349
8.5.5 ATTEMPT TO BOOT AN OPERATING SYSTEM (DOS) ... 349
8.5.6 WHEN NOTHING HAPPENS 349

PART II ... 351

BIOS FEATURES ... 351

THE INTEGRATED BIOS DEBUGGER ... 353

9.1 HOW TO USE THE DEBUGGER ... 353
9.2 DEBUGGER COMMAND SYNTAX ... 354
9.2.1 OPERAND TYPES ... 354
9.2.2 EXPRESSIONS ... 354
9.2.3 ADDRESSES ... 355
9.3 COMMAND REFERENCE ... 356
9.3.1 ? COMMAND ... 356
9.3.2 + COMMAND ... 356
9.3.3 - COMMAND ... 357
9.3.4 BC COMMAND ... 357
9.3.5 BIOSDATA COMMAND ... 357
9.3.6 BL COMMAND ... 358
9.3.7 BP COMMAND ... 358
9.3.8 CIS COMMAND ... 359
9.3.9 CONSOLE COMMAND ... 359
9.3.10 CSR COMMAND ... 360
9.3.11 CSW COMMAND ... 361
9.3.12 D COMMAND ... 361
9.3.13 DA20 COMMAND ... 362
9.3.14 DB COMMAND ... 362
9.3.15 DCACHE COMMAND ... 363
9.3.16 DD COMMAND ... 363
9.3.17 DW COMMAND ... 364
9.3.18 E COMMAND ... 364
9.3.19 EA20 COMMAND ... 365
9.3.20 ECACHE COMMAND ... 365
9.3.21 EFL COMMAND ... 365
9.3.22 G COMMAND ... 366
9.3.23 HELP COMMAND ... 366

9.3.24 I COMMAND 367
9.3.25 ID COMMAND..367
9.3.26 IW COMMAND..367
9.3.27 LFL COMMAND..368
9.3.28 MASK COMMAND 368
9.3.29 MODE COMMAND 369
9.3.30 O COMMAND..369
9.3.31 OD COMMAND 370
9.3.32 OW COMMAND 370
9.3.33 PCID COMMAND..371
9.3.34 PCIR COMMAND 371
9.3.35 PCIRB COMMAND 372
9.3.36 PCIRW COMMAND 372
9.3.37 PCIRD COMMAND 373
9.3.38 PCIW COMMAND..374
9.3.39 PCIWB COMMAND 374
9.3.40 PCIWW COMMAND 375
9.3.41 PCIWD COMMAND 376
9.3.42 R COMMAND 376
9.3.43 R16 COMMAND 377
9.3.44 R32 COMMAND 377
9.3.45 R32X COMMAND 377
9.3.46 RC COMMAND 378
9.3.47 RD COMMAND 378
9.3.48 RDMSR COMMAND..379
9.3.49 REBOOT COMMAND..379
9.3.50 RFL COMMAND 380
9.3.51 SFL COMMAND..381
9.3.52 SIOR COMMAND..381
9.3.53 SIOW COMMAND..382
9.3.54 SO COMMAND..382
9.3.55 T COMMAND 383
9.3.56 TIME COMMAND..383
9.3.57 TORAM COMMAND 384
9.3.58 TOROM COMMAND 385
9.3.59 U COMMAND..385
9.3.60 U16 COMMAND..386
9.3.61 U32 COMMAND..386
9.3.62 UFL COMMAND 387
9.3.63 V COMMAND..387
9.3.64 WATCH COMMAND 388
9.3.65 WC COMMAND 389
9.3.66 WCOMX COMMAND..389
9.3.67 WD COMMAND 390
9.3.68 WFL COMMAND 390
9.3.69 WP COMMAND..391
9.3.70 WRMSR COMMAND..391
9.4 PRINTF OUTPUT FORMATTING MACRO..392
9.4.1 LITERAL SPECIFICATIONS 393
9.4.2 FORMAT SPECIFICATIONS 393

9.4.2.1 $c Format Specification 394
9.4.2.2 $b Format Specification 394
9.4.2.3 $x Format Specification 394
9.4.2.4 $u Format Specification 395
9.4.2.5 $d Format Specification 395
9.4.2.6 $lx Format Specification 395
9.4.2.7 $lu Format Specification 395
9.4.2.8 $ld Format Specification 395
9.4.2.9 $s Format Specification 396
9.4.2.10 s Format Specification 397
9.4.2.11 $s[n] Format Specification 397

THE BIOS POST INTERFACE 399

10.1 LEGACY POST INTERFACE 399
10.1.1 POST MESSAGES 399
10.1.2 CRITICAL ERRORS 400
10.1.3 SOFT ERRORS 400
10.1.4 CONSOLE REDIRECTION 401
10.2 GRAPHICAL POST INTERFACE 401
10.2.1 SPLASH SCREENS 402
10.2.2 POST PROGRESS ICONS 402
10.2.3 STILL AND ANIMATED BITMAPS 403
10.2.4 CONFIGURATION 403
10.2.4.1 The SPLASH_TABLE Table 403
10.2.4.2 Graphical Resources 405
10.2.4.3 Creating Bitmaps and Icons 406
10.2.4.4 Creating Timed Sequences 408
10.2.4.5 Configuring the Graphics Driver Software 408

PCI SUBSYSTEM 409

11.1 OVERVIEW 409
11.2 PCI SERVICES 410
11.3 THE 32-BIT PCI BUILD PROCESS 410
11.4 CONFIGURING PCI IN THE PROJECT FILE 412
11.5 CONFIGURING PCI IN THE BOARD PERSONALITY MODULE 413
11.5.1 PCI INTERRUPT ROUTING TABLE 413
11.5.2 BOARD PERSONALITY MODULE ROUTINES 415
11.5.3 CHIPSET PERSONALITY MODULE ROUTINES 415

DISK FILE SYSTEM MANAGEMENT 417

12.1 FILE SYSTEM CONTROL LAYER 417
12.1.1 FSCL ARCHITECTURE 418
12.1.2 FILE SYSTEM TYPES 418
12.1.3 FILE_SYSTEM TABLE 419

12.1.4 FSCL DATA STRUCTURES 420
12.1.4.1 FS_BASE Structure 421
12.1.4.2 FS_UNIT Structure 421
12.1.4.3 FS_PACKET Structure 422
12.1.5 FSHLP API 423
12.1.5.1 FsHlpInit Function 424
12.1.5.2 FsHlpFind Function 424
12.2 FILE SYSTEM DRIVERS 425
12.2.1 FSD ARCHITECTURE 425
12.2.2 FSD ENTRYPOINT 426
12.3 FLOPPY DISK DRIVE SUPPORT 428
12.3.1 ENABLING FLOPPY DISK SUPPORT IN THE BUILD 428
12.3.2 CONFIGURING FLOPPY DISKS IN SETUP 428
12.3.3 TUNING THE FLOPPY DISK DRIVER 429
12.3.3.1 82077 FIFOs 429
12.3.3.2 Seek During Boot 429
12.3.3.3 Debugging Polled I/O 429
12.3.3.4 DMA or Polled Data Transfers 429
12.4 HARD DISK (IDE/ATA) SUPPORT 430
12.4.1 ENABLING IDE/ATA DISK SUPPORT IN THE BUILD 430
12.4.2 CONFIGURING IDE/ATA DISKS IN SETUP 431
12.4.3 TUNING THE IDE/ATA DISK DRIVER 431
12.4.3.1 Drive Autodetection 431
12.4.3.2 Drive Geometry Translation (LBA and CHS) 431
12.4.3.3 Polled .vs. Interrupt-Driven I/O Completion 431
12.4.3.4 Disabling Interrupts During Transfers 432
12.4.3.5 Slowing Down I/O Transfers 432
12.4.3.6 Drive Reset During POST 432
12.4.3.7 Drive Seek During POST 432
12.5 "EL TORITO" CD-ROM SUPPORT 432
12.5.1 ENABLING CD-ROM SUPPORT IN THE BUILD 432
12.5.2 CONFIGURING CD-ROM DRIVES IN SETUP 433
12.6 EMULATING DISKS WITH ROM 433
12.6.1 ENABLING THE ROM DISK SUPPORT OPTIONS 434
12.6.2 ENABLING THE ROM DISK IN SETUP 434
12.6.3 BUILDING A ROM DISK IMAGE 435
12.6.4 TROUBLESHOOTING THE ROM DISK 436
12.6.5 USING PAGED OR WINDOWED ROM DISKS 439
12.7 EMULATING DISKS WITH RAM 440
12.7.1 ENABLING THE RAM DISK SUPPORT OPTIONS 440
12.7.2 ENABLING THE RAM DISK IN SETUP 441
12.7.3 INITIALIZING THE RAM DISK 441
12.7.4 TROUBLESHOOTING THE RAM DISK 441
12.8 EMULATING DISKS WITH FLASH 444
12.8.1 ENABLING THE RFD SUPPORT OPTIONS 445
12.8.2 PROTECTED MODE .VS. WINDOWING ACCESS TO FLASH 446
12.8.3 ENABLING THE RFD IN SETUP 447
12.8.4 TESTING THE FLASH ARRAY 447
12.8.5 INITIALIZING THE RFD (LOW-LEVEL FORMATTING) 450
12.8.6 USING DOS TO FDISK A HARD-FORMATTED RFD 451

12.8.7 USING DOS TO FORMAT THE RFD....451
12.8.8 USING MANUFACTURING MODE TO FORMAT THE RFD....452

DRIVERS FOR FLASH AND OTHER MEDIA 453

13.1 MEDIA CONTROL LAYER....453
13.1.1 MCL ARCHITECTURE....454
13.1.1.1 Media Types....454
13.1.1.2 Media Addressing 455
13.1.1.3 Vpp Control....457
13.1.1.4 Interrupt Latency 457
13.1.2 MCL ENTRYPOINTS 458
13.1.2.1 MediaPwrLevel Entrypoint....458
13.1.2.2 MediaLockBlock Procedure....459
13.1.2.3 MediaStartErase Procedure....460
13.1.2.4 MediaEraseComplete Procedure....460
13.1.2.5 MediaReadBlock Procedure....461
13.1.2.6 MediaWriteBlock Procedure....461
13.1.2.7 MediaQuery Procedure 462
13.1.3 MTDHLP API....463
13.1.3.1 MtdHlpToProt API Function 464
13.1.3.2 MtdHlpToReal API Function....464
13.1.3.3 MtdHlpMapAddress API Function 465
13.1.3.4 MtdHlpMapReal API Function....465
13.1.3.5 MtdHlpQueryRegion API Function 466
13.1.3.6 MtdHlpDelay API Function 466
13.1.3.7 MtdHlpEnableVpp API Function....466
13.1.3.8 MtdHlpDisableVpp API Function 467
13.1.3.9 MtdHlpRead API Function 468
13.2 MEDIA TECHNOLOGY DRIVERS (MTDS)....468
13.2.1 MTD ARCHITECTURE 468
13.2.2 MTD ENTRYPOINTS....469
13.2.2.1 MTD Request Entrypoint....469
13.2.2.2 MTD Power Management Entrypoint....470
13.2.3 DISPATCHING TO FUNCTION HANDLERS 470
13.2.4 PROTECTED-MODE AND REAL-MODE CONTROL PATHS....471
13.2.5 ADDING A CUSTOM MTD TO THE BOARD PERSONALITY MODULE....473
13.2.6 ADDING WINDOWING TO THE BOARD PERSONALITY MODULE 474
13.2.7 MTD I/O REQUEST INTERFACE 474
13.2.7.1 LockBlock MTD Procedure 474
13.2.7.2 StartErase MTD Procedure 475
13.2.7.3 EraseComplete MTD Procedure 476
13.2.7.4 ReadBlock MTD Procedure 476
13.2.7.5 WriteBlock MTD Procedure 477
13.2.7.6 Query MTD Procedure....478
13.2.7.7 Init MTD Procedure 479
13.3 MEDIA_REGION ADDRESSING TABLE....479
13.4 COMMON FLASH DEVICE LAYOUT 480

MANUFACTURING MODE....483

14.1 ENTERING MANUFACTURING MODE....483
14.2 HOST PC OPERATION....484
14.2.1 SAMPLE MANUFACTURING MODE HOST PROGRAM....484
14.2.2 MANUFACTURING MODE DRIVE REDIRECTION....485
14.3 HOST-SIDE MANUFACTURING MODE FUNCTIONS....486
14.3.1 MSGINITIALIZE FUNCTION....486
14.3.2 MSGDEINITIALIZE FUNCTION....487
14.3.3 MSGPINGTARGET FUNCTION....487
14.3.4 MSGRECEIVE FUNCTION....487
14.3.5 MSGSEND FUNCTION....488
14.3.6 MSGBOOTTARGET FUNCTION....488
14.3.7 MSGGETLASTPOSTCODE FUNCTION....489
14.3.8 MSGCHECKSUM FUNCTION....489
14.3.9 MSGTESTMEMORY FUNCTION....489
14.3.10 MSGREADFLASH FUNCTION....490
14.3.11 MSGWRITEFLASH FUNCTION....491
14.3.12 MSGREADBUFFER FUNCTION....491
14.3.13 MSGWRITEBUFFER FUNCTION....492
14.3.14 MSGLOCKFLASH FUNCTION....493
14.3.15 MSGERASEFLASH FUNCTION....493
14.3.16 MSGINT13 FUNCTION....494

ADVANCED POWER MANAGEMENT....497

15.1 APM SYSTEM MODEL....497
15.2 APM SOFTWARE LAYERS....498
15.3 APM BIOS INTERFACE....499
15.4 POWER MANAGEMENT SUBSYSTEM (PMS)....500
15.4.1 POWER_DEVID DEVICE TREE....500
15.4.2 DEVICE POWER CONTROL ENTRYPOINTS....501

SETUP AND DIAGNOSTICS SYSTEM....503

16.1 SETUP BUILD OPTIONS....503
16.2 ENTERING SETUP....504
16.3 SETUP SCREENS....504
16.3.1 BASIC CMOS CONFIGURATION SCREEN....505
16.3.1.1 CONFIGURING DRIVE ASSIGNMENTS....505
16.3.1.2 CONFIGURING FLOPPY DRIVE TYPES....506
16.3.1.3 CONFIGURING IDE DRIVE TYPES....506
16.3.1.4 CONFIGURING BOOT ACTIONS....506
16.3.2 CUSTOM CONFIGURATION SETUP SCREEN....507
16.3.3 SHADOW CONFIGURATION SETUP SCREEN....508
16.3.4 STANDARD DIAGNOSTIC ROUTINES SETUP SCREEN....508
16.3.5 START SYSTEM BIOS DEBUGGER SETUP SCREEN....509
16.3.6 START RS232 MANUFACTURING LINK SETUP SCREEN....509

16.3.7 OTHER PRE-BOOT SETUP SCREENS 510

POWER ON SELF TEST (POST) 513

17.1 INITIALIZATION WITHOUT A STACK OR RAM 513
17.1.1 STACK-BASED PROCEDURES 514
17.1.2 REGISTER-BASED COROUTINES 514
17.1.3 HYBRID PROCEDURES WITH DUAL LINKAGE 514
17.2 EARLY INITIALIZATION PROCESS 515
17.2.1 BOARDINIT0 PROCESSING 515
17.2.2 POSTTESTRESETVALUE PROCESSING 515
17.2.3 POSTCODECOMINIT PROCESSING 515
17.2.4 BOARDINIT1 PROCESSING 516
17.2.5 MAIN POST PROCESSING 516
17.2.6 BOARDMEMCONFIG PROCESSING 516
17.2.7 FURTHER POST PROCESSING 517
17.2.8 BOARDINIT4 PROCESSING 517
17.2.9 BOARDINIT6 PROCESSING 517
17.2.10 DEVICE INITIALIZATION PROCESSING 518
17.2.11 FINAL POST, BOARDINIT8 PROCESSING 518
17.3 POST CODES 518
17.3.1 SPEAKER POST CODES 518
17.3.2 VIDEO POST MESSAGES 519

CPU PERSONALITY MODULES 521

18.1 HOW CPM OVERRIDE ROUTINES WORK 521
18.2 HOW CPMS ARE PACKAGED IN FILES 522
18.3 OTHER CPU PERSONALITY MODULES 522
18.4 THE CPM INTERFACE 522
18.4.1 CPUBEEP ROUTINE 523
18.4.2 CPUDISABLEA20 HYBRID 524
18.4.3 CPUDISABLECACHE PROCEDURE 524
18.4.4 CPUDISABLEDMACTRL ROUTINE 525
18.4.5 CPUDISABLEINTCTRL HYBRID 525
18.4.6 CPUDISABLEWATCHDOG PROCEDURE 526
18.4.7 CPUENABLEA20 HYBRID 526
18.4.8 CPUENABLECACHE PROCEDURE 527
18.4.9 CPUENABLEAPM PROCEDURE 528
18.4.10 CPUENABLEDMACTRL ROUTINE 528
18.4.11 CPUENABLEINTCTRL HYBRID 529
18.4.12 CPUENABLEWATCHDOG PROCEDURE 529
18.4.13 CPUEOI PROCEDURE 530
18.4.14 CPUEXTRWCTRL PROCEDURE 530
18.4.15 CPUFLOPPYDMA PROCEDURE 531
18.4.16 CPUGETPROCESSORNAME PROCEDURE 532
18.4.17 CPUGETPROCESSORTYPE PROCEDURE 532
18.4.18 CPUHOOKVECTORS PROCEDURE 533

18.4.19 CPUINIT0 ROUTINE 533
18.4.20 CPUINIT1 ROUTINE 534
18.4.21 CPUINITDMA ROUTINE 535
18.4.22 CPUINITINTCTRL ROUTINE 535
18.4.23 CPUINITPARALLEL ROUTINE 536
18.4.24 CPUINITREFRESH ROUTINE 536
18.4.25 CPUINITSERBIOS PROCEDURE 537
18.4.26 CPUINITSERIAL ROUTINE 538
18.4.27 CPUINITTIMER ROUTINE 538
18.4.28 CPUINITWATCHDOG ROUTINE 539
18.4.29 CPUKICKWATCHDOG PROCEDURE 539
18.4.30 CPUPWRLVL PROCEDURE 540
18.4.31 CPUSERGETCH PROCEDURE 541
18.4.32 CPUSERGETSTATUS PROCEDURE 541
18.4.33 CPUSERINIT PROCEDURE 542
18.4.34 CPUSERINITEXT PROCEDURE 543
18.4.35 CPUSERPUTCH PROCEDURE 544
18.4.36 CPUSETFASTSPEED PROCEDURE 545
18.4.37 CPUSETSLOWSPEED PROCEDURE 546
18.4.38 CPUSTARTDMA PROCEDURE 546
18.4.39 CPUTESTSYNCIO ROUTINE 547
18.4.40 CPUUNMASKINT PROCEDURE 547

CHIPSET PERSONALITY MODULES 549

19.1 HOW CSPM OVERRIDE ROUTINES WORK 550
19.2 HOW CSPMS ARE PACKAGED IN FILES 550
19.3 OTHER CHIPSET PERSONALITY MODULES 551
19.4 THE CSPM INTERFACE 551
19.4.1 CSASSIGNPCIIRQ PROCEDURE 551
19.4.2 CSDISABLEA20 PROCEDURE 552
19.4.3 CSDISABLECACHE PROCEDURE 552
19.4.4 CSDISABLESHADOW PROCEDURE 553
19.4.5 CSDISABLEWATCHDOG PROCEDURE 553
19.4.6 CSDISPLAYCHIPSET PROCEDURE 554
19.4.7 CSENABLEA20 PROCEDURE 554
19.4.8 CSENABLEAPM PROCEDURE 555
19.4.9 CSENABLECACHE PROCEDURE 555
19.4.10 CSENABLEWATCHDOG PROCEDURE 556
19.4.11 CSGETPCIINFO PROCEDURE 556
19.4.12 CSINIT0 ROUTINE 557
19.4.13 CSINIT1 ROUTINE 557
19.4.14 CSINITREFRESH ROUTINE 558
19.4.15 CSINITWATCHDOG ROUTINE 559
19.4.16 CSKICKWATCHDOG ROUTINE 559
19.4.17 CSMAPADDRESS PROCEDURE 560
19.4.18 CSMEMCONFIG ROUTINE 561
18.4.19 CSPWRLVL PROCEDURE 561
19.4.20 CSREADREG PROCEDURE 562

19.4.21 CsReboot Procedure 562
19.4.22 CsSetFastSpeed Procedure 563
19.4.23 CsSetSlowSpeed Procedure 563
19.4.24 CsShadowArea Procedure 564
19.4.25 CsShadowWriteCtl Procedure 565
19.4.26 CsTimerTick Procedure 566
19.4.27 CsUnMapAddress Procedure 566
19.4.28 CsWriteReg Procedure 567

BOARD PERSONALITY MODULES 569

20.1 How BPM Override Routines Work 569
20.2 How BPMs are Packaged in Files 570
20.3 Other Board Personality Modules 570
20.4 The BPM Interface 571
20.4.1 BoardApmMode Procedure 571
20.4.2 BoardAssignPciIrq Procedure 572
20.4.3 BoardAutoRedirect Procedure 572
20.4.4 BoardBeep Routine 573
20.4.5 BoardDelayUsec Routine 573
20.4.6 BoardDisableA20 Hybrid 574
20.4.7 BoardDisableCache Procedure 575
20.4.8 BoardDisableDmaCtrl Routine 575
20.4.9 BoardDisableIntCtrl Hybrid 576
20.4.10 BoardDisableShadow Procedure 576
20.4.11 BoardDisableTestMode Procedure 577
20.4.12 BoardDisableWatchdog Procedure 577
20.4.13 BoardDisableWrites Procedure 578
20.4.14 BoardEnableA20 Hybrid 578
20.4.15 BoardEnableApm Procedure 579
20.4.16 BoardEnableCache Procedure 579
20.4.17 BoardEnableDmaCtrl Routine 580
20.4.18 BoardEnableIntCtrl Hybrid 580
20.4.19 BoardEnablePciRegion Procedure 581
20.4.20 BoardEnableWatchdog Procedure 582
20.4.21 BoardEnableWrites Procedure 582
20.4.22 BoardEoi Procedure 583
20.4.23 BoardFloppyDma Procedure 583
20.4.24 BoardFsInit Procedure 584
20.4.25 BoardGetPciInfo Procedure 585
20.4.26 BoardHelp1 Procedure 585
20.4.27 BoardHelp2 Procedure 586
20.4.28 BoardIdeAutoDetect Procedure 586
20.4.29 BoardInit0 Routine 587
20.4.30 BoardInit1 Routine 588
20.4.31 BoardInit4 Procedure 588
20.4.32 BoardInit6 Procedure 589
20.4.33 BoardInit8 Procedure 589
20.4.34 BoardInitAppRom Routine 590

20.4.35 BOARDINITDMA ROUTINE 590
20.4.36 BOARDINITFIELDS PROCEDURE 591
20.4.37 BOARDINITINTCTRL ROUTINE 591
20.4.38 BOARDINITREFRESH ROUTINE....592
20.4.39 BOARDINITTIMER ROUTINE....593
20.4.40 BOARDINITWATCHDOG ROUTINE....593
20.4.41 BOARDKICKWATCHDOG PROCEDURE....594
20.4.42 BOARDMAPADDRESS PROCEDURE....594
20.4.43 BOARDPCICONTROL PROCEDURE....595
20.4.44 BOARDPCIREADSCRATCH PROCEDURE....598
20.4.45 BOARDPCIWRITESCRATCH PROCEDURE 599
20.4.46 BOARDPOSTCODECOM ROUTINE....599
20.4.47 BOARDPOSTCODECOMINIT ROUTINE....601
20.4.48 BOARDMEMCONFIG ROUTINE 601
20.4.49 BOARDPWRLVL PROCEDURE....602
20.4.50 BOARDPOSTERROR ROUTINE 603
20.4.51 BOARDREBOOT PROCEDURE....603
20.4.52 BOARDRESETCMOS ROUTINE....604
20.4.53 BOARDSAVECMOS PROCEDURE 604
20.4.54 BOARDSAVEFIELDS PROCEDURE....605
20.4.55 BOARDSETFASTSPEED PROCEDURE 605
20.4.56 BOARDSETSLOWSPEED PROCEDURE....606
20.4.57 BOARDSETVIDEOMODE PROCEDURE....606
20.4.58 BOARDSIOREADREG PROCEDURE 607
20.4.59 BOARDSIOWRITEREG PROCEDURE 607
20.4.60 BOARDSHADOWAREA PROCEDURE 608
20.4.61 BOARDTESTMODE PROCEDURE....608
20.4.62 BOARDTIMERTICK PROCEDURE 609
20.4.63 BOARDUNMAPADDRESS PROCEDURE....609
20.4.64 BOARDUNMASKINT PROCEDURE....610

PART III 613

BIOS FUNCTION REFERENCE 615

21.1 INT 10H, VIDEO BIOS SERVICES 615
21.1.1 SET VIDEO MODE (00H)....615
21.1.2 SET CURSOR SIZE (01H)....616
21.1.3 SET CURSOR POSITION (02H) 616
21.1.4 READ CURSOR POSITION (03H)....617
21.1.5 READ LIGHT PEN POSITION (04H)....617
21.1.6 SELECT VIDEO PAGE (05H) 617
21.1.7 SCROLL UP WINDOW (06H) 618
21.1.8 SCROLL DOWN WINDOW (07H)....618
21.1.9 READ CHAR/ATTR FROM SCREEN (08H)....619
21.1.10 WRITE CHAR/ATTR TO SCREEN (09H) 619
21.1.11 WRITE CHARACTER TO SCREEN (0AH)....619

21.1.12 SET COLOR PALETTE (0BH) 620
21.1.13 WRITE PIXEL (0CH) 620
21.1.14 READ PIXEL (0DH) 620
21.1.15 WRITE TELETYPE MODE (0EH) 621
21.1.16 RETURN VIDEO STATUS (0FH) 621
21.2 INT 11H, EQUIPMENT LIST SERVICE 621
21.3 INT 12H, LOW MEMORY SIZE SERVICE 622
21.4 INT 13H, DISK SERVICES 622
21.4.1 RESET (00H) 623
21.4.2 READ STATUS (01H) 623
21.4.3 READ SECTORS (02H) 624
21.4.4 WRITE SECTORS (03H) 624
21.4.5 VERIFY SECTORS (04H) 625
21.4.6 FORMAT TRACK (05H) 625
21.4.7 READ DRIVE PARAMETERS (08H) 626
21.4.8 INITIALIZE HARD DISK CONTROLLER (09H) 626
21.4.9 READ LONG SECTORS (0AH) 627
21.4.10 WRITE LONG SECTORS (0BH) 627
21.4.11 SEEK TO CYLINDER (0CH) 628
21.4.12 RESET HARD DISK CONTROLLER (0DH) 628
21.4.13 TEST DRIVE READY (10H) 629
21.4.14 RECALIBRATE DRIVE (11H) 629
21.4.15 CONTROLLER DIAGNOSTIC (14H) 629
21.4.16 READ DRIVE TYPE (15H) 630
21.4.17 DETECT MEDIA CHANGE (16H) 630
21.4.18 SET DISKETTE TYPE (17H) 631
21.4.19 SET MEDIA TYPE FOR FORMAT (18H) 631
21.5 INT 14H, SERIAL I/O SERVICES 632
21.5.1 INITIALIZE SERIAL PORT (00H) 632
21.5.2 SEND CHARACTER (01H) 633
21.5.3 RECEIVE CHARACTER (02H) 634
21.5.4 READ SERIAL PORT STATUS (03H) 634
21.5.5 EXTENDED INITIALIZE SERIAL PORT (04H) 635
21.6 INT 15H, GENERAL SERVICES 636
21.6.1 QUERY PORT 92H A20 GATE CAPABILITY (24H) 636
21.6.2 KEYBOARD INTERCEPT UP-CALL (4FH) 637
21.6.3 APM INSTALLATION CHECK (5300H) 637
21.6.4 APM INTERFACE CONNECT (5301H) 638
21.6.5 APM PROTECTED MODE 16-BIT INTERFACE CONNECT (5302H) 638
21.6.6 APM PROTECTED MODE 32-BIT INTERFACE CONNECT (5303H) 639
21.6.7 APM INTERFACE DISCONNECT (5304H) 640
21.6.8 APM CPU IDLE (5305H) 640
21.6.9 APM CPU BUSY (5306H) 641
21.6.10 APM SET POWER STATE (5307H) 641
21.6.11 APM ENABLE/DISABLE APM FUNCTIONALITY (5308H) 642
21.6.12 APM RESTORE APM POWER-ON DEFAULTS (5309H) 643
21.6.13 APM GET POWER STATUS (530AH) 643
21.6.14 APM GET APM EVENT (530BH) 644
21.6.15 SYSTEM REQUEST KEY (58H) 644
21.6.16 WAIT FUNCTION (86H) 645

21.6.17 MOVE EXTENDED MEMORY BLOCK (87H) 645
21.6.18 EXTENDED MEMORY SIZE (88H)....646
21.6.19 SWITCH TO PROTECTED MODE (89H) 646
21.6.20 DEVICE BUSY UP-CALL (90H) 647
21.6.21 DEVICE INTERRUPT UP-CALL (91H)....648
21.6.22 READ/WRITE CMOS RAM CELL (A0H)....648
21.6.23 SET CONSOLE I/O REDIRECTION (A1H) 649
21.6.24 GET EMBEDDED BIOS VERSION (A3H) 649
21.6.27 RETURN SYSTEM CONFIGURATION (C0H) 650
21.6.28 RETURN EXTENDED BIOS DATA AREA (C1H) 650
21.6.29 PS/2 MOUSE REQUEST (C2H) 650
21.6.30 WATCHDOG TIMER CONTROL (C3H) 651
21.6.31 DEBUGGER BREAKPOINT (D0H)....652
21.6.32 FLASH PROGRAMMING (E0H)....652
21.7 INT 16H, KEYBOARD SERVICES....653
21.7.1 READ KEYBOARD INPUT (00H) 653
21.7.2 RETURN KEYBOARD STATUS (01H) 654
21.7.3 RETURN SHIFT FLAG STATUS (02H)....654
21.7.4 SET TYPEMATIC RATE (03H)....654
21.7.5 PUSH DATA TO KEYBOARD (05H) 655
21.7.6 ENHANCED READ KEYBOARD (10H)....656
21.7.7 ENHANCED READ KEYBOARD STATUS (11H) 656
21.7.8 ENHANCED READ KEYBOARD FLAGS (12H) 656
21.7.9 SET CPU SPEED (F0H) 657
21.7.10 GET CPU SPEED (F1H)....657
21.7.11 READ CACHE STATUS (F400H) 658
21.7.12 ENABLE CACHE (F401H)....659
21.7.13 DISABLE CACHE (F402H)....659
21.8 INT 17H, PARALLEL I/O SERVICES 660
21.8.1 WRITE CHARACTER (00H)....660
21.8.2 INITIALIZE PRINTER (01H)....660
21.8.3 READ PRINTER STATUS (02H) 661
21.9 INT 1AH, TIME SERVICES....661
21.9.1 READ SYSTEM TIMER COUNT (00H) 661
21.9.2 WRITE SYSTEM TIMER COUNT (01H)....662
21.9.3 READ REAL TIME CLOCK TIME (02H)....662
21.9.4 WRITE REAL TIME CLOCK TIME (03H) 662
21.9.5 READ REAL TIME CLOCK DATE (04H) 663
21.9.6 WRITE REAL TIME CLOCK DATE (05H) 663
21.9.7 PCI SERVICES (B1H) 664

PART IV 665

TROUBLESHOOTING 667

22.1 COMPILING, ASSEMBLING, & LINKING 668
22.2 3RD-PARTY TECHNICAL SUPPORT....669

CALLING INTEL CORPORATION 670
CALLING MICROSOFT CORPORATION 670
CALLING PARADIGM SYSTEMS 670
CALLING PHARLAP SOFTWARE 671
22.3 TECHNICAL SUPPORT FROM GENERAL SOFTWARE 671
22.3.1 SUPPORT BY EMAIL 672
22.3.2 SUPPORT BY FAX 672
22.3.3 SUPPORT BY PHONE 672
22.3.4 REPRODUCING THE PROBLEM 672
22.3.5 USING TECH SUPPORT REQUESTS (TSRS) 673
22.4 ADVANCED TROUBLESHOOTING 673
22.4.1 DIAGNOSING POST 673
22.4.2 PCI ISSUES 675
22.4.3 BOOTING ISSUES 676
22.4.4 RS-232 COMMUNICATIONS ISSUES 676
22.4.5 CONSOLE I/O ISSUES 677
22.4.6 HARD DISK ISSUES 677
22.4.7 V20, V25, V30, AND 80186 ISSUES 678

PRODUCT CHANGE NOTES 679

EMBEDDED BIOS DOCUMENTATION 679
EMBEDDED BIOS SOFTWARE 680

Conventions Used in This Manual

Glossary of Terms

186-EC	A high integration CPU manufactured by Intel, consisting of a 186 core with proprietary (non PC-standard) UARTs, DMA controllers, interrupt controllers, and other peripherals in the same package.
386-EX	A high integration CPU manufactured by Intel, consisting of a 386SX core with additional UARTs, DMA controllers, interrupt controllers, and other peripherals in the same package. Normally used with the RadiSys R300 or R380 chipsets for PC/AT compatibility.
430HX	A Pentium chipset manufactured by Intel, with PCI support, for PC mainboards. Now bundled with Pentium Processors on a family of mezzanine boards, the EMBMOD133 and EMBMOD166.
430TX	A Pentium chipset manufactured by Intel, with PCI support, for laptop mainboards with low power requirements. Bundled with Pentium CPUs in families of mezzanine boards, the EMBMOD133/EMBMOD166, and Mobile Modules.
440BX	A Pentium II/Pentium III chipset manufactured by Intel, with PCI support, for high-performance systems. Bundled with Pentium II, Pentium III, and Celeron Processors in families of mezzanine boards.
810	A Pentium II/Pentium III chipset manufactured by Intel, with a new Hub Link architecture, for high-performance systems.
840	A Pentium Pentium III chipset manufactured by Intel, with a new Hub Link architecture, for high-performance multiprocessor systems.
Am186CC	A high integration, 40Mhz CPU manufactured by AMD, consisting of a 186 core with proprietary (non PC-standard) UARTs, DMA controllers, interrupt controllers, and other peripherals in the same package, especially designed for the communications market.
Am186EM	A high integration, 40Mhz CPU manufactured by AMD, consisting of a 186 core with proprietary (non PC-standard) UARTs, DMA controllers, interrupt controllers, and other peripherals in the same package.
Am186ES	A high integration, 40Mhz CPU manufactured by AMD, consisting of a 186 core with proprietary (non PC-standard) UARTs, DMA controllers, interrupt controllers, and other peripherals in the same package.
API	Application Programming Interface, a term for the architected system by which clients of a software system communicate their requests and receive information from a software component. There are many APIs provided by EMBEDDED BIOS, documented in Chapter 21, that provide access to BIOS-controlled devices and functions.
APM	Advanced Power Management, a system-wide architecture that includes the target hardware, system BIOS, operating system, and application. EMBEDDED BIOS provides APM services to operating systems and

	applications, and in the process of serving APM requests, manipulates the underlying hardware by calling functions in the BPM, CPM, and CSPM.
BIOStart™	The Windows-compatible rule-based expert system from General Software that is used to create new BPMs, CPMs, CSPMs, Project files, and build the BIOS. If the OEM does not have access to a Windows-based machine, then these files may be created and edited manually with a text editor, and the BIOS can be built using GSMAKE from the DOS prompt. BIOStart provides a point-and-click interface with rule checking to help guide the OEM through the BIOS configuration decisions.
BoardInit0	The BPM routine containing very early board-specific initialization. Usually, the OEM does not need to create this routine in new BPMs for custom targets.
BoardInit1	The BPM routine containing the bulk of board-specific initialization, usually involving the loading of chipset and Super I/O registers with default values, but may include identification of DRAM banks and their geometries. Commonly, the OEM must create this routine in new BPMs for custom targets.
BoardInit4	The BPM routine containing board-specific initialization that must occur with RAM enabled, but before the video and keyboard are initialized. Rarely used, but in some cases useful for downloading field-programmable software into the keyboard or video controllers.
BoardInit6	The BPM routine containing board-specific initialization that must occur with RAM enabled, and after the video and keyboard are initialized. This is the routine commonly used by OEMs requiring the console I/O to be redirected over a serial port based on the detection of a hardware shunt or attached RS-232 cable.
BPM	Board Personality Module, a set of source modules, including at minimum one .ASM file and one .INC file, that implements zero, one, or more of the architected functions documented in Chapter 20. The BPM contains board-related code, such as Super I/O programming and some chipset or high-integration CPU programming specific to a given board design, such as the Intel EXPLR1 or EXPLR2 reference designs, or the AMD SC300, SC310, SC400, or SC410 reference designs.
Build	The process by which the source modules and configuration files are used to form a binary file suitable for programming into a BIOS ROM. The Build process may be performed entirely within BIOStart, or from the DOS prompt with GSMAKE. The Build process requires invocation of the Microsoft or Borland assembler, linker, and various tools from General Software (see Chapter 4, Setting up Development Tools). There are two parts to the Build process: the 16-bit BIOS build that generates a 64KB-256KB .ABS file, and the 32-bit BIOS extensions build that creates 32-bit components like the 32-bit directory services and 32-bit PCI code. When the 32-bit build is performed, a utility called GSMERGE is used to merge the .ABS file from the 16-bit BIOS build, together with any other .ABS files such as VGA BIOS components, with the 32-bit BIOS components to produce a final output file that may be programmed into the BIOS ROM. Not all targets or projects may need to use the 32-bit build or GSMERGE process—for example if no 32-bit PCI services are required.
Chipset	One or more VLSI packages containing logic normally peripheral to a CPU, but necessary for the control of an external bus (such as ISA, EISA,

	or PCI), DRAM (including refresh and bank geometry), cache control, and other general hardware glue functions.
Console Redirection	A feature of EMBEDDED BIOS that allows debugger, SETUP, POST, and DOS keyboard and video I/O to be rerouted over an RS-232 serial port. EMBEDDED BIOS provides for different redirection assignments for the debugger, SETUP screen, and then all other console activities.
CE Ready™	A special feature of EMBEDDED BIOS that can boot Windows CE on a target directly from the pre-boot BIOS environment, without the need for special 3rd-party loaders or intermediary operating systems.
CONFIG.INC	An include file found in the INC directory that contains default symbol definitions for many of the BIOS configuration parameters. This file was modified directly by the OEM in prior versions of the BIOS, but is not modified in versions beyond 4.0. See Chapter 7 for details about the symbols found in this file.
CPM	CPU Personality Module, a set of source modules, including at minimum one .ASM file and one .INC file, that implements zero, one, or more of the architected functions documented in Chapter 18. The CPM contains CPU programming code specific to a given high-integration chipset, such as the Intel 186-EC or Intel 386-EX.
CSPM	Chipset Personality Module, a set of source modules, including at minimum one .ASM file and one .INC file, that implements zero, one, or more of the architected functions documented in Chapter 19. The CSPM contains chipset programming code specific to a given chipset, such as the AMD SC300, SC310, SC400, SC410, RadiSys R380, Ali M1487, or Acer M6117.
Embedded BIOS™	General Software's BIOS designed specifically for embedded systems.
Embedded DOS™	General Software's DOS architecture, of which two varieties are sold.
Embedded DOS™ 6-XL	General Software's real-time DOS for embedded systems, employing prioritized scheduling of lightweight threads with 32,767 priorities selectable on-the-fly, mutual exclusion semaphores, signaling semaphores, message ports and queues, and a fully reentrant INT 21h API. With a full set of utilities, device drivers, and two versions of COMMAND.COM (one with a small footprint and a full capability version).
Embedded DOS™-ROM	General Software's ROMmable DOS for consumer electronics, with a configurable footprint as small as 32KB, integrated autoscanning ROM disk, VG230 support, full set of utilities and device drivers, a built-in resident COMMAND.COM, and an external COMMAND.COM.
File System	A mass storage system, responding to INT 13h disk I/O BIOS requests, either consisting of a real diskette drive/IDE drive, or an emulated drive. File Systems are defined with the FILE_SYSTEM macro in the project file. File Systems are managed by File System Drivers, or FSDs. The File System Control Layer (FSCL) manages File System Drivers.
File System Driver	A body of code that manages file systems of a given class. Predefined file systems include Floppy, Ide, Rom, Ram, and Flash.
FSCL	File System Control Layer, the subsystem within EMBEDDED BIOS that manages File System Drivers (FSDs) and processes INT 13h requests.

FSD — See File System Driver.

Flash — A special class of nonvolatile memory, capable of being read, written, and erased. Flash may be based on NOR or NAND technologies. NOR Flash is usually packaged in Bulk Erase, Boot Block, or Sectored components, and may be directly mapped into the memory address space of the CPU. NAND Flash has smaller block sizes and is typically I/O mapped. EMBEDDED BIOS uses MTDs to access Flash components. Flash parts may be interleaved to widen the data path; when this technique is employed, MTDs must be made aware of this design, as it doubles the logical block size of the Flash array.

INT 10h — The BIOS software API that provides access to the video display. The INT 10h API is documented in Chapter 21.

INT 13h — The BIOS software API that provides access to IDE disks, floppy disks, and their emulators. The INT 13h API is documented in Chapter 21.

INT 16h — The BIOS software API that provides access to the keyboard. The INT 16h API is documented in Chapter 21.

INT 1ah — The BIOS software API that provides access to the real time clock, and also to PCI services. The INT 1ah API is documented in Chapter 21.

M1487 — A chipset manufactured by Acer Labs, Inc. (ALI), commonly referred to as “ALI Finali”, that provides PCI bus management, and works with 386 and 486 CPUs manufactured by Intel, AMD, Cyrix, and SGS Thomson.

M1541 — A chipset manufactured by Acer Labs, Inc. (ALI) that provides PCI bus management, and works with AMD K6 CPUs manufactured by AMD, and Pentium II Processors manufactured by Intel.

M6117 — An embedded CPU manufactured by Acer that behaves like a 386 CPU core with a chipset in a single package. As of this writing, the latest stepping is M6117C, which requires a different chipset module than the M6117 chipset does.

Manufacturing Mode — A feature of EMBEDDED BIOS that provides host PC control over an embedded target running EMBEDDED BIOS through an RS-232 asynchronous serial communications protocol. See Chapter 14 for details about Manufacturing Mode.

MBR — Master Boot Record, the first 512-byte sector on a hard drive or its emulator. The MBR contains the primary bootstrap code necessary to load the PBR of the active partition and transfer control to it. The MBR, like the PBR, contains a two-byte 55h/aah signature in the last two bytes, and contains a Partition Table with four entries, each of which specifies a section of a hard drive to be treated as a separate logical drive or storage medium.

MCL — Media Control Layer, the EMBEDDED BIOS subsystem that manages requests to read, write, erase, and lock storage in Flash, ROM, and RAM media. The MCL is generalized to permit I/O to other types of media (i.e., remote storage over Ethernet) should this be desired by the OEM.

MMU — Memory Management Unit, a hardware component usually found in a chipset or high-integration CPU, providing hardware mapping of storage

	into the physical address space available to the CPU. Commonly, such hardware maps (under programmed control) portions of ROM or Flash devices wired to chip select lines on a high-integration CPU into a segment of memory below the 1MB address marker for accessing by the EMBEDDED BIOS MCL.
MTD	Media Technology Driver, a small software driver providing MCL with functionality that drives a particular type of storage in a specific interleave factor. For example, Bulk Erase Flash programming is handled with one MTD, ROM with another, RAM with yet another MTD, and sectored Intel Flash managed with a different MTD.
OPTIONS.INC	An include file found in the INC directory that contains default symbol definitions for many of the BIOS configuration parameters. This file was modified directly by the OEM in prior versions of the BIOS, but is not modified in versions beyond 4.0. See Chapter 7 for details about the symbols found in this file.
PBR	Partition Boot Record, the first 512-byte sector on a Floppy or its emulator. The PBR contains the operating system bootstrap code, has a two-byte 55h/aah signature in the last two bytes, and contains a BPB at offset 0bh. A PBR is also present as the first sector within a hard disk partition, but is not the same as an MBR, which is the first sector on a hard disk.
POST	Power-On Self-Test, the process by which EMBEDDED BIOS performs its initialization sequence to ready the target hardware and the BIOS software for support of the operating system or Manufacturing Mode.
Project File	A configuration file for the BIOS build, produced with the BIOStart utility or with any text editor, that contains overrides (symbol redefinitions) for the default configuration parameters found in INC\OPTIONS.INC or INC\CONFIG.INC.
RAM Disk	A feature of EMBEDDED BIOS that emulates a floppy disk by using random access memory to maintain an image of a floppy disk. When the RAM disk is read, sectors within the floppy disk image in RAM are copied by the RAM disk software into the user buffer with CPU move instructions. When the RAM disk is written, sectors from the user buffer are copied into the floppy disk image in RAM to replace the previous data.
RFD	A feature of EMBEDDED BIOS that emulates a floppy disk by using NOR-technology Flash memory to maintain a simulated image of a floppy disk. Unlike the ROM or RAM disks, the Flash disk's storage does not look byte-for-byte like the data on a floppy disk. Instead, the RFD moves data from block to block, to even the wear on Flash media, and to accommodate for the basic write/erase principles of NOR Flash memory technology.
ROM Disk	A feature of EMBEDDED BIOS that emulates a floppy disk by using read-only memory to maintain an image of a floppy disk. When the ROM disk is read, sectors within the floppy disk image in ROM are copied by the ROM disk software into the user buffer with CPU move instructions.
SC300	An embedded CPU manufactured by AMD that behaves like a 386 CPU core with a chipset in a single package. The SC300 contains, among other components, an integrated LCD controller, memory management unit (MMU) and a proprietary PCMCIA controller.

SC310	An embedded CPU manufactured by AMD that has all the functionality of the SC300 except LCD controller and PCMCIA controller.
SC400	An embedded CPU manufactued by AMD that behaves like a 486 CPU core with a chipset in a single package. The SC400 contains, among other components, an integrated LCD controller, memory management unit (MMU) and a 386-compatible PCMCIA controller.
SC410	An embedded CPU manufactured by AMD that has all the functionality of the SC400 except LCD controller and PCMCIA controller.
SC520	An embedded CPU manufactured by AMD that provides a 5x86 CPU core with 16KB on-chip cache, PCI, SDRAM memory controller, and PC/AT glue logic and peripherals.
SETUP	A feature of EMBEDDED BIOS that provides a full-screen interface used by the end-user to manipulate CMOS settings and configure the BIOS's operation on the target, as well as enter additional BIOS modes, such as the built-in debugger and Manufacturing Mode.
Super I/O	A VLSI package containing one or more functions traditionally performed by discrete controllers. Typically, Super I/O controllers from National or other manufacturers contain UARTs, parallel ports with ECP and EPP support, a floppy disk controller, IDE controller interface logic, and an IRDA interface. Super I/O controllers are usually software configurable by the BIOS through the setting of configuration registers on the package. The configuration registers are commonly accessed by the CPU by writing a configuration register index value into an I/O port such as 22h, then reading or writing the data for the configuration register through another I/O port, such as 23h.
Support Module	A package of one or more files comprising a BPM, CPM, or CSPM that can enhance the EMBEDDED BIOS core software to provide support for an evaluation board, a high-integration CPU, or a chipset. Support Modules are available from General Software and from its Technology Centers.
Technology Center	An authorized software or hardware development center that provides development services for General Software and/or its OEM customers.
Distribution Center	An authorized sales representative that provides sales and first-level technical support and licensing in the native language and customs for General Software products.
Support Center	An authorized full service, highly-technical, support center that provides technical support in the native language and customs for General Software products. Contact the General Software web site for a Support Center in your area.

Chapter 1

INTRODUCTION

Introducing EMBEDDED BIOS

Thank you for choosing General Software's EMBEDDED BIOS brand BIOS (Basic Input/Output System) firmware for use in your embedded system. EMBEDDED BIOS offers a superior combination of configurability, performance, and functionality that enables it to satisfy the most demanding ROM BIOS needs for your embedded system. Its modular architecture and high degree of configurability make it the most flexible BIOS in the world.

Configurability

The configurability of EMBEDDED BIOS is unsurpassed by any other BIOS product in the industry. At the lowest level, EMBEDDED BIOS comes with full source code, enabling the BIOS adaptation engineer to make custom modifications that would otherwise not be possible with a "binary-only" adaptation kit from a desktop BIOS manufacturer. Source code offers greater security and the ultimate low-level control for embedded designs.

All source-level configurable options are defined as symbolic equates in two source code files: INC\OPTIONS.INC and INC\CONFIG.INC. The BIOS build architecture provides for a project file named PROJECTS*projectname**projectname*.INC, that contains OEM-specified overrides for any or all of the over 400 standard source-level configuration options. The separation of the released BIOS sources and the OEM's needed changes allows for a high degree of maintainability, and at the same time permits many simultaneous projects to use the same source tree.

Although project files can be created and edited with any simple text editor, EMBEDDED BIOS provides BIOStart, a Windows-hosted expert system that the OEM can use to create and edit project files with the guidance of expert BIOS building knowledge at General Software. BIOStart's knowledge base, derived from core BIOS developers and customer support engineers, provides multiple views of the BIOS options, and cross-references options and tuning parameters so that parameter dependencies are handled properly.

BIOStart can also patch binary copies of EMBEDDED BIOS, allowing the OEM's customers to configure certain aspects of prebuilt BIOSes without the need for rebuilding from the source code. This provides substantially the same functionality as other "Binary Configuration" programs on the market.

At run-time, EMBEDDED BIOS can be configured with a comprehensive SETUP screen system (itself a configurable subsystem at the source code level). SETUP options include standard CMOS configuration, custom programming, built-in diagnostics, access to Manufacturing Mode, Debugger access, and support for initialization of RAM and Flash disks. The SETUP screen can even be configured to run over an RS-232 serial line, if desired.

The EMBEDDED BIOS architecture provides for operation with different high-integration chipsets and classes of CPU. Chipsets are supported by EMBEDDED BIOS through an architected interface that the core BIOS makes calls to (the Chipset Personality Module, or CSPM). The adaptation engineer adds customized chipset-programming code to the standard chipset programming template, so that it is localized to a few routines inside one custom module that other BIOS components call. CPU support is handled in a similar fashion, with CPU Personality Modules (CSPMs) that support different classes of CPUs. Standard support for 8086, 80286, 80386, 80486, Pentium, Pentium II, Pentium III, Celeron, and Pentium Pro is provided as a CPU class, and other CPU class modules are available from General Software. Finally, a third type of module called the Board Personality Module (BPM) provides a place for board-specific modifications to the BIOS to be placed by the OEM.

Embedded Features

Serving the entire embedded 80x86-based market, EMBEDDED BIOS offers special-purpose features not provided by typical desktop BIOS implementations.

With embedded CPU support, virtually any type of CPU can be supported, provided it is reasonably compatible with the Intel 8086 instruction set. For example, Intel's 186EA, 186EB, 186EC, 386EX, and other CPUs are supported by EMBEDDED BIOS when the associated CPU-specific personality module is selected for the project. This allows support of high-integration processors that have on-board timers, DMA controllers, serial ports, watchdog timers, power management, and other features.

With Chipset support, virtually any add-on chipset, or CPU with on-board chipset (such as the AMD SC300, SC310, SC400, and SC410 processors) can be supported by EMBEDDED BIOS. Traditionally, chipsets provide DRAM memory management, bus control, and address space management. The EMBEDDED BIOS architecture provides for Chipset Personality Modules that can be selected for a project.

EMBEDDED BIOS's board-level support provides for the OEM to control the BIOS's access to Chipset and CPU modules in major or subtle ways. Essentially a routing module, the board module contains routines which call associated routines in the Chipset and CPU Personality Modules. The board module routines can be modified as needed to replace the calls to the underlying CPU and chipset modules with custom code, as needed for hardware designs that work differently than standard reference designs supported by General Software.

EMBEDDED BIOS is implemented in hand-optimized 8086 assembly language, with special code paths for 80186, 80286, 80386, i486, and Pentium processors. The code paths have been hand-tuned to minimize the interrupt latency commonly found in desktop BIOS

implementations, and many of the "hot paths" of the BIOS have been straight-line optimized for the common case.

ROM disk software is integrated directly with the system BIOS itself, eliminating the need to populate the ROM scan area with ROM BIOS extensions to simulate one or more floppy or hard disks in ROM. Instead, with the ROM disk configuration feature enabled, an image of a floppy or hard disk can be stored in ROM anywhere in the address space of the target and treated as a solid-state drive. If the ROM disk feature is enabled, the ROM disk can be selectively turned on or off in the Setup screen.

RAM disk software is also integrated directly into the system BIOS to support PCMCIA SRAM cards and other RAM areas as floppy or hard disk emulators. SETUP even has a formatting screen for the RAM disk.

The system BIOS supports a Resident Flash Disk (RFD) that provides read/write access to sectored Flash devices as though they were a floppy or hard disk of up to 32MB in size. The inclusion of this software makes it simple and easy to support Flash in embedded and hand-held consumer electronics. Multiple RFDs can be supported in the same target.

The integrated BIOS debugger gives the adaptation engineer the capability of debugging the hardware and bringing-up the system with powerful tools like a disassembler, breakpoints, CMOS editing, A20-line gating commands, cache control commands, PCI bus management commands, and Super I/O controls. The debugger is very useful for debugging chipset modules, CPU class modules, and initialization of user ROM extensions and hardware. Like the Setup screen, the integrated BIOS debugger can run directly on a PC keyboard and video screen, or it can be redirected over an RS-232 serial link.

Embedded systems deployed into more inaccessible areas need watchdog timer support, so that they can automatically restart in the event that application or system software fails. EMBEDDED BIOS provides watchdog timer control functions to allow operating systems and application programs to use watchdog timer hardware found in chipsets and certain CPU classes.

Keyboard and video output may be selectively redirected over RS-232 serial links for different system components. For example, standard console I/O, such as that used by DOS and DOS applications, can be redirected over any COM port, including those built-into high-integration CPUs. Debugger I/O and Setup screen I/O can also be redirected over the same or different RS-232 serial links.

A special Manufacturing Mode feature provides the necessary provisions for programming electronics products through a high-speed serial link, and then testing and repairing the same items in the field at service centers. The OEM can write custom software that uses EMBEDDED BIOS Manufacturing Mode functions to perform virtually any maintenance or programming task on the target under host control.

Desktop PC Features

EMBEDDED BIOS provides a comprehensive Power-On Self-Test (POST) algorithm that is automatically configured for the peripherals and capabilities selected by the adaptation engineer. During POST, hardware is initialized and tested, including the CPU, RAM, and peripherals. POST provides "beep code" diagnostics for errors when a display is not available, as well as error message diagnostics on the display when available. POST can also be configured to output status report codes to a manufacturing port (typically, port 80h) so that automated Q/A equipment can determine the status of a system during POST. A special set of ASCII POST

status codes are also available through a serial port, for flexibility in the debugging process when new hardware is being brought up. Either POST code system, or both, can be used during debugging.

The EMBEDDED BIOS SETUP screen system is configurable at the source level by the adaptation engineer to contain any combination of subscreens, including Basic CMOS Configuration, Custom Configuration, Shadow Configuration, Diagnostics screens, Manufacturing Mode, Debugger access, and formatting of drive emulators such as RAM and RFD drives. SETUP screens can also be customized at the source level (in the Board Personality Module) to contain custom fields as required by the application.

Also available is a password protection system, so that a password must be provided by the end-user before POST allows booting of an operating system. The password is stored in CMOS, is one-way encrypted, and can be modified in a Setup screen.

The ability to shadow slower ROM devices with DRAM or SRAM is selectable in the Shadow Setup screen and calls chipset-specific code to enable shadowing for the BIOS ROM itself or for feature ROMs on a 16KB region basis. DRAM may take the form of FP, EDO, SDRAM, RDRAM, or other technologies.

EMBEDDED BIOS provides extensive support for both internal CPU cache control (i486 and above) and external cache control (typically chipset-controlled). Internal cache is managed by the CPU class personality modules, whereas external cache is managed by the cache manager, which directs peripherals (chipset, 8042, custom I/O ports, or CPU integrated peripherals) to manage the cache. Keyboard controls on the PC/AT keyboard are implemented for enabling and disabling the cache on-the-fly (while the system is running). The BIOS provides cache control services to applications that allow operating systems and user code to control and inspect the status of the cache.

CPU speed controls are handled by the system BIOS by routing control through the appropriate logic (chipset, 8042, custom I/O ports, or CPU integrated peripherals). As with cache control, CPU speed is controllable on-the-fly at the keyboard or via programming interfaces.

Software Compatibility

EMBEDDED BIOS offers a high degree of compatibility with past and current BIOS standards, allowing it to run off-the-shelf operating system software and application software.

EMBEDDED BIOS has been tested with all major versions of DOS, including MS-DOS, DR-DOS, Embedded DOS-ROM, and Embedded DOS 6-XL; OS/2; Windows 3.1, Windows 95, Windows 98, Windows NT, and real time operating systems such as QNX.

EMBEDDED BIOS is rigorously tested with programs such as AMI Diag, MSD, Check-It, Manifest, Q/A Plus, and so on, ensuring its compatibility with established desktop application standards.

In addition to its standard data structures and programming interfaces, EMBEDDED BIOS provides additional industry-standard interfaces, such as APM and PCI.

Applications for EMBEDDED BIOS

EMBEDDED BIOS addresses the architectural needs of several different classes of applications; namely, Hand-Held Consumer Electronics, Consumer Appliances, Single Board Computers, and Special Purpose Devices.

Hand-held consumer electronics work better and cost less with EMBEDDED BIOS. This BIOS has a small footprint, high configurability, Flash file system, DOS operating system, Manufacturing Mode, Advanced Power Management, and low cost.

Consumer Appliances, such as televisions, set-top boxes, internet terminal devices, microwaves, and telephones, can all be designed and developed quicker with EMBEDDED BIOS. The BIOS provides access to standard peripherals, which can then be replaced by solid-state disk emulators in the final production BIOS. This means consumer appliances can be developed based on the model that they are a PC-compatible that runs DOS programs in response to user commands. The tried-and-proven techniques of DOS application programming are applied to the design, development, and testing of consumer appliances to yield the quickest time to market and lowest risk.

Single Board Computers designed with EMBEDDED BIOS in mind can provide users with the extra features they need to be competitive in their field. Consider the flexibility that ROM, RAM, and Flash disks offer the licensee, in addition to console redirection. Industry-standard implementation of data structures and software interrupts make EMBEDDED BIOS a solid choice for the SBC vendor and for the vendor’s customers.

Highly-Specialized Devices built around EMBEDDED BIOS can be debugged early with the integrated BIOS debugger, loaded with software using Manufacturing Mode, and verified in Q/A with the burn-in Diagnostics SETUP screens. When the hardware is highly custom, EMBEDDED BIOS provides the richest tool environment, and the most configurable options, making it the safest route to bringing up new designs quickly.

Lowered System Cost With Embedded DOS-ROM

Cost-sensitive applications are high-volume, low-priced commodity electronics products that require a very low per-copy royalty scheme for licensing system software such as BIOS and DOS. EMBEDDED BIOS's pricing is highly competitive, and at the same time provides Embedded DOS-ROM, so that there is no need to license a separate DOS product from another vendor.

Choosing Embedded DOS-ROM or Embedded DOS 6-XL

General Software's Embedded DOS 6-XL is a DOS Adaptation Kit that, like the EMBEDDED BIOS Adaptation Kit, enables the embedded system developer to produce a custom DOS environment for an embedded target.

Embedded DOS 6-XL is a real-time, reentrant, multitasking operating system that runs embedded application software built with DOS tools, such as Microsoft MSVC++ and Borland C++. It is configured with over 70 configuration options, supporting a wide variety of real-time DOS-based applications.

If you have a need for a DOS that is also a real-time kernel, we suggest using Embedded DOS 6-XL as your operating system. If you do not have multitasking or real-time requirements, then Embedded DOS-ROM will be an excellent choice.

Related Reading

For background information, or reference use, we suggest that you read the following related publications. We use these resources for developing applications for our customers.

American Megatrends, WINBIOS and AMIBIOS Technical Reference, AMI, Norcross, Georgia.

Brown, Ralf, PC Interrupts, Addison/Wesley, Reading, Mass.

Dipert, Brian, & Levy, Markus, Designing with Flash Memory, Annabooks, 1993.

Duncan, Ray, MS-DOS Functions, Microsoft Press.

Gilluwe, Frank Van, The Undocumented PC, Addison/Wesley, Reading, Mass., 1994.

IBM Corporation, Technical Reference Personal Computer XT Model 286, Order No. 68X2210, New York, NY, 1986.

Intel, 386 DX Microprocessor Programmer's Reference Manual #230985-003, Intel Corporation.

Microsoft Press, MS-DOS 6.22 Programmer's Reference, Microsoft Corporation.

Phoenix, System BIOS for IBM PC/XT/AT Computers, Addison/Wesley, Reading, Mass., 1991.

Schulman, Andrew, Undocumented DOS, Addison/Wesley.

About the EMBEDDED BIOS Adaptation Kit

Your new EMBEDDED BIOS 4.3 Adaptation Kit includes the following parts:

- EMBEDDED BIOS Adaptation Guide (this manual)
- Core BIOS Software on Diskettes or CD-ROM (1st diskette contains installer)

- Optional Support Module Disks (each contains INSTALL.BAT)
- Optional CPU Personality Module Disk (each contains INSTALL.BAT)

- EMBEDDED BIOS OEM License Agreement
- Technical Support Request (TSR) form
- Product Registration form

The rest of this part of the manual describes the steps needed to produce a BIOS with this adaptation kit. Chapter 2 will explain how to install the EMBEDDED BIOS Adaptation Kit software on your development system. Chapter 3 provides a good background for newcomers to the BIOS world, and an interesting architectural refresher tour for x86 PC veterans. Chapter 4 provides the information you need to set-up your development tools to work with EMBEDDED

BIOS. Finally, in Chapters 5, 6, and 7, you'll learn how to customize and build a BIOS to your specifications.

Chapter 22 provides helpful troubleshooting techniques and procedures that can save time and resolve technical problems quickly without guesswork.

The *Release Disks* contain the core BIOS software, utilities, BIOStart, and installer. DOS users can use the INSTALL.BAT on disk #1 to install the system. Windows users should use the SETUP.EXE program disk #1 to install the system. If a README.TXT file is present, the up-to-the-minute instructions in that file should be used instead of the instructions in this Chapter.

The *Support Module Disks* each contain the INSTALL.BAT file used to install a personality module for any plug-in support for chipsets, CPUs, or reference design boards. These personality modules are sold separately and augment the core BIOS kit.

The enclosed *OEM License Agreement* enables you to license binary adaptations of the EMBEDED BIOS software. Contact General Software for help with this form and for current pricing for the volume you are interested in. We suggest that you begin the licensing process early so that you can take advantage of current rates (they are subject to change and have never gone down before).

The enclosed *Technical Support Request (TSR)* form should be used to submit technical support requests to General Software by FAX. You may duplicate this form as needed to make multiple requests.

General has implemented a proprietary and sophisticated *Product Support Database* that allows tracking the nature, content, progress, and history of each call made to or from our Product Support Group. This allows any Support Technician to access the needed information should your situation require more than one call and the original Support Technician is not currently available to take your call or respond to your fax.

Also enclosed is a *Product Registration* form that should be completed and mailed immediately. This information is needed for technical support and also makes you eligible to receive upgrades and access General Software's on-line services.

For Customers with Version 4.0 of Embedded BIOS

If you are already using EMBEDDED BIOS 4.0, then you already know about project files, board modules, chipset modules, and CPU modules. The new additions then, are relatively straightforward:

1. **Do not edit INC/OPTIONS.INC and INC/CONFIG.INC**. Just in case you have a beta version of 4.0 that did not use project files, this is an essential concept. If this is new to you, please read this entire section, plus the section that follows it, to make sure you're aware of the project architecture.

2. **Disk Support is Reorganized-- the FILE_SYSTEM table defines them**. Whereas past versions of EMBEDDED BIOS used symbols like **OPTION_SUPPORT_IDE**, **OPTION_SUPPORT_FLOPPY**, **OPTION_SUPPORT_RFD_DISK**, **OPTION_SUPPORT_ROM_DISK**, and **OPTION_SUPPORT_RAM_DISK** to enable device drivers, these are now thought of as "file systems" and are enabled with the **FILE_SYSTEM** macro in the project file.

3. Multiple Disk Emulators are Supported. Previously the ROM, RAM, and RFD disk emulators could only support one image each. Now they are reentrant and each can support up to 64 drives. This makes it possible for a system to have many ROM, RAM, and RFD disks in a system.

4. OEM-Written File Systems Supported. Previously only ROM, RAM, and RFD disks were supported. Now OEMs can add their own file systems, perhaps even clones of these basic systems with special features, without modifying the core BIOS. This is accomplished by giving the new OEM-written file system a name other than ROM, RAM, or RFD, inserting this code into the board module, and then declaring the new file system using the **FILE_SYSTEM** macro in the project file.

5. Drive Assignments are more Dynamic. Previously the Setup screen assignments were simple-- floppy types were extended to support ROM, RAM, and RFD disks. Now file systems are mapped to drive letters in Setup's BASIC screen. This allows file systems to be mapped to any drive letter, soft or hard.

For Customers with Versions of Embedded BIOS Earlier than 4.0

If you are already using an earlier version (3.x or less) of EMBEDDED BIOS, then you'll want to know how this version of the software differs from yours so that you can use it properly. In order to accommodate the needs of more customers with diverse needs, we have made changes to the build architecture of EMBEDDED BIOS with which you'll need to be familiar.

1. **Do not edit INC/OPTIONS.INC and INC/CONFIG.INC**. Whereas past versions of EMBEDDED BIOS were configured with these files, version 4.3 uses an OEM-edited *project file* that contains only overriding definitions to these two standard files. Project files reside in subdirectories underneath the new PROJECTS directory.

2. Use the new version of GSMAKE supplied with this adaptation kit. The new version includes support for features required by BIOStart and additions to the MAKEFILEs. You must make certain that any copies of the old GSMAKE in your path do not come before the new version. The old version of GSMAKE does not work with the new MAKEFILEs in this release.

3. There are now three personality module types: *CPU*, *Chipset*, and *Board* types. CPU Personality Modules each reside in their own subdirectory underneath the `CPUS` subdirectory. Chipset Personality Modules each reside in their own subdirectory underneath the `CHIPSETS` subdirectory. Finally, the new Board Personality Modules each reside in their own subdirectory underneath the `BOARDS` subdirectory. The new Board Personality Module is used to contain the board-specific code that used to reside in the Chipset Personality Module.

4. A project file should be created for each new project. Project files not only include the overrides for the options and parameters in the `OPTIONS.INC` and `CONFIG.INC` files, but they also specify which board, chipset, and CPU modules are to be used in the project. Here is how we suggest using this new flexibility:

   4a) Create a board module for each new board you design with. Don't create a board module just because you're developing the next revision of a BIOS for a product. Instead, see if you can create a new project file for it and reconfigure it with just BIOS options. This will reduce your maintenance overhead in the long run.

4b) Create new project files that refer to the same CPU, chipset, and board modules when you want to vary the feature support of the core BIOS. This makes total reuse of the board module's functionality, and simplifies maintenance when the board module is adjusted.

4c) Don't modify chipset or CPU support modules from General Software just to change initialization for the CPU or chipset. Instead, comment out the "Rcall" to the chipset or CPU routine in the appropriate board module routine, and insert the code that needs to be used for initialization in place of it. This will keep your change local, and should be compatible with future versions of the BIOS.

4d) When creating new chipset, CPU, or board modules, start with an empty module file that is essentially a clone of the NOCPU.ASM, NOBOARD.ASM, or NOCHPSET.ASM files. Add only those routines to the module that are different from the established standard, found in SYSTEM/CPU.ASM, SYSTEM/BOARD.ASM, and SYSTEM/CHIPSET.ASM, respectively. Note that there is a slight routine definition change from the routine in the SYSTEM directory to the routine in the personality module: (i) remove the IFNDEF/ENDIF bracket around the routine, and (ii) add ", OVERRIDE" to the parameters on the **DefProc** or **DefRtn** statement.

5. CMOS cell assignments have changed. A signature has also been installed in the CMOS array to ensure that CMOS checksums that are produced by other BIOSes do not cause the chipset-programming portions to be blindly loaded and used as running values.

6. Learn BIOStart. It is the new way projects are managed, and its knowledge base is continually enhanced to provide more automation for BIOS customization.

7. Some options are deleted, some new ones added, and some renamed. For example, all of the old-style Flash part options are replaced with a table created with the **MEDIA_REGION** macro. We added lots of new control over POST and error handling, and renamed a few features like **OPTION_FORCE_9600_BAUD** to **OPTION_SERIAL_9600_BAUD** for consistency.

8. The Flash support is completely reorganized. **OPTION_FLASH_xxx** is no longer used to specify what Flash parts will be supported in a configuration. Instead, use the **MEDIA_REGION** macro to build a table in your project file. This allows support of multiple Flash and other media types, and provides a great framework for new media support through the new Media Technology Driver architecture.

9. Embedded DOS-ROM can execute directly out of ROM. This means that it can be combined with the BIOS image and need not consume ROM or Flash disk space. Further, the optional mini-COMMAND feature supported by Embedded DOS-ROM that causes the commands to be parsed inside the DOS kernel itself, so that an external loadable COMMAND.COM file is unnecessary.

PART I

BASIC STEPS FOR BIOS BUILDING

This part of the EMBEDDED BIOS reference documentation discusses the basic information needed by the BIOS adaptation engineer to use this adaptation kit, including installation of the software, build procedures, and recommended adaptation methodology. Here is your roadmap:

1. Install the Core BIOS Software (from diskettes or CD-ROM) (see Chapter 2 for Details).
   - 1.1 Windows users should use the Windows-based `SETUP.EXE`.
   - 1.2 DOS users should use `INSTALL.BAT`.

2. Install any Optional Support Modules (from diskettes) (see Chapter 2 for Details).
   - 2.1 Both Windows and DOS users use `INSTALL.BAT` on each disk.

3. Configure your development environment (see Chapter 3 for Details).
   - 3.1 Set the `PATH` to include the `TOOLS` subdirectory.
   - 3.2 Set the `BORLAND` environment variable if you use Borland tools.
   - 3.3 Set the `MASM61` environment variable if you use MASM 6.1.

4. Build the "SAMPLE" BIOS project to make sure the software is installed correctly.
   - 4.1 DOS users:
     - 4.1.1 `SET GSPROJ=SAMPLE`
     - 4.1.2 `CD PROJECTS`
     - 4.1.3 `GSMAKE`
   - 4.2 Windows users: Run BIOStart, select Build, then `SAMPLE` project.
   - 4.3 If the build completed successfully, you're ready to take the next step.

5. If you have a reference design board, build a BIOS and install it in ROM on the board.
   - 5.1 Determine the project name (shown here as `projname`) for the reference design.
   - 5.2 DOS users:
     - 5.2.1 `SET GSPROJ=projname`
     - 5.2.2 `CD PROJECTS`
     - 5.2.3 `GSMAKE`
   - 5.3 Windows users: Run BIOStart, select Build, then `projname` project.
   - 5.4 The BIOS image is `PROJECTS\projname\projname.ABS`, (or `.BIN` if 16/32 build)

Chapter 2

INSTALLATION

Backing-up Your Release Disks

If you received your copy of the EMBEDDED BIOS software on diskettes, you should not modify any of the files on the release disk(s). You should immediately make a backup of your disks with DISKCOPY and store the originals in a safe place. If you did not receive release disks, or if you received disks that are unreadable (sector not found, data error, etc.), please contact General Software for replacements.

Installing the Core EMBEDDED BIOS Software

The core EMBEDDED BIOS software, including build tools and source code, is installed separately from any additional support modules. The core BIOS software comes on CD-ROM or diskettes, and support modules come on diskettes only.

It is recommended that Windows NT 4.0 or greater be used as the operating system environment for BIOStart or the DOS box environment.

If you have Windows 95, Windows 98, or Windows NT on your development machine, you should use Windows to install the core BIOS by selecting `Start|Run|A:SETUP`. This will install the BIOStart software and will run the installer inside BIOStart. This process is quick and seamless. A new icon will appear on your desktop, called BIOStart. This is the program that is used to perform option-level customization of the BIOS.

If you do not have Windows on your development machine, you need to use the DOS-based batch file to install the core BIOS software. Before running the `INSTALL.BAT` batch file, you need to create a subdirectory on your hard disk where you wish to install the system; we recommend `C:\EBIOS43`. After creating the subdirectory, use the CD command to make that directory the current one. Then, after changing into the new directory, run the installation batch file from drive A: or B: as follows:

```
C:\> MD EBIOS43
C:\> CD EBIOS43
C:\EBIOS43> A:INSTALL
```

Installing Additional Support Modules

If you have purchased additional support modules to go with the core BIOS, then they must be installed after the core BIOS, using the `INSTALL.BAT` file that comes on each diskette. For each disk, place it in drive A:, and type:

```
C:\EBIOS43> A:INSTALL
```

After the installation process is completed, a tree of subdirectories will be created underneath the `EBIOS43` directory.

Organization of the Software

After the installation process is completed, a tree of subdirectories will be created underneath the `EBIOS43` directory. The `EBIOS43` directory is used to anchor all of the directories in the EMBEDDED BIOS development environment. The directory structure is outlined in Figure 2.1 (this figure may be incomplete if new components have been added to the release since this printing of the manual).

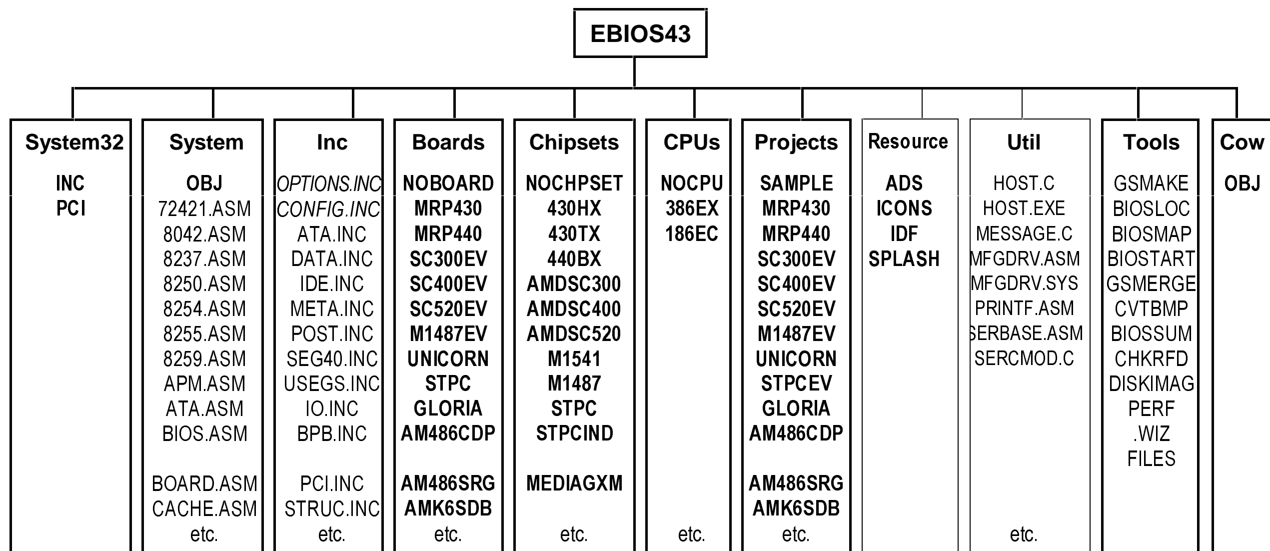


Figure 2.1. EMBEDDED BIOS Source Code Adaptation Kit Software Organization.

PROJECTS Subdirectory

The `PROJECTS` subdirectory contains one or more subdirectories, each associated with a BIOS adaptation project. Each project has a name, and its associated subdirectory must be given the same name.

Inside a project's subdirectory are one or more files that define the BIOS build for the project. The file with the .INC extension (called the "project file") specifies the parameters for the 16-bit BIOS build—this file must always be present.

A second file in this subdirectory with an .IDF extension (called the "IDF file") specifies how the 16-bit output file and other components, such as VGA BIOS extensions and output files from the

32-bit BIOS build, are to be combined into a single output file, usually with a .BIN extension. Many BIOS projects do not need an IDF file because they do not use 32-bit BIOS components (such as 32-bit PCI).

> *Note: If an IDF file is present, then the 32-bit build will be invoked, unless the special environment variable NOGSMERGE is defined. Use "SET NOGSMERGE=Y" on the command line to disable the 32-bit BIOS build and merge step. Use "SET NOGSMERGE=" on the command line to remove the environment variable, allowing the merge to take place if an IDF file exists.*

Both the project file and the IDF file must be named the same as the subdirectory name, with the proper extensions.

The core BIOS comes with a SAMPLE project, defined with its own `PROJECTS\SAMPLE` subdirectory that contains a file called `SAMPLE.INC`. This file is editable with any text editor, and defines which board, chipset, and CPU personality modules will be used, and any build option overrides associated with the BIOS build.

When preparing to build a BIOS under DOS, the `PROJECTS` subdirectory must be made the current working directory of the current drive. This subdirectory contains the master `MAKEFILE`, used as instructions to the GSMAKE process.

It will be noted later, but is a good idea to reiterate here, that the GSPROJ environment variable is used by this `MAKEFILE` to determine which project to build in the DOS environment. Thus, if a dozen or so projects are defined, and the SAMPLE project is to be built, then the OEM must set GSPROJ=SAMPLE before running GSMAKE. BIOStart automatically sets this environment variable based on the selected project when it is used instead of manually invoking GSMAKE.

SYSTEM Subdirectory

The `SYSTEM` subdirectory contains the assembly source files for the core system BIOS..

The `OBJ` subdirectory in the `SYSTEM` directory is used to hold the OBJ files during the build process; they are generated by the assembly process and used by the LINK process, but they are not output files from the build.

> *Note: In order to process the MAKEFILE properly, you must be sure to use the GSMAKE.EXE program supplied with this Adaptation Kit; do not use NMAKE.EXE or some other MAKE.EXE supplied by your compiler vendor. The General Software GSMAKE utility can read the enhanced MAKEFILEs used in building this software to run both the Microsoft and Borland development tools based on the BORLAND= environment variable. If you define this variable, then Borland tools will be used; otherwise, Microsoft tools will be used.*

SYSTEM32 Subdirectory

The `SYSTEM32` subdirectory contains subdirectories, each containing the assembly source files for a 32-bit BIOS.

The following subdirectories (and maybe additional ones since this printing) are defined in the SYSTEM32 subdirectory:

INC - Contains the common include files for the 32-bit components.
PCI - Contains the source code for the 32-bit PCI services component.

None of the 32-bit BIOS components are a necessary component of the 16-bit BIOS; these extensions of the BIOS architecture provide support for things like 32-bit BIOS Directory Services, 32-bit PCI Services, and other things.

Note: These components are built using 32-bit assemblers and linkers. If you are using Borland tools, your TASM32 and TLINK32 tools will be invoked by the MAKEFILE. If you are using Microsoft tools, MASM and LINK32 will be invoked. Please note that the Microsoft assembler you purchased may not include a LINK32.EXE program. If this is the case, you may use the LINK32.EXE program supplied with your Microsoft Visual C++ compiler. For the latest information on how to obtain these build tools, contact General Software.

Note: In order to process the MAKEFILE properly, you must be sure to use the GSMAKE.EXE program supplied with this Adaptation Kit; do not use NMAKE.EXE or some other MAKE.EXE supplied by your compiler vendor. The General Software GSMAKE utility can read the enhanced MAKEFILEs used in building this software to run both the Microsoft and Borland development tools based on the BORLAND= environment variable. If you define this variable, then Borland tools will be used; otherwise, Microsoft tools will be used.

INC Subdirectory

The INC subdirectory contains the common header files used by the EMBEDDED BIOS components. None of these files should ever be modified by the OEM. Two files, OPTIONS.INC and CONFIG.INC, contain defaults for configurable options. The OEM should never modify these files to change the defaults. Instead, the specific lines being changed should be copied into the appropriate project file, and changed in the project file copy.

Each configuration parameter in OPTIONS.INC or CONFIG.INC is coded as a symbol equated to a value. In the OPTIONS.INC file, almost all of the symbols are set to 0 to disable the specific option, or 1 to enable it. In the CONFIG.INC, almost all of the symbols are set to a numeric parameter that fine-tunes the system.

CHIPSETS Subdirectory

The CHIPSETS subdirectory contains one or more subdirectories, each of which holds exactly one Chipset Personality Module (CPM). If you have purchased additional support modules to work with EMBEDDED BIOS, then there may be subdirectories underneath this subdirectory.

The name of a Chipset Personality Module's subdirectory underneath CHIPSETS is significant. It must be the same name as the chipset files contained in the subdirectory. Thus, if the module is named MYCHPSET, then the directory's name is MYCHPSET, and there must be two files in the subdirectory: MYCHPSET.ASM (containing the code), and MYCHPSET.INC (containing additional definitions for the code).

EMBEDDED BIOS comes with one Chipset Personality Module, NOCHPSET, that is used as a placeholder for systems that do not have a chipset. The adaptation engineer can use this file as a structured template for adding custom chipset support for any design.

No build process occurs in the CHIPSETS directory. Instead, these files are included using assembly directives in the SYSTEM\CHIPSET.ASM file.

CPUS Subdirectory

The CPUS subdirectory contains one or more subdirectories, each of which holds exactly one CPU Personality Module. If you have purchased additional support modules to work with EMBEDDED BIOS, then there may be subdirectories underneath this subdirectory.

The name of a CPU Personality Module's subdirectory underneath CPUS is significant. It must be the same name as the CPU files contained in the subdirectory. Thus, if the module is named MYCPU, then the directory's name is MYCPU, and there must be two files in the subdirectory: MYCPU.ASM (containing the code), and MYCPU.INC (containing additional definitions for the code).

EMBEDDED BIOS comes with one CPU Personality Module, NOCPU, that is used as a placeholder for systems that do not have a nonstandard CPU that requires additional setup or configuration programming.

No build process occurs in the CPUS directory. Instead, these files are included using assembly directives in the SYSTEM\CPU.ASM file.

BOARDS Subdirectory

The BOARDS subdirectory contains one or more subdirectories, each of which holds exactly one Board Personality Module. If you have purchased additional support modules to work with EMBEDDED BIOS, then there may be subdirectories underneath this subdirectory.

The name of a Board Personality Module's subdirectory underneath BOARDS is significant. It must be the same name as the board files contained in the subdirectory. Thus, if the module is named MYBOARD, then the directory's name is MYBOARD, and there must be two files in the subdirectory: MYBOARD.ASM (containing the code), and MYBOARD.INC (containing additional definitions for the code).

EMBEDDED BIOS comes with one Board Personality Module, NOBOARD, that is used as a placeholder for systems that do not require any nonstandard initialization or configuration other than the standard calls POST makes to the chipset and CPU modules.

No build process occurs in the BOARDS directory. Instead, these files are included using assembly directives in the SYSTEM\BOARD.ASM file.

TOOLS Subdirectory

The TOOLS subdirectory contains the build utilities necessary to supplement your assembler and linker development tools so that you can build a binary copy of EMBEDDED BIOS. The tools provided are as follows:

GSMAKE.EXE	- Program Maintenance Utility
GSMERGE.EXE	- Binary 16-bit/32-bit File Merge Utility
PERF.EXE	- Disk Performance Analyzer
BIOSLOC.EXE	- Locate Utility
BIOSSUM.EXE	- ROM BIOS Extension Checksum Tool

`BIOSMAP.EXE`	- Map File Analysis Utility
`DISKIMAG.EXE`	- Disk Image Utility
`BIOSTART.EXE`	- Windows-based BIOS configuration and build utility
`CVTBMP.EXE`	- DOS-based BMP->RLE conversion and BMP display utility
`PALCHECK.EXE`	- DOS-based BMP analyzer utility
`CHKRFD.EXE`	- DOS-based RFD image analyzer

The TOOLS directory must be added to your path prior to building the system so that the utilities are available to the build process from the SYSTEM directory. BIOStart may automatically perform this function for you depending on your host operating system. If you find that your DOS box can't access your tools, you will need to perform this function manually using your operating system's tools (i.e., right click on "My Computer", Properties, and then update the environment's PATH variable as necessary).

UTIL Subdirectory

The `UTIL` subdirectory contains the build files necessary to build any auxilliary utilities associated with EMBEDDED BIOS, such as the remote disk server device driver. A `MAKEFILE`, associated .LRF linker response files, and source files are provided.

The `UTIL` subdirectory contains an `OBJ` subdirectory, used to contain the temporary object files created by the assembler.

The two most important components of `UTIL` that you will be building are MFGDRV.SYS and HOST.EXE. Both of these utilities are used on the host side to make Manufacturing Mode work. It is necessary to build `COW` before building `UTIL`, because the `UTIL` build requires `COW` object files to complete.

COW Subdirectory

The `COW` subdirectory contains the build files necessary to build the Character-Oriented-Windows user interface system that is used by `UTIL\HOST.EXE`. A `MAKEFILE`, associated .LRF linker response files, and source files are provided.

The `COW` subdirectory contains an `OBJ` subdirectory, used to contain the temporary object files created by the C compiler.

RESOURCE Subdirectory

The `RESOURCE` subdirectory contains four subdirectories that contain graphical and other resources for use in combined BIOSes. These resources include Splash Screens, Advertisements, Icons, and IDF files. General Software provides some basic resources with this adaptation kit. Additional resources may be added to these directories by the OEM.

What's Next?

Once you've installed the software, you are ready to sit back and review the information in the next chapter, because its concepts will give you the minimal background necessary to understand what the BIOS does, and help identify what components you'll need in your BIOS adaptation.

After a leisurely reading of Chapter 3, go ahead and make short work of Chapter 4 (setting up tools), and then dive into Chapter 5, where you'll learn how to actually build a BIOS.

Chapter 3

KEY EMBEDDED BIOS CONCEPTS

This chapter presents an architectural overview of EMBEDDED BIOS. OEMs with an understanding of these concepts generally produce BIOSes more efficiently in two ways. First, an appreciation of all the functional issues is an important thing to have before starting a design, so that the design can accommodate those issues. Second, with this material as background, the OEM will have a longer view of the adaptation process. Understanding this material will make your adaptation move more smoothly.

You may also wish to take advantage of General Software's published white papers on selected design topics. Contact General Software for details.

3.1 Architectural Overview

EMBEDDED BIOS is functionally similar to the BIOS in a PC, in many ways. First, the BIOS tests and initializes all of the equipment on the system when power is applied. Once the system has been initialized, it transfers control to an operating system or application. Finally, it provides software services through architected mechanisms that allow the operating system and application to manipulate the hardware; for example, to perform floppy disk I/O, read keystrokes from the keyboard, and display characters on a video display.

Because the BIOS is ultimately responsible for managing the hardware, it must implement policies for initialization and management of the devices. For example, the BIOS's *memory model* determines how much memory will be available to operating systems and applications, and where the memory will be located in the address space.

Similarly, its *interrupt model* determines the policy used to make interrupt assignments of external hardware devices, establish their priorities, and define how operating system and application software will request services from the BIOS.

The BIOS *Power-On Self-Test* (commonly, POST) is responsible for testing and initializing the hardware components in the target such as the DMA controllers, interrupt controllers, programmable timers, and other components so that they work together to provide a viable environment. For example, if dynamic RAM (DRAM) is used in a design, it must be

periodically refreshed; this is the responsibility of the BIOS. Using configuration options, the developer directs POST to provide refresh through on-board CPU functions, through chipset functionality, or using more elaborate techniques such as tying an 8254 programmable interval timer to an 8237 DMA controller to cause DMA cycles to perform the refreshing. POST sets up the policies to be used for performing DRAM refresh and many other tasks so that operating systems and applications don't have to do these tasks by themselves.

These and many other architectural issues are described in detail in this chapter.

3.1.1 Memory Model

EMBEDDED BIOS employs a memory model that is compatible with desktop PC standards. Because the BIOS is used primarily in a real-mode environment, it does not define any standards for the use of extended memory beyond 1MB. Instead it is concerned with the layout and usage of memory below 1MB in the address space.

Because Intel-architecture processors can be programmed to respond to a variety of different kinds of addresses (physical, linear, virtual, and real-mode addresses), we will refer to 32-bit physical addresses whenever describing where some object is located in the target machine. When referring to how the object is referenced with actual machine instructions, we will use what is called 16:16 notation for addresses. In this format, addresses contain two parts, each 16 bits in width. The first 16-bit entity is a segment address, and the second 16-bit entity is a byte offset relative to the specified segment. A segment address can be transformed into a physical address by multiplying it by 16 (10h in hexadecimal).

3.1.1.1 The Interrupt Vector Table

At physical location 00000000h in the address space is the real-mode Interrupt Vector Table, or IVT. This table is defined by Intel 80x86 architecture and by other PC standards to be an array of far (16:16) pointers to objects, some being Interrupt Service Routines (ISRs), while other elements are pointers to data structures. This table contains 256 elements and each element is four bytes long, so the table is exactly 1KB in size.

3.1.1.2 The BIOS Data Area

The first address immediately following the IVT is 00000400h. Addressed with the equivalent real-mode segment 0040h, the space following the IVT is called the BIOS Data Area, or BDA. The BDA is used by the BIOS to keep track of how the system is configured; i.e., how many serial and parallel ports exist. It is also used to keep track of the state of the running BIOS, such as the track number over which a floppy disk recording head is positioned. The BDA extends up to but not including physical address 00000500h, so that the first free address to be used by operating systems and application program is 00000500h.

All the fields in the BDA are architected by IBM. Slight modifications to this area have been made by other desktop BIOS vendors since PC clones have matured, to accommodate new BIOS functionality. When these modifications become industry-standard on the desktop, they are incorporated into the EMBEDDED BIOS BDA.

3.1.1.3 Free Low RAM

Starting at physical address 00000500h, or segment 0050h, operating systems and user programs use memory as they see fit. The amount of memory, or size of free low RAM (including the IVT and BDA), is kept in the BIOS Data Area by the BIOS itself, and can be retrieved with a BIOS software service (INT 12h).

3.1.1.4 The Extended BIOS Data Area

The last several KB of low memory are reserved by the BIOS for extending the BIOS Data Area without interfering with the well-established user address, 00000500h. During POST, the BIOS determines the amount of low RAM, and reserves the top 1KB of this RAM for itself. When the operating system or user application use the INT 12h BIOS service to determine the amount of low memory, the BIOS actually returns 1KB less than is actually present. In a desktop PC environment, the Extended BIOS Data Area usually ends at physical address 000A0000h to make room for video adapter hardware such as the VGA screen regeneration memory). In designs that do not have VGA hardware at segment A000h, additional memory can be mapped to this address space by the hardware (or possibly by the chipset), so that the BIOS can provide access to a larger amount of low memory.

3.1.1.5 Expanded Memory

In the 1980's a standard emerged for add-on memory cards that provided 64KB pages of memory within the memory range 000A0000h - 000E0000h called expanded memory. Several application programs, such as Lotus 1-2-3 and Windows for example, took advantage of this memory to store program data while they were running. This standard was primarily for application programs, but operating systems evolved to manage this memory. The BIOS, however, never manages this memory by itself (EMBEDDED BIOS does not provide any support for EMS by itself).

3.1.1.6 Video ROM Extensions

Physical address 000C0000h or 000E0000h is inspected by the BIOS during POST for the presence of a possible EGA or VGA ROM BIOS Extension. By checking for a special signature and checksumming the ROM, the BIOS determines if the ROM exists, and if so, it is invoked by the BIOS POST to initialize any video hardware that the core system BIOS is not aware of. For example, the common VGA screens used in desktop PCs are actually not directly supported by the video BIOS on the PC motherboard; instead, the video ROM BIOS Extension on the VGA controller card hooks the BIOS service (INT 10h) so that it can handle video requests instead of the system BIOS.

If a video ROM is not detected by the BIOS, and video services are enabled by the adaptation engineer, then the default video routines in the video module of the BIOS are used to provide video service for monochrome and color graphics adapters.

3.1.1.7 Other ROM Extensions

Additional ROM extensions are detected by POST during system initialization within a special address range (usually 000C8000h - 000EE000h) at 2KB intervals using a special signature pattern and checksum technique. When valid ROM extensions are found, they are called just as video ROM extensions are called, and they perform operations as necessary to support their

function. For example, SCSI disk controllers may have ROM BIOS extensions to provide basic disk services (INT 13h) so that the bootstrap process can actually boot from a SCSI device. Similarly, network interface cards (NICs) may have a remote boot ROM that gets control as a ROM extension so that it can initialize the NIC and request a download of the operating system over a network.

3.1.1.8 The System ROM

The BIOS itself is stored in ROM so that it fits neatly at the end of the 1MB address space. Typically, a 64KB ROM such as a 27C512, or a 128KB bulk Flash part such as a 28F010, is used to hold the system BIOS code itself. This code receives control at power-on reset time at physical address 000FFFF0h; this address is equivalent to the 16:16 address F000:FFF0.

On 80386 and above CPUs, the high bits of the physical address are all set, requiring the glue hardware surrounding the CPU to either double-map the ROM BIOS into the top of extended memory, or to disable the high bits so that the CPU really boots from the top of the lower 1MB address space.

Regardless of how the CPU gets control, the system ROM usually occupies 64KB-128KB, although the BIOS may be configured to use from 16KB to 256KB of that total file's size with a build option. Naturally, features must be removed from a full-featured BIOS to allow its size to be reduced to arbitrarily small sizes.

3.1.1.9 Extended Memory

Just as the BIOS sizes low memory below 1MB during POST, it also determines the amount of RAM above the 1MB address line and keeps this size in CMOS, if available. The amount of usable extended memory is returned through a BIOS software service (INT 15h, function 88h), although the BIOS does not provide any other services for managing this memory beyond simple data copying functionality (INT 15h, function 87h). There is an additional service defined by the ACPI specification (INT 15h, function E820h) that can report memory beyond the amount that INT 15h function 88h can report. This is important because function 88h can only report up to 63MB of extended memory. This function, called SMAP, is supported by EMBEDDED BIOS.

The management of extended memory is the function of operating system software such as HIMEM.SYS. This driver is available in the Embedded DOS-ROM source tree.

3.1.1.10 CMOS Memory

Actually separate from the memory address space of the processor, an amount of battery-backed CMOS RAM is usually available in AT-compatible systems. In such a compatible configuration, this memory is accessed by reading and writing to I/O ports 70h and 71h.

The BIOS uses this memory to store the equipment configuration and user options associated with the operation of the BIOS, and the integrated BIOS Setup screen system is used to edit the CMOS memory in a running system.

3.1.2 Interrupt Model

In addition to defining the way memory is used in a system, EMBEDDED BIOS has an interrupt model for receiving BIOS service requests via software interrupts, handling CPU traps and faults, processing device hardware interrupts, and managing points in the IVT that point to data structures used by BIOS service modules.

The following table shows the IVT entries used by EMBEDDED BIOS. Note that some interrupts (notably, vectors 08h through 12h) are used by the BIOS although they also be be generated by the CPU in protected mode circumstances.

Vector	Type	Function or Service
00h	CPU	Divide by zero trap
01h	CPU	Single-step trap
02h	CPU	NMI interrupt
03h	CPU	Breakpoint trap (INT 3)
04h	CPU	Arithmetic overflow trap
05h	CPU	Array bounds exception
06h	CPU	Invalid Opcode Trap
07h	CPU	Device Not Available Trap
08h	IRQ0	18.2 Hz Timer Tick
09h	IRQ1	Keyboard
0ah	IRQ2	Cascaded to PIC 2
0bh	IRQ3	COM2 Serial Port
0ch	IRQ4	COM1 Serial Port
0dh	IRQ5	LPT2 Parallel Port
0eh	IRQ6	Floppy Disk Controller
0fh	IRQ7	LPT1 Parallel Port
10h	Service	Video Services
11h	Service	Equipment List Service
12h	Service	Low Memory Size Service
13h	Service	Floppy/IDE/ROM/Remote Disk Services
14h	Service	Serial Port Services
15h	Service	General Services, Up-Calls
16h	Service	Keyboard Services
17h	Service	Parallel Port Services
18h	Up-Call	Boot Fault Up-Call
19h	Up-Call	Bootstrap Up-Call
1ah	Service	Date/Time Services, PCI Services
1bh	Up-Call	Control-Break Up-Call
1ch	Up-Call	18.2 Hz Application Timer Up-Call
1dh	Table	Pointer to Video Control Param Table
1eh	Table	Pointer to Diskette Parameter Table
1fh	Table	Pointer to Video Graphics Table
20h-3fh	DOS	-- reserved by DOS --
40h	Redirector	Floppy disk services redirected by IDE
41h	Table	Fixed Disk Parameter Table (Drive 80h)
42h	Extension	EGA Default Video Driver
43h	Extension	Video Graphics Characters
44h-45h	N/A	-- not used --
46h	Table	Fixed Disk Parameter Table (Drive 81h)
47h-49h	N/A	-- open --
4ah	Up-Call	User Alarm
4bh-6fh	N/A	-- open --

70h	IRQ8	Real-Time Clock Interrupt (1 Khz)
71h	IRQ9	-- open --
72h	IRQ10	-- open --
73h	IRQ11	-- open --
74h	IRQ12	PS/2 Mouse
75h	IRQ13	Math Coprocessor
76h	IRQ14	IDE Drive Controller
77h	IRQ15	APM Suspend Request
78h-ffh	N/A	-- open --

3.1.2.1 BIOS Service Interrupts

The BIOS receives requests to perform functions through software interrupts. Software interrupts, generated by the operating system or by a user application, are generated with INT *nn*h instructions, where *nn*h is a number that is assigned to a specific type of service, such as 16h for keyboard input, 10h for video output, or 13h for disk I/O.

In most cases, a BIOS service has multiple functions. For example, the disk BIOS service interrupt supports resetting the device, reading data from the media, writing data to the media, and checking the type of media inserted into the drive. For multifunction BIOS services, the requesting application places a function code in the AH CPU register, fills other registers as necessary with operands, and executes the appropriate software interrupt for the service. When the service completes, it returns to the caller to execute the instruction following the software interrupt.

Upon return, the BIOS services return status or other information in CPU registers, many times including the CPU flags register. For example, when an INT 13h disk read function is requested to read from a disk that has been removed from the drive itself, the disk BIOS returns with the carry flag set (CY) and a disk subsystem error code in the AH register. If the function were to complete successfully, then the carry flag would not be set (NC). Remember that not all BIOS services use the same return status conventions; therefore, you should consult the service reference in Chapter 22 for complete details.

3.1.2.1.1 INT 10h, Video Services

All video functions are provided through the INT 10h software interrupt mechanism. The caller provides a function code in the AH CPU register and specifies operands as appropriate for the given function in other CPU registers before issuing the INT 10h instruction.

EMBEDDED BIOS actually begins handling an INT 10h request in its CONIO module, which determines whether the video should be redirected over a serial link. This console redirection enables embedded systems that don't have a real MDA, CGA, EGA, or VGA video system to display their output via more inexpensive means. Console redirection may play a part in the final shipped embedded product, or it may simply be used during development and test in liu of an actual PC keyboard and screen.

If CONIO determines that the INT 10h service should not be redirected to a serial device, then it passes control to one of the modules that handle video controllers, such as module VIDEO, which manipulates the 6845 CRT controller registers directly to manage the display. Actual writing of data to the video screen and reading characters from the screen is accomplished by memory reads and writes to video regeneration memory, mapped into the memory address space

at physical address 000b0000h for monochrome output, or 000b8000h for color output. Both monochrome and color adapters may be present in a system, in which case using the INT 10h set mode function can be used to switch between the displays.

If CONIO determines that INT 10h services should be redirected, then it calls the SERIAL module to perform the work of transmitting characters to the remote terminal equipment. In addition to writing characters to the display, the BIOS also supports the set cursor address function, and several other functions that manipulate the video display in some manner. These functions are translated to ANSI escape sequences that are transmitted to the remote terminal equipment just as other data characters via the SERIAL module's services through INT 14h.

The basic functions provided by the INT 10h BIOS are given below:

Function	Video Service
00h	Set Video Mode
01h	Set Cursor Type
02h	Set Cursor Position
03h	Return Cursor Position
04h	Return Light Pen Condition (not in core BIOS)
05h	Set Current Video Page
06h	Scroll Up Region
07h	Scroll Down Region
08h	Return Character and Attribute
09h	Write Character and Attribute
0ah	Write Character
0bh	Set Color Palette
0ch	Write Graphic Pixel (not in core BIOS)
0dh	Read Graphic Pixel (not in core BIOS)
0eh	Write Character Only
0fh	Return Video Display Mode

3.1.2.1.2 INT 11h, Equipment List Service

The BIOS provides a way for the application to determine what equipment is available through the INT 11h software interrupt mechanism. Unlike many of the other BIOS software interrupts, INT 11h does not require a function code or any operands. Instead, it returns a bit mask in its AX CPU register that can be inspected to determine what equipment is supported by BIOS services. For a complete description of this function, see Chapter 22.

The equipment list is stored in the BIOS Data Area (BDA) by POST during system initialization in a 16-bit field called **DevFlags**. ROM Extensions that extend BIOS services to support additional equipment must edit this field if the equipment is to be made available to the operating system or application.

3.1.2.1.3 INT 12h, Low Memory Size Service

The BIOS returns the amount of physical memory below the 1MB boundary (exclusive of the 1KB Extended BIOS Data Segment) in response to the INT 12h software interrupt. Like INT 11h (Equipment List), this software interrupt returns its information in the AX CPU register and does not accept function codes or operands. See Chapter 22 for full details.

The low memory size is stored in the BIOS Data Area (BDA) by POST during system initialization in a 16-bit field called **LowMemorySize**. ROM Extensions or other software that uses memory from the end of available low memory must reduce this field by the amount of memory reserved so that the operating system and applications will not overwrite the reserved memory. This technique is used by the BIOS itself during POST to establish the Extended BIOS Data Area (EBDA), a 1KB region located at the top of physical low memory.

3.1.2.1.4 INT 13h, Disk Services

All mass-storage devices, including floppy disk, hard disk, ROM disks, RAM disks, RFD disks, and OEM-defined disks, are accessed through the INT 13h software interrupt. As with the video services, INT 13h services accept a function code in the AH CPU register, with operands appropriate to a given function placed in the other CPU registers before executing the INT 13h instruction.

The following functions are supported by the FLOPPY disk driver (note that gaps in the function numbers indicate unassigned functions for floppy I/O):

Function	Floppy Disk Service
00h	Reset Floppy Controller
01h	Read Last Status
02h	Read Sectors
03h	Write Sectors
04h	Verify Sectors
05h	Format Track
08h	Read Drive Parameters
15h	Read Drive Type
16h	Determine Media Change
17h	Set Disk Type
18h	Set Media Type for Format

The following functions are supported by the IDE disk driver (note that gaps in the function numbers indicate unassigned functions for hard drive I/O):

Function	IDE Disk Service
00h	Reset IDE Controller
01h	Read Last Status
02h	Read Sectors
03h	Write Sectors
04h	Verify Sectors
05h	Format Track
08h	Read Drive Parameters
09h	Initialize Parameters
0ah	Read Long Sectors
0bh	Write Long Sectors
0ch	Seek to Cylinder
0dh	Alternate Reset
10h	Test Drive Ready

14h	Run Controller Diagnostic
15h	Read Disk Type
41h-48h	Extended Disk Functions

The following functions are supported by the ROM disk driver (note that write-oriented functions return a write-protected status for the ROM disk):

Function	ROM Disk Service
00h	Reset ROM Disk
01h	Read Last Status
02h	Read Sectors
04h	Verify Sectors
08h	Read Drive Parameters
15h	Read Drive Type
16h	Determine Media Change
18h	Set Media Type for Format

The following functions are supported by the RAM disk driver:

Function	RAM Disk Service
00h	Reset RAM Disk
01h	Read Last Status
02h	Read Sectors
03h	Write Sectors
04h	Verify Sectors
08h	Read Drive Parameters
15h	Read Drive Type
16h	Determine Media Change
18h	Set Media Type for Format

The following functions are supported by the Resident Flash Disk (RFD) driver:

Function	RFD Disk Service
00h	Reset Flash Disk
01h	Read Last Status
02h	Read Sectors
03h	Write Sectors
04h	Verify Sectors
08h	Read Drive Parameters
15h	Read Drive Type
16h	Determine Media Change
18h	Set Media Type for Format

Disk I/O is handled by different code modules in the BIOS, depending on whether a specific request is directed at a floppy device, an IDE hard drive, a ROM disk, a RAM disk, or a Resident Flash disk. During POST, the FLOPPY1/2/3, IDE1/2, ROMDISK, RAMDISK, and RFD1/2 modules are initialized if enabled through CMOS. POST maps these servers to specific drives when CMOS is scanned.

Disk I/O is logically divided into two types: floppy-compatible and hard drive-compatible. Traditionally, DOS requires floppy-compatible drives to have a FAT file system layout with a Partition Boot Record (PBR) in the first sector, two File Allocation Tables (FATs) following the

PBR, and a root directory following the FATs. Hard drives are expected to be partitioned, and have a different logical layout. Starting with a Master Boot Record (MBR) in the first sector that contains a Partition Table, the remainder of the hard disk is divided into partitions that each have their own format logically similar to floppy disks. Each partition starts with a PBR, two FATs, and a root directory.

Because floppy disks and hard drives have different information organizations, the BIOS separates them into two sets of devices. Disks numbered 00h, 01h, 02h, and so on, are floppy drives, and DOS can expect floppy-style file systems on them. Disks numbered 80h, 81h, 82h, and so on, are hard drives, and DOS expects them to have an MBR, not a PBR, in the first sector.

The EMBEDDED BIOS ROM drive simulates one or more disks by treating the INT 13h read sectors function as simply a memory copy from OEM-specified areas of ROM to the application's data buffer. The memory image for each disk is created by the adaptation engineer using the DISKIMAG utility (provided with the EMBEDDED BIOS Adaptation Kit). ROM disks can be either soft (formatted like a floppy disk) or hard (formattted like a hard drive). The **FILE_SYSTEM** table entry in the project file that defines a ROM disk has a parameter that specifies whether the image is formatted as a floppy or a hard disk.

The EMBEDDED BIOS RAM drive is similar to the ROM drive, except that it supports both reading and writing. Build options specify the location of the RAM in the address space, and if automatic formatting is to be used by the BIOS during POST, in the event the RAM disk contents are not properly initialized.

The EMBEDDED BIOS Resident Flash Disk (RFD) operates only on Flash media, and can simulate both floppy disks and hard drives up to 32MB in size with a special wear-leveling algorithm that is built right into the core BIOS. Support for Flash media is provided through a Media Control Layer (MCL), which in turn calls Media Technology Drivers (MTDs) to perform the low-level I/O to the Flash. The **FILE_SYSTEM** table entry in the project file that defines a RFD has a parameter that specifies whether the image is formatted as a floppy or a hard disk.

The OEM can also implement special file system drivers and integrate them into the BIOS disk system without modifying the core BIOS source code. This is done by adding code to the board module, and then adding a special **FILE_SYSTEM** table entry in the project file that refers to the new file system's entrypoint.

3.1.2.1.5 INT 14h, Serial Port Services

All serial I/O functions are provided by the BIOS through the INT 14h software interrupt mechanism. As with disk drives, serial ports are numbered; the logical port numbers are 00h for COM1, 01h for COM2, 02h for COM3, and 03h for COM4.

The serial I/O service accepts a function code in the AH CPU register and operands in other registers. The logical port number is normally passed in the DX CPU register, so that the serial service can operate on a specific serial port. The following table shows a summary of the serial port services:

Function	Serial Port Service
00h	Initialize Serial Port
01h	Send Character
02h	Receive Character

03h	Read Port Status
04h	Extended Initialize
05h	Manipulate Modem Control Register

Upon return from the INT 14h instruction, status is returned in a complex way, with the AX CPU register containing both a Line Status Register and a Modem Status Register. Because this service exposes actual bit patterns used in the 8250, serial ports tied to incompatible UARTs (such as those on the 80C186-EC CPU) are supported by translating the status returned by such a UART into the the most-equivalent bitmask that would correspond to the 8250's status registers.

The BIOS handles INT 14h requests in the SERIAL module. This module translates logical port numbers into physical port numbers by indexing into the **ComPorts** array in the BIOS Data Area. Physical port numbers above 10h are assumed to be handled by the 8250 UART module, whereas port numbers 00h-10h are assumed to be handled by the CPU personality module. Thus, serial I/O requests are distributed on-the-fly to the appropriate hardware handler based on the configuration data in the BIOS Data Area.

The original IBM BIOS supported serial port data transfer rates through 9600 baud. The baud rate, as well as other communications parameters associated with a serial port, are configured using an INT 14h Initialize Serial Port function. EMBEDDED BIOS supports this standard as well as the Extended Initialize INT 14h function supported by modern desktop PC BIOS implementations, allowing higher baud rates through 115K baud.

3.1.2.1.6 INT 15h, General System Services

Often called a catch-all general service, the INT 15h software interrupt is actually used two ways: one where the application requests services, and another where the BIOS notifies the application that it is about to enter or leave a spin-loop in order to wait for a device to complete a task that will take some amount of real time. These "up-calls" or "call-outs" as they are sometimes called, interrupt the user application, which may choose to "hook" INT 15h to receive the notification, or not hook INT 15h, and therefore not be informed about the spin-loops. See the section on "BIOS Up-Calls" later in this chapter for further details.

The INT 15h services used by the application are implemented by the BIOS. These services are diverse; from returning the amount of available extended memory above 1MB, moving memory from one physical address to another, and switching into protected mode, to returning the address of the Extended BIOS Data Segment, returning the System Configuration Table (SCT) address, and manipulating the watchdog timer (see Chapter 22 for programming details). These services are all requested by placing a function code in the AH CPU register, setting other CPU registers to operand values, and executing an INT 15h instruction. Upon return, status is returned in several different ways; consult Chapter 21 for details. A summary of INT 15h services is shown in the following table.

Function	General Service
24h	Query A20 Port 92h Support
4fh	Scancode Translate Up-Call
53h	Advanced Power Management
85h	System Request Key Up-Call
86h	Wait Micro Interval
87h	Protected Mode Memory Block Move
88h	Return Extended Memory Size in KB

89h	Switch to Protected Mode
90h	Device Busy Up-Call
91h	Device Interrupt Up-Call
A0h	Read/Write CMOS Cell
A1h	Set Current I/O Redirection
A3h	Get Embedded BIOS Version
A4h	Query Embedded DOS-ROM file system
C0h	Return System Configuration
C1h	Return Extended BIOS Data Area
C2h	PS/2 Mouse
C3h	Enable/Disable Watchdog Timer
D0h	Breakpoint into BIOS Debugger
D8h	EISA Slot Configuration
E0h	Resident Flash Array Functions
E8h	Get Extended Memory Information
FFh	Print Character for Embedded DOS-ROM Debug I/O

All INT 15h requests are dispatched by module MISC. Of course, if the operating system or application "hooks" IVT entry 15h so that it can receive up-calls, then it will also receive other function requests as well, and should pass them on to the BIOS in a "chained" approach.

Some functions are routed by MISC to the PROTMODE module, which handles steady-state protected mode processing in the BIOS. PROTMODE is complex because it must deal with several different mode switching techniques, and must also save the state of the CPU cache across mode switches. For details about what methods are available, consult Chapter 7.

3.1.2.1.7 INT 16h, Keyboard Services

All keyboard I/O functions are provided through the INT 16h software interrupt mechanism. Before executing an INT 16h instruction, the application places a function code in the AH CPU register, and other operands as appropriate in other CPU registers. Upon return from the INT 16h software interrupt, the status is returned in special ways, including through the Zero flag in the CPU. See Chapter 21 for programming details. A summary of INT 16h services is shown in the following table.

Function	Keyboard Service
00h	Read Character
01h	Return Keyboard Status
02h	Return Keyboard Flags
03h	Set Keyboard Typematic Rate
05h	Push Character/Scancode to Buffer
10h	Enhanced Read Character
11h	Enhanced Write Character
12h	Enhanced Return Keyboard Flags
f0h	Set CPU Speed
f1h	Get CPU Speed
f4h	Cache Control

As the table indicates, several keyboard functions in fact don’t manipulate the keyboard. Instead, they manipulate other system components, such as the CPU’s clocking and the system’s cache. Because these features were commonly implemented in the 8042 keyboard controller of many

desktop PC systems, their controlling BIOS functions were added to the INT 16h services. The CPU speed functions are routed to the HELPER module, and the cache control function is routed to the CACHE module.

As with video output through INT 10h, EMBEDDED BIOS is able to support a real PC or AT keyboard, or it can redirect INT 16h services over a serial port. Module CONIO receives the application INT 16h requests and determines how keyboard requests are to be serviced. If redirection to a serial port is enabled, then it calls the SERIAL module to read a character from a serial port or determine the serial port's status.

3.1.2.1.8 INT 17h, Parallel Port Services

All parallel port I/O services are provided through the INT 17h software interrupt mechanism. A function code is passed by the application in the AH CPU register, with additional operands as required in other CPU registers. In particular, the DX register is programmed with a logical printer port number, where 0=LPT1, 1=LPT2, and 2=LPT3. The following table shows the functions available in the INT 17h service family:

Function	Parallel Port Service
00h	Write Character
01h	Initialize Parallel Port
02h	Return Parallel Port Status

INT 17h requests are handled by the BIOS through module PARALLEL. This module translates the logical parallel port number into a physical port I/O address, and then manipulates that port directly to perform the function. Parallel port hardware is expected to be compatible with the IBM hardware.

3.1.2.1.9 INT 18h, Boot Fault Routine

After POST initializes the system, it calls INT 19h to boot the operating system from the appropriate device. If the INT 19h service fails to load the operating system, then the BIOS (or the operating system boot record) executes an INT 18h instruction, so that the ROM BIOS can regain control and perform an alternate function.

By default, EMBEDDED BIOS initializes the INT 18h function to a routine that prints "No boot device available.", and prompts to enter the debugger or SETUP system, or reboot the system.

At any point prior to the boot process, user-written code, such as code in ROM BIOS Extensions, can "hook" the INT 18h interrupt vector and gain control in this situation, thereby replacing the default handler in the BIOS. In the original PC, INT 18h jumped to a separate ROM that contained ROM BASIC. The embedded system developer might use this mechanism to execute application code from ROM in the event of a boot device failure.

3.1.2.1.10 INT 19h, Bootstrap Routine

After POST initializes the system, it calls module BOOTOS, which executes an INT 19h instruction to load the operating system or start the embedded application.

By default, EMBEDDED BIOS initializes the INT 19h function to a routine in module BOOTOS that cycles through six boot actions defined in CMOS cells. The six boot actions are attempted in order until one is successful. Keep in mind that, if ROM extensions such as DOS hook INT 19h so that they can get control when the system boots, then BOOTOS will not receive control to cycle through all of the boot actions, and the boot action sequence will be defeated.

Boot actions are Boot from any drive (A: through K:), Boot Windows CE in ROM, Boot Embedded DOS-ROM out of ROM, Enter Manufacturing Mode, Enter Debugger, and No Action.

The boot actions for drives A: through K: read one sector from the drive at sector 1, head 0, track 0 into physical memory location 00007C00h. If the read is successful and if the boot record contains the byte sequence 55h, aah, as the last two bytes in the 512-byte sector, then control is transferred to the boot record at 16:16 address 07C0:0000, and the BIOS plays no further role in the bootstrap process.

There are two ways to boot Windows CE with EMBEDDED BIOS. The first way is selected as a boot action "Boot Windows CE." The second way is by enabling a feature in SETUP that instructs EMBEDDED BIOS to attempt to find the Windows CE system file (NK.BIN) on disks that BOOTOS is told to boot from. If NK.BIN can be found during POST, it will be loaded into memory and booted. Otherwise, the boot record for that drive will be loaded and booted. This makes it possible to load Windows CE without loading DOS to run LOADCEPC. This feature of EMBEDDED BIOS is called "CE Ready."

As with the INT 18h interrupt vector, user-written code, such as code in a ROM BIOS Extension, can "hook" the INT 19h interrupt vector and gain control when it is time to load the operating system or start an embedded application. For systems with application code located and burned into ROM, the INT 19h vector can be hooked during POST and then be used as a way to receive control after the system has been initialized. This is how Embedded DOS-ROM receives control when it is configured to boot out of ROM.

3.1.2.1.11 INT 1ah, Time/Date Services

All time and date services are provided through the INT 1ah software interrupt mechanism. The application places a function code in the AH CPU register, and places any appropriate operands in other CPU registers, before executing an INT 1ah instruction. Upon return, INT 1ah services return their status in a complex way. The following table summarizes the available date/time services. Refer to Chapter 21 for complete programming details.

Function	Date/Time Service
00h	Return Ticks Since Midnight
01h	Set Ticks Since Midnight
02h	Return Time
03h	Set Time
04h	Return Date
05h	Set Date
b1h	PCI Services (architected by Intel)

The INT 1ah service manages the time and date as separate pieces of information, and in two ways. In systems with a PC-compatible Real Time Clock (RTC) component, the BIOS is

capable of reading the contents of the RTC and updating it under program control. Both the date, and the time, can be stored in this device.

In systems that do not have a RTC component (and in those that do), the system time is maintained in a different way as a 32-bit number that represents "the number of ticks since midnight", in a location in the BIOS Data Area. It is common for DOS to detect whether the RTC services are available, and then use the ticks since midnight value as a system time when the real time clock is not present. When available, the RTC is normally the preferred method of obtaining the time, and is the only way of obtaining the date, since the RTC part is usually kept running with a battery when the system is turned off.

As can be seen from the table, INT 1ah is also the access point for PCI services, available on some targets. Consult Chapter 21 for details.

3.1.2.2 Table Pointers

Not all IVT entries point to a BIOS service routine. Several BIOS-managed interrupt vectors actually point to data structures maintained by the BIOS. These data structures are the Video Parameter Table (VPT), the Diskette Parameter Table (DPT), Video Graphics Character Table (VGCT) and the Fixed Disk Parameter Tables (VDPTs).

3.1.2.2.1 INT 1dh, Video Parameter Table (VPT)

The Video Parameter Table (VPT) is used by the VIDEO module to program the 6845 CRT controller's internal registers according to the specific mode requested by the application. The VPT is pointed to by IVT entry 1dh, and may be changed by software such as a VGA ROM Extension that supports additional modes.

The default VPT used by the integrated VIDEO module is shown below (this table is found in module BIOS in the source code):

```
;       The following table contains parameters (indexed by the user mode)
;       to load into the 6845's 16 operating registers.  Vector 1dh points
;       to this table, and the user software may replace it.

        PUBLIC  VideoTbl
VideoTbl label  byte
        db      38h, 40, 2dh, 10, 1fh, 6, 19h, 1ch, 2, 7, 6, 7, 0, 0, 0, 0
        db      71h, 80, 5ah, 10, 1fh, 6, 19h, 1ch, 2, 7, 6, 7, 0, 0, 0, 0
        db      38h, 40, 2dh, 10, 7fh, 6, 64h, 70h, 2, 1, 6, 7, 0, 0, 0, 0
        db      61h, 80, 52h, 15, 19h, 6, 19h, 19h, 2, 13, 11, 12, 0, 0, 0, 0
```

3.1.2.2.2 INT 1eh, Floppy Diskette Parameter Table (DPT)

IVT entry 1eh points to the current Diskette Parameter Table, or DPT, being used by the floppy disk BIOS. Because there are potentially several floppy drives in a system, the DPT defines the operational characteristics of the floppy currently being accessed.

The DPT pointer in the IVT is used by more than just the FLOPPY module. During the initialization of DOS, it copies the ROM-based DPT established by the BIOS into its own RAM

buffer, and re-points the 1eh vector to the RAM location. This allows it to modify the default DPT before performing diskette operations so that they can be optimized.

Reestablishing the DPT in RAM serves another purpose as well. Softguard copy-protection relies on the fact that the DPT is copied into RAM by DOS, and edits the DPT pointed to by the 1eh vector to tell the BIOS that it will be reading 128-byte sectors during its check for a special, 128-byte sector on a certain track of a release diskette. When it is done calling INT 13h services to verify that this sector exists, then it restores the DPT to its original state.

Clearly, the DPT is not an architecturally sound way of providing more control over floppy disk services provided by the BIOS. Unfortunately, this architectural relic was firmly established with the first IBM PC BIOS and must be provided in other BIOS products.

The format of the DPT is shown below in an assembly language structure. This structure can be found in your INC directory in the STRUC.INC header file.

```
;       Diskette parameter table structure format.

DPT             struc
dpt_specify1    db      ?               ; specify command 1.
dpt_specify2    db      ?               ; specify command 2.
dpt_motoroff    db      ?               ; motor off time.
dpt_bps         db      ?               ; bytes per sector (coded, above).
dpt_spt         db      ?               ; sectors per track.
dpt_gap         db      ?               ; gap length between sectors.
dpt_dtl         db      ?               ; data length (always ffh).
dpt_gap3        db      ?               ; gap length for FORMAT.
dpt_fill        db      ?               ; fill byte for FORMAT.
dpt_headsettle  db      ?               ; head settle time.
dpt_motoron     db      ?               ; motor-on start time.
dpt_maxtrack    db      ?               ; max track number for this drive.
dpt_drr         db      ?               ; data transfer rate.
dpt_unused1     db      ?               ; unused byte.
dpt_unused2     db      ?               ; unused byte.
DPT             ends
```

The fields in the DPT are actually used as operand bytes when the FLOPPY1, FLOPPY2, and FLOPPY3 modules send commands to the Intel 82077A or 82078-compatible floppy disk controller. The specific values for each field are governed by the established standards for recording information on DOS-compatible floppy disks for the various drive types and media types. By manipulating these fields, the application program can cause the BIOS to read, write, format, and verify nonstandard media. For exact specifications on the values to be stored in the DPT, consult the Intel documentation on the 82077A or 82078 floppy disk controllers.

3.1.2.2.3 INT 1fh, Video Graphics Character Table (VGCT)

The Video Graphics Character Table (VGCT) is pointed to by IVT entry 1fh, and is used by VGA ROM BIOS Extensions to define the shape of the IBM-compatible character set when in graphics modes. When in character modes, the built-in VIDEO module in the BIOS does not use this entry. If you are internationalizing your adaptation of EMBEDDED BIOS to foreign character sets, this is the table to change the fonts for standard BIOS resolutions.

3.1.2.2.4 INT 41h/46h, Fixed Disk Paramter Tables (FDPTs)

IVT entries 41h and 46h are used in versions of the BIOS that support IDE drives, so that operating system software can determine the fixed disk drive types. Introduced by IBM with the IBM PC/AT Personal Computer, IVT entry 41h points to a data structure that describes the primary hard drive (drive 80h) and IVT entry 46h points to a data structure that describes the secondary hard drive (drive 81h). These structures should not be used by application software, and are rarely used by operating system software, since INT 13h function 08h can provide substantially the same information about both floppies and fixed disks in the system.

To maintain compatibility with the IBM PC/AT BIOS, EMBEDDED BIOS establishes these vectors to point to structures with the following format (see module IDE1 for how they are created):

```
FDPT          STRUC
fdpt_cyl      dw     ?            ; maximum number of cylinders.
fdpt_hd       db     ?            ; maximum number of heads.
              dw     ?            ; reserved, MBZ (not used, see PC/XT).
fdpt_wp       dw     ?            ; starting cyl for write precompensation.
              db     ?            ; reserved, MB. (max ECC data burst length).
              db     ?            ; DTE_CONTROL.
              db     ?            ; reserved, MBZ.
fdpt_cap      dw     ?            ; disk capacity in megabytes.
fdpt_lz       dw     ?            ; landing zone cylinder.
fdpt_spt      db     ?            ; sectors per track.
              db     ?            ; reserved.
FDPT          ENDS
```

3.1.2.3 BIOS Upcalls

While nearly all of the software interrupts associated with the BIOS are invoked by the operating system or application and serviced by the BIOS itself, there are a few software interrupts that are actually generated by the BIOS, and may be "hooked" by the operating system or the application. These software interrupts, called "up-calls" or "call-outs", are used to notify application software that events in the BIOS have occurred.

3.1.2.3.1 INT 15h Device Management

The INT 15h software interrupt is a two-way interrupt service. Functions such as 87h, 88h, and 89h are made by the application and serviced by the BIOS to provide protected mode support. Other functions, such as 90h and 91h, are invoked by the BIOS, and "hooked" by DOS or by application software to be notified when events inside the BIOS occur. These invocations of INT 15h functions by the BIOS are called "up-calls", or simply, "call-outs".

INT 15h up-calls are generated by various modules within the BIOS. The floppy and hard disk modules are the most important ones, as they involve comparatively large intervals of time during head seeks and waiting for rotational latency of the media. During seeks and disk rotations, an operating system can use INT 15h to gain control and perform other tasks until notified that the operation has completed. Just as with the other INT 15h services, the BIOS

places a function code in the AH CPU register, with a device code in other registers, before executing its INT 15h instruction.

3.1.2.3.1.1 INT 15h Function 4fh

Function code 4fh is used to indicate that the keyboard has received a keypress or key release interrupt, with a scancode in the AL CPU register from the keyboard controller. The KEYBOARD module issues the INT 15h function to give the application a chance to interpret the scancode and modify it if required.

Upon return from the INT 15h function 4fh call, the keyboard BIOS checks the state of the carry flag. If the carry flag is cleared by the application, then the BIOS performs no more processing on the scan code and assumes that the application handled it. If the carry flag was set by the application, then the BIOS handles the scan code as returned by the application in the AL CPU register. The latter case allows the application to modify the scan code in the AL register without handling it directly. Because the BIOS sets the carry flag before issuing the INT 15h instruction, the BIOS will handle the scan code by default if the application does not modify the carry flag.

3.1.2.3.1.2 INT 15h Function 90h

Function code 90h is used to indicate that a spin-loop is about to be executed by a BIOS component. When the BIOS invokes INT 15h function 90h, it passes a device code in the AL CPU register that indicates what device is causing the wait. The following device codes are architected by IBM:

Code	Device Name
00h	IDE Hard Drive
01h	Floppy Disk Drive
02h	Keyboard
03h	PS/2 Mouse
80h	Network
FCh	Hard Disk Reset Operation
FDh	Floppy Disk Drive Motor Control Operation
FEh	Printer

Upon return, the application software that hooks the INT 15h function 90h service should set the CY flag if it did not wait for the device to complete its operation, or clear the CY flag if a wait or timeout occurred in the application code. The state of the CY flag is tested by the BIOS when the INT 15h function 90h routine returns to determine whether to actually perform or skip the spin-loop.

3.1.2.3.1.3 INT 15h Function 91h

Function code 91h is used to indicate that a device interrupt has just been received that would complete the spin-loop. As with function 90h, a device code is passed in the AL CPU register to indicate which device has just completed an operation. The device codes for functions 90h and 91h are identical.

Upon return, the application software that hooks the INT 15h function 91h service should set the AH CPU register to 00h and clear the CY flag.

3.1.2.3.1.4 INT 15h Function 85h

Another INT 15h up-call provides notification that the user has pressed or released the **SysReq** key on an AT-class (101-key) keyboard. When the KEYBOARD.ASM module detects that this key is pressed, it issues an INT 15h with AH=85h, and sets the AL CPU register to 00h, indicating that they key was depressed. When the key is released, it issues an INT 15h with AH=85h, and sets the AL CPU register to 01h.

3.1.2.3.2 INT 1bh Control-Break Signal

The KEYBOARD.ASM module executes an INT 1bh software interrupt if it detects that the user pressed the Control and Break keys simultaneously. This allows the application to gain control when this happens. Upon return from the INT 1bh instruction, the BIOS stores a 00h scan code and 00h character code in the keyboard's typeahead buffer.

DOS normally hooks the INT 1bh Interrupt Vector Table entry so that it can terminate a program prematurely. The mechanisms used by DOS to make this happen are proprietary to the specific version of DOS and are beyond the scope of this manual.

3.1.2.3.3 INT 1ch User Timer Interrupt

The BIOS provides a regular 18.2Hz heartbeat for operating systems and applications by executing an INT 1ch instruction every 55 milliseconds. By default, the BIOS has its own INT 1ch handler that does nothing, so that the application software is not required to provide a handler unless one is needed.

In strictly ISA systems, INT 1ch is executed inside the IRQ0 Interrupt Service Routine of the BIOS after the 8254 Programmable Interval Timer's T0 timer expires, and after the End-Of-Interrupt (EOI) has been issued to the primary Programmable Interrupt Controller (PIC). Thus, suspension inside the INT 1ch handler by the application does not degrade system performance the way it would be if the application suspended operations inside the INT 08h handler.

In non-ISA systems, such as those designed around the NEC V-Series (i.e., V25) processors, EMBEDDED BIOS cannot program the on-board timers to generate an interrupt on INT 08h. Instead, the timer is actually hard-wired to interrupt vector 1ch. The BIOS accounts for this and calls INT 08h inside the INT 1ch handler. Application software should be aware of this possibility and not block inside the INT 1ch handler. Instead, they should chain the INT 1ch handler, call the lower layer first, and then perform any work as required. This technique gives the BIOS a chance to issue an EOI before any application code runs.

3.1.2.3.4 INT 4ah Real Time Software Interrupt

Just as the ISA IRQ0 hardware timer interrupt routed to INT 08h causes the INT 1ch user timer software interrupt to be generated every 55ms, ISA IRQ8 is tied to a 1Khz timer routed to INT 70h, which in turn causes an INT 4ah instruction to be generated every 1ms (at a 1Khz rate).

Commonly called the real-time clock interrupt, INT 4ah can be "hooked" by real-time kernels to gain control for rescheduling purposes on ISA platforms. Warning: Nonstandard platforms may not provide this support, as it is provided by the Dallas Real-Time Clock (RTC) chip in PC/AT-compatible targets.

To enable the 1Khz INT 4ah interrupt heartbeat, the operating system or application must manipulate the CMOS RTC registers. The BIOS automatically routes the hardware interrupt (IRQ8) to interrupt vector 70h. The BIOS-supplied ISR for INT 70h then calls INT 4ah after issuing an EOI to both Programmable Interrupt Controllers (PICs).

3.1.2.4 CPU Traps/Faults

Intel 8086-family processors and their architectural equivalents all provide a way for the operating system or application program to gain control when an instruction cannot be executed for some reason. When the CPU encounters a problem with executing an instruction, it generates an exception.

When EMBEDDED BIOS is built with the option to enable the integrated BIOS debugger, the BIOS routes all the CPU-generated exceptions to the debugger itself, so that the adaptation engineer can determine why the exception occurred and then debug the problem. Without the debugger enabled, the operating system or application program is responsible for catching exceptions and handling them in an appropriate manner.

There are two types of exceptions; namely, traps and faults. Traps are generated when something happens that makes it impossible for the instruction to be restarted. When an invalid instruction is detected, for example, an "Invalid Instruction Trap" occurs.

Faults are different from traps in that a fault handler can perform some sort of work that would potentially allow the problem instruction to be able to re-execute correctly. A good example of such a fault is the "Page Fault" mechanism commonly used in virtual memory management systems in protected-mode operating systems. Because an instruction may execute from a page that is not present in memory, or perhaps because its operands in memory are located in pages of memory that are not present, the page fault mechanism gives the operating system control so that the necessary pages of virtual memory can be mapped to real physical memory. Once the mapping is completed, the fault routine returns to the interrupted context, and the instruction proceeds as though the fault never happened.

In Intel CPUs, the following exceptions can be generated by the CPU. Note that some are marked as traps, and some are faults. Also note that the interrupt vector numbers assigned to the exceptions conflict with the BIOS service interrupt numbers. This is not a misprint; it is an historical part of the BIOS architecture first defined by IBM.

Vector	Processors	Exception Type
00h	8086	Divide Error Trap
01h	8086	Instruction Trace Trap
02h	8086	NMI Interrupt (Trap)
03h	8086	Breakpoint Trap
04h	8086	Arithmetic Overflow Trap (INTO Instructions)
05h	80286	Array Bounds Trap
06h	8086	Invalid Opcode Trap

07h	8086	Device Not Available Trap
08h	80286	Double Fault
09h	N/A	-- Reserved for Future Use --
0ah	80286	Invalid Task State Segment Fault
0bh	80286	Segment Not Present Fault
0ch	80286	Stack Exception Fault
0dh	80286	General Protection Fault
0eh	80386	Page Fault
0fh	N/A	-- Reserved for Future Use --
10h	80386	Floating Point Fault
11h	80486	Alignment Fault
12h	80486	Machine Check Fault

In NEC V-Series CPUs, the following exceptions can be generated by the CPU.

Vector	Processors	Exception Type
00h	V20	Divide Error Trap
01h	V20	Instruction Trace Trap
02h	V20	NMI Interrupt (Trap)
03h	V20	Breakpoint Trap
04h	V20	BRKV Instruction
05h	V20	CHKIND Instruction

3.1.2.5 Hardware Interrupts

EMBEDDED BIOS is configurable to support a wide variety of processors that provide at least the functionality of the Intel 8088 CPU. Processors in the Intel 8086 family include the 80286, the 80386, the i486, Pentium, and Pentium-Pro CPUs, and these CPUs are generally deployed in ISA, PCI, or local bus-type system architectures.

Intel's 80C186-EA/EB/EC family of processors provide a superset of the instruction set, but in addition have on-board peripherals that are not like those in an ISA-class machine. Instead, the on-board timers, serial ports, and so on, are internally wired to different interrupt request lines, which in turn translate to different interrupt vectors that must be serviced by the BIOS.

Similarly, the NEC V-Series processors execute supersets of the Intel 8086 CPU's instruction set; however, their on-board peripherals are also proprietary and are internally-wired to different interrupt request lines. These different IRQs are necessarily routed through different interrupt vectors.

ISA-class systems all have a similar interrupt model for hardware interrupts. The ISA interrupt assignments are as follows:

IRQ	Vector	Device
IRQ0	08h	8254 Programmable Interval Timer
IRQ1	09h	Keyboard Controller
IRQ2	0ah	Cascade Interrupt to PIC2
IRQ3	0bh	COM2 Serial Port (8250)
IRQ4	0ch	COM1 Serial Port (8250)

IRQ5	0dh	LPT2 Parallel Port
IRQ6	0eh	Floppy Disk Controller
IRQ7	0fh	LPT1 Parallel Port
IRQ8	70h	Real-Time Clock Interrupt (1Khz)
IRQ9	71h	-- open --
IRQ10	72h	-- open --
IRQ11	73h	-- open --
IRQ12	74h	PS/2 Mouse
IRQ13	75h	Math Coprocessor
IRQ14	76h	Primary IDE Controller
IRQ15	77h	Secondary IDE Controller

3.3 Setup Screens

The EMBEDDED BIOS SETUP screen system is a comprehensive in-system configuration utility that can be invoked by the user during a power-on reset. SETUP provides a menu-driven, multi-screen interface that allows the user to quickly navigate through options at all levels of complexity. SETUP can even be configured by the adaptation engineer to redirect its keyboard and screen I/O over a serial port.

SETUP consists of ten components. The first component is the SETUP module itself, which implements the main menu and dispatches to the other components, as shown below.

Setup Module	Menu Option
SETUP	Main Menu
SETUPBAS	"Basic CMOS Configuration"
SETUPCST	"Custom Configuration"
SETUPDEM	"General Software Demonstration Screen"
SETUPDIA	"Standard Diagnostic Routines"
SETUPPMT	"Power Management Timer Configuration"
SETUPPMF	"Power Management Device Configuration"
SETUPPWD	"Password Configuration"
SETUPSHA	"Shadow Configuration"
SETUP	"Start System Debugger"
SETUP	"Enter Manufacturing Mode"
SETUP	"Format RAM Disk"
SETUP	"Format Flash Disk"
SETUP	"Reset CMOS To Last Known Values"
SETUP	"Reset CMOS To Factory Defaults"
SETUP	"Write To CMOS And Exit"
SETUP	"Exit Without Saving Changes"
DIAG	Diagnostics SETUP Screen, Part 1
DIAG2	Diagnostics SETUP Screen, Part 2

3.4 API Service Modules

The implementation of EMBEDDED BIOS is highly modular. Service modules, such as those that receive control when application software executes an INT 16h service request, are separated from the modules that actually manipulate hardware wherever possible. This enables the BIOS

to be expanded to support new devices without affecting its architectural integrity. The following is a list of EMBEDDED BIOS service modules.

Module Name	Function
CONIO	Keyboard and Video I/O (INT 16h, INT 10h)
DISKIO	All Disk I/O (INT 13h)
MISC	Information Services (INT 11h, 12h, 15h)
PARALLEL	Parallel I/O (INT 17h)
SERIAL	Serial I/O (INT 14h)
TIME	Date/Time (INT 1ah)
PCIAPI	PCI BIOS Function Router and Handlers

3.5 Device Service Modules

The modules that actually interact with devices are shown below. Overlap with the Service Modules list above occurs because sometimes a module both services a software interrupt and also interacts with the hardware.

Module Name	Function
72421	72421 Real Time Clock Device Driver
8042	8042-Compatible Keyboard Controller Device Driver
8237	8237A-Compatible DMA Controller Device Driver
8250	8250-Compatible UART Device Driver
8254	8254-Compatible Counter Timer Device Driver
8255	8255 PC/XT PIO Device Driver
8259	8259-Compatible Interrupt Controller Device Driver
HD61830	Hitachi 61830 LCD Driver
CUSTKBD	Custom Keyboard Module (for OEM use)
CUSTVID	Custom Video Module (for OEM use)
FLOPPY1	Floppy Device Driver, Part 1
FLOPPY2	Floppy Device Driver, Part 2
FLOPPY3	Floppy Device Driver, Part 3
IDE1	IDE/Hard Disk Driver, Part 1
IDE2	IDE/Hard Disk Driver, Part 2
KEYBOARD	PC, PC/XT, PC/AT Keyboard Driver
MTDAMD16	16-Bit AMD Flash Media Technology Driver
MTDAMD81	8-Bit 1-Way Interleave AMD Flash (with background erase) MTD
MTDAM81S	8-Bit 1-Way Interleave AMD Flash (no background erase) MTD
MTDAMD82	8-Bit 2-Way Interleave AMD Flash Media Technology Driver
MTDAMD84	8-Bit 4-Way Interleave AMD Flash Media Technology Driver
MTDATM81	8-Bit 1-Way Interleave Atmel Flash Media Technology Driver
MTDBULK	8-Bit Bulk Flash Media Technology Driver
MTDINT16	16-Bit Intel Flash Media Technology Driver
MTDINT81	8-Bit Intel Flash Media Technology Driver
MTDINT82	8-Bit 2-Way Interleave Intel Flash Media Technology Driver
MTDINT8A	8-Bit 1-Way Boot Block Intel Flash Media Technology Driver
MTDINT8B	8-Bit 2-Way Intel Flash Media Technology Driver
MTDINTA	16-Bit Advanced Intel Flash Media Technology Driver
MTDINTA2	16-Bit 2-Way Advanced Intel Flash Media Technology Driver
MTDTNAND	Toshiba NAND Media Technology Driver
MTDRAM	RAM Media Technology Driver

MTDROM	ROM Media Technology Driver
PS2MOUSE	PS/2 Mouse Device Driver
RAMDISK	RAM Disk Driver
RFD1	Resident Flash Disk, Part 1
RFD2	Resident Flash Disk, Part 2
ROMDISK	ROM Disk Driver
VIDEO	6845 CRT Controller Device Driver
SERMSG	RS-232 Driver for Manufacturing Mode

3.6 Other Modules

Other modules that aren't service request handlers or device drivers are shown below. Some of these modules provide other types of functionality, such as the integrated BIOS debugger. Others serve as routers for directing requests to the appropriate device handler (i.e., cache control).

Module Name	Function
APM	Advanced Power Management Request Handler
BIOS	Data Structures & BCPA Patch Area
BOARD	Board Personality Module Default Routines
BOOTOS	Boot-Time Action Logic
CACHE	Cache Request Router
CFGBOX	Configuration Box Display
CHIPSET	Chipset Personality Module Default Routines
CPU	CPU Personality Module Default Routines
DEBUG	Debugger Main Loop and Command Dispatching
DEBUGASM	Debugger Disassembler
DEBUGCMD	Debugger Command Handlers
DEBUGISR	Debugger Interrupt Service Routines
DEBUGOBJ	Debugger Command Handlers
DEBUGTBL	Debugger Opcode Table
EMULATE	PCODE interpreter
HELPER	Helper Routines for BIOS
MEDIA	Media Control Layer
MEMTEST	Exhaustive Memory Tests
MFGPROT	Manufacturing Mode Protocol Engine
PCI	PCI Bus Support
POST	Power-On Self-Test Main Routine
POST1	POST Routines
POST2	POST Routines
POST3	POST Routines
POST4	POST Routines
POST5	POST Routines
POWER	Power Management Engine
PRINTF	Output Formatting Package
PROTMODE	Protected Mode Switching, A20 Gating, Data Copying
ROMSCAN	ROM Extension Scan

3.7 CPU Personality Modules

EMBEDDED BIOS derives part of its configurability from the way its architecture permits routing of I/O requests to peripheral device managers or CPU Personality Modules (CPMs), so that on-board devices can be supported. In effect, all CPU-specific code in EMBEDDED BIOS is confined to one module called CPU.

Actually, the CPU module is a shell for convenience. Based on the CPUCLASS parameter in the project file, it includes the right .ASM files from a subdirectory of the CPUS directory that provide support for a particular CPU class.

The EMBEDDED BIOS Adaptation Kit includes standard support for NOCPU CPU class, which includes the generic forms of the following processors: 8088, 8086, 80286, 80386, i486, Pentium, and Pentium Pro. Exotic forms of these processors, such as the 80486SLC, or high integration processors such as the 80C186-EC and 386-EX, are supported by other class modules not provided with the standard Adaptation Kit. Typically, however, the standard NOCPU CPU class module can be adapted to support similar CPUs without too much effort.

EMBEDDED BIOS CPU Personality Modules all have the same architecture, regardless of the CPU class being supported. They all implement functions that are called by other components of the BIOS. A summary of the CPM functions is given below. The complete CPM specification is presented in Chapter 18.

CPM Function	Purpose
CpuInit0	Early CPU initialization
CpuInit1	Normal CPU initialization
CpuInitRefresh	Test and initialize CPU DRAM refresh controller
CpuEnableApm	Enable APM Support in CPU Module
CpuGetProcessorType	Return CPU type ordinal
CpuGetProcessorName	Return pointer to ASCIIZ CPU name
CpuHookVectors	Allow CPU Personality Module to route interrupts
CpuInitDma	Test and initialize CPU DMA controller
CpuEnableDmaCtrl	Enable CPU DMA controller
CpuDisableDmaCtrl	Disable CPU DMA controller
CpuStartDma	Start CPU DMA process
CpuFloppyDma	Start CPU DMA process for floppy channel
CpuInitIntCtrl	Test and initialize CPU interrupt controller
CpuEnableIntCtrl	Enable interrupt controller
CpuDisableIntCtrl	Disable interrupt controller
CpuUnmaskInt	Enable interrupt level at CPU interrupt controller
CpuEoi	Perform EOI on CPU interrupt controller
CpuInitTimer	Test and initialize CPU timer controller
CpuBeep	Beep speaker using the CPU timer controller
CpuInitWatchdog	Test and initialize CPU watchdog timer
CpuEnableWatchdog	Start watchdog timer
CpuDisableWatchdog	Stop watchdog timer
CpuKickWatchdog	Kick watchdog timer
CpuEnableCache	Enable CPU L1 cache
CpuDisableCache	Disable CPU L1 cache
CpuSetFastSpeed	Set CPU speed to high
CpuSetSlowSpeed	Set CPU speed to low
CpuEnableA20	Enable A20 line in CPU-specific manner
CpuDisableA20	Disable A20 line in CPU-specific manner
CpuTestSyncIo	Test CPU synchronous I/O controller

CpuInitSerial	Test and initialize CPU serial ports
CpuInitParallel	Test and initialize CPU parallel ports
CpuInitSerBios	Initialize serial I/O subsystem in CPU
CpuSerPutCh	Write byte to CPU serial port
CpuSerGetCh	Read byte from CPU serial port
CpuSerGetStatus	Read status of CPU serial port
CpuSerInit	Handle INT 14h fn 00h for CPU serial port
CpuSerInitExt	Handle INT 14h fn 04h for CPU serial port
CpuExtRwCtrl	Handle INT 14h fn 45h for CPU serial port

3.8 Chipset Personality Modules

EMBEDDED BIOS can also be configured to support high-integration chipsets used in high-density motherboard designs, and chipsets that are actually packaged with embedded processors such as the AMD Elan series. Chipsets are feature-rich and are all programmed differently, because there is no one hardware interface standard that all the chipset vendors would be able to agree to use. Main functional areas handled by Chipset Personality Modules are L2/L3 Cache Control, DRAM Configuration and Initialization, PCI Interrupt Routing, and ROM Shadowing.

Chipsets are generally responsible for managing the Single and Dual Inline Memory Modules (SIMMs, DIMMs, and SODIMMs), interleaving them properly to achieve high memory bandwidth. Chipsets also control the wait states used to access memory and I/O devices, support shadowing, DRAM refresh, and external bus clocking. The chipset programming here is what makes it possible for the BIOS to support FP, EDO, SDRAM, Registered SDRAM, RDRAM, and other memory types.

Memory and bus management is just one function that generally falls into a chipset's feature domain. Another is the emulation of standard ISA hardware: an 8042 keyboard controller, two 8237A DMA controllers, an 8254 programmable interval timer, two 8259 programmable interrupt controllers, and a real-time clock with integrated CMOS. Some chipsets add floppy disk controllers, serial, and parallel ports.

Specialty functions provided by some chipsets include CPU and ISA bus clock control, external cache management, turbo mode control, A20 line gating, and the PS/2-compatible I/O port 92h.

Because EMBEDDED BIOS components, such as the CACHE module, have a need to control the functionality of the chipset, General Software has defined a standard software interface for the core system BIOS to a Chipset Personality Module (CSPM). The core BIOS contains default routines that perform these functions in a standard way for systems without chipsets. These routines can be overridden by replacements in a CSPM. General Software makes available support modules containing one or more CSPMs each, for many chipsets.

The following is a summary of the functions provided by the CSPM that are called by the core system BIOS. A detailed specification for CSPMs is presented in Chapter 19.

CSPM Function	Purpose
CsInit0	Early chipset initialization
CsInit1	Normal chipset initialization
CsMemConfig	Autodetect DRAM geometry and size
CsInitRefresh	Test and initialize chipset DRAM refresh controller
CsEnableApm	Enable APM Support in Chipset Module

CsDisplayChipset	Display chipset module identification
CsShadowArea	Enable shadowing for region of address space
CsDisableShadow	Disable all shadowing
CsShadowWriteCtl	Enable/disable writes to shadow memory
CsInitWatchdog	Test and initialize chipset watchdog timer
CsEnableWatchdog	Start chipset watchdog timer
CsDisableWatchdog	Stop chipset watchdog timer
CsKickWatchdog	Kick chipset watchdog timer
CsEnableCache	Enable external (L2) chipset-controlled cache
CsDisableCache	Disable external (L2) chipset-controlled cache
CsSetFastSpeed	Set CPU speed to fast with chipset control
CsSetSlowSpeed	Set CPU speed to slow with chipset control
CsEnableA20	Enable A20 with chipset control
CsDisableA20	Disable A20 with chipset control
CsReboot	Reboot CPU with chipset control
CsGetPciIrq	Return assigned PCI interrupt line for IRQ level
CsMapAddress	Map 32-bit media address using chipset MMU
CsUnMapAddress	Restore mapping registers in chipset MMU
CsReadReg	Read chipset register for debugger
CsWriteReg	Write chipset register for debugger
CsEnableApm	Enable APM support in chipset
CsTimerTick	Receive control on each timer tick for convenience

3.9 Board Personality Modules

The EMBEDDED BIOS architecture supports a third class of personality modules, related to how the chipset, CPU, and any other components in the target system are interconnected. This module, called the Board Personality Module (BPM), receives requests from the core BIOS to perform functions that would ordinarily be passed-on to the CPM and/or the CSPM. However, circumstances may dictate that the predefined routines in the CPM or CSPM not be called, or be called in a different order, or that additional instructions be placed around the calls to the underlying CPM or CSPM routines. This is the job of the BPM.

As with the CPM and CSPM, default routines inside the core BIOS perform standard routing functions, and these default routines are overridden by a specified BPM module. Based on the BOARD parameter in the project file, it includes the right .ASM files from a subdirectory of the BOARDS directory that provide support for a particular board design.

The EMBEDDED BIOS Adaptation Kit includes standard support for NOBOARD design, which performs pass-through routing of requests to chipset and CPU modules where appropriate, and performs default no-operation functions for non-CPU or chipset functions.

EMBEDDED BIOS Board Personality Modules all have the same architecture, regardless of the board being supported. They all implement functions that are called by other components of the BIOS. A summary of the BPM functions is given below. The complete BPM specification is presented in Chapter 20.

BPM Function	Purpose
BoardInit0	Early board initialization
BoardInit1	Normal board initialization
BoardInit4	Board initialization before keyboard and video init

BoardInit6	Board initialization after keyboard and video init
BoardInit8	Chipset/board initialization from CMOS cells
BoardInitFields	Load fields from CMOS into RAM
BoardSaveFields	Store fields from RAM into CMOS
BoardResetCmos	Reset fields in CMOS to predefined conditions
BoardSaveCmos	Save CMOS cells to nonstandard hardware (ie, Flash)
BoardInitAppRom	Initialize board for application ROM access
BoardDelayUsec	Perform delay using board-specific mechanism
BoardInitRefresh	Test and initialize DRAM refresh
BoardMemConfig	Determine size and geometry of DRAMs
BoardShadowArea	Shadow an area of address space <1MB
BoardDisableShadow	Disable all shadowing
BoardInitDma	Test and initialize DMA hardware
BoardEnableDmaCtrl	Enable DMA controller
BoardDisableDmaCtrl	Disable DMA controller
BoardFloppyDma	Start DMA operation
BoardInitIntCtrl	Test and initialize interrupt controller
BoardEnableIntCtrl	Enable interrupt controller
BoardDisableIntCtrl	Disable interrupt controller
BoardUnmaskInt	Enable specific IRQ level
BoardEoi	Perform EOI on interrupt controller
BoardInitTimer	Test and initialize timer controller
BoardBeep	Beep speaker
BoardInitWatchdog	Test and initialize watchdog timer
BoardEnableWatchdog	Start watchdog timer
BoardDisableWatchdog	Stop watchdog timer
BoardKickWatchdog	Kick watchdog timer
BoardEnableCache	Enable external (L2) cache
BoardDisableCache	Disable external (L2) cache
BoardSetFastSpeed	Set CPU speed to high
BoardSetSlowSpeed	Set CPU speed to low
BoardEnableA20	Enable A20 line
BoardDisableA20	Disable A20 line
BoardReboot	Reboot system
BoardPostError	Handle critical error during POST
BoardEnableApm	Enable APM functions
BoardApmMode	Set power management mode
BoardPwrLvl	Set power level for the board itself
BoardTestMode	Determine if Manufacturing Mode should be entered
BoardDisableTestMode	Disable Manufacturing Mode entry flag
BoardEnableWrites	Enable Vpp to Flash devices
BoardDisableWrites	Disable Vpp to Flash devices
BoardSetVideoMode	Adjust video BIOS parameters (rows, cols, etc.)
BoardTimerTick	Receive control from INT 08h timer tick
BoardPciControl	Manage PCI policy
BoardEnablePciRegion	Enable access to PCI region
BoardHelp1	Unarchitected board helper for CSPM if required
BoardHelp2	Unarchitected board helper for CSPM if required
BoardMapAddress	Map 32-bit address using board/chipset/CPU hardware
BoardUnMapAddress	Restore prior mapping using board/chipset CPU hardware
BoardFsInit	Perform board-specific file-system initialization
BoardIdeAutoDetect	Examine IDE parameters during IDE driver initialization
BoardSioReadReg	Read Super I/O register for debugger
BoardSioWriteReg	Write Super I/O register for debugger

BoardPostCodeComInit	Initialize POSTCODECOM output device during POST
BoardPostCodeCom	Output message to POSTCODECOM device for POST

3.10 BIOS Configuration

EMBEDDED BIOS is configurable in many ways. Of course, the EMBEDDED BIOS Adaptation Kit includes full source, so you could in theory modify the source code to fit your hardware.

In early BIOS implementations, this was in fact how adaptations of desktop PC BIOSes were performed. The task took months and costs were high. Modularity started becoming important because of the critical time to market factor.

Some vendors announced BIOSes that were excessively modular and abstract, in a 180-degree turnaround to respond to the growing number of desktop PC manufacturers who needed a custom BIOS. For example, experimental BIOSes were implemented in C at the expense of overall system performance, excessive stack depth, and general control over the hardware. These BIOSes were fine learning aids, but they don't solve real-world engineering problems.

EMBEDDED BIOS strikes a middle ground. It comes with source code so that the adaptation engineer can make changes to it if absolutely necessary. In nearly all cases, however, it just isn't necessary. Two include files, called `INC\OPTIONS.INC` and `INC\CONFIG.INC`, are included by every core BIOS source code module. These two files contain the default values for roughly 400 symbol definitions that tell the rest of the BIOS code how to be assembled through conditional and parametric assembly. After these files are included, a third file called the project file is included. The project file, maintained by the OEM, specifies the board, chipset, and CPU personality modules to be used in a build. Further, the project file contains new definitions that override the parameters in `INC\OPTIONS.INC` and `INC\CONFIG.INC` as necessary for the customized build of the BIOS.

Thus, a change to the project file causes the system-wide changes necessary to start supporting, or omit support for, some function. Configuring the BIOS by editing a project file is totally automated with the Windows-based BIOStart utility, or can take only a short amount of time in a text editor with a reassembly of the BIOS with GSMAKE in the DOS command-line environment.

Once a binary version of a BIOS is create, it can still be further configured BIOStart by a binary patching process. See Chapter 6 for more details about binary configuration.

Finally, EMBEDDED BIOS can be configured by the end user during POST with the comprehensive Setup system. This set of full-screen menus allows the user to edit CMOS values used by the Chipset Personality Module (CSPM), the device control modules (floppy disk, hard disk, and so on), and other modules, such as the cache control system.

3.10.1 Project Files

The adaptation engineer can configure EMBEDDED BIOS features and underlying mechanisms by creating and editing a project file associated with a BIOS project. Projects are given a name by the OEM, and are then folded into the BIOS build environment by creating a subdirectory of the project's name underneath the `PROJECTS` directory, and then creating a text file by the same name with an .INC extension inside the new subdirectory. This file is called the project file.

Three configuration parameters are required in any project file: **BOARD**, **CPUCLASS**, and **CHIPSET**. These three parameters name the Board, CPU, and Chipset Personality Modules, respectively, that should be used to build a BIOS for the project. The rest of the project file contains zero, one, or more overrides for symbol definitions found in the `INC\OPTIONS.INC` and `INC\CONFIG.INC` files.

This file and directory structure can be created and edited with BIOStart, the Windows-based BIOS configuration program, or this can all be done by hand with DOS file management commands and a text editor.

Before making configurations, the adaptation engineer should make a plan, on paper, that describes the hardware to be supported, the features to be provided, and how the features will be supported by the given hardware.

Each option in the `INC\OPTIONS.INC` and `INC\CONFIG.INC` default symbol definition files is represented by a symbol equate. Most options in the `INC\OPTIONS.INC` file are set to 1 to indicate that the associated support should be enabled, and to 0 if support should not be provided. Note that not all configuration possibilities are necessarily valid ones. For example, enabling the second 8237A option but not enabling the primary 8237A would be nonsensical, since it takes one of an object to have an additional one. Most options in the `INC\CONFIG.INC` file are qualifiers for options set elsewhere; i.e., the starting physical address of the Flash disk array. Of course, if the associated feature were not enabled, then the value may not have meaning to the BIOS build.

3.10.2 Binary Configuration Patch Area

Most of the configuration parameters in the `INC\CONFIG.INC` header file are simply assembled into an area called the Binary Configuration Patch Area (BCPA). Then, other BIOS modules needing access to these values reference the fields in the BCPA instead of the assembly symbols in the `INC\CONFIG.INC` include file.

By dereferencing these values through the BCPA table, the BIOStart program can be used to patch the BCPA stored in a binary image of the BIOS, after it is assembled. The BCPA table is prefixed with a special signature that makes it easy for the BCP program to locate.

3.10.3 System Configuration Table

BIOS service INT 15h function C0h returns a pointer to a data structure called the System Configuration Table (SCT), an area inspected by DOS and applications to determine which features are supported by the underlying BIOS. The SCT is defined in module BIOS and may be modified by the adaptation engineer.

The contents of the standard SCT are given below:

```
        PUBLIC  SCT
SCT:
        dw      SCT_End - SCT - 2
        db      BIOS_HDWR               ; hardware ID byte.
        db      BIOS_MAJOR_VERSION
        db      BIOS_MINOR_VERSION

;       The next byte contains bitflags as follows, indicating what
```

```
;       features the BIOS supports.
;
;       bit 7 - BIOS using DMA ch3
;       bit 6 - cascaded IRQ2
;       bit 5 - real-time clock present
;       bit 4 - int 1Ah is keyboard scan

SCTFLAGS =      00000000B               ; start out with nothing.

        IF      OPTION_SUPPORT_8259_2
SCTFLAGS =      SCTFLAGS OR 01000000B   ; bit 6 - cascaded IRQ2.
        ENDIF   ; (OPTION_SUPPORT_8259_2

        IF      OPTION_SUPPORT_CMOS
SCTFLAGS =      SCTFLAGS OR 00100000B   ; bit 5 - real-time clock present.
        ENDIF   ; (OPTION_SUPPORT_8259_2)

SCTFLAGS =      SCTFLAGS OR 00010000B   ; bit 4 - INT 1Ah is keyboard scan.

        db      SCTFLAGS                ; define flag byte with above bitflags.

        db      4 dup(0)                ; reserved
SCT_End EQU     $
```

3.10.4 Keyboard Scancode Translation Table

The KEYBOARD module uses a lookup table in module BIOS to translate scancodes read from the PC/XT-style 8255 or the PC/AT-style 8042 keyboard controller, into ASCII characters. This table also defines how the keyboard shift keys, such as CTRL, SHIFT, CAPS_LOCK, SCROLL_LOCK, and NUM_LOCK, affect other keys when they are pressed or released.

The keyboard translation table can be modified by the adaptation engineer to handle special keyboards that do not conform to the IBM PC or PC/AT standards. If this is desired, some modifications to routine **Int09Isr** in module `KEYBOARD.ASM` will be necessary to handle communication with the alternate controller. Alternatively, the OEM may insert custom keyboard handling code in the `CUSTKBD.ASM` module, and that code in the project file.

3.11 Console I/O Redirection

Both INT 10h (video) and INT 16h (keyboard) services may be redirected by EMBEDDED BIOS to any serial port, so that the application, Setup screen, and integrated BIOS debugger can all communicate over an RS-232 link to a remote host running a terminal program. These three classes of I/O can be redirected independently, to any valid serial port in the system supported by the `SERIAL.ASM` module.

3.11.1 Video (INT 10h) Redirection

Redirection of INT 10h (video) requests happens in CONIO by looking at the **CurrIo** field in the Extended BIOS Data Area (EBDA). If this field is set to **IO_CONSOLE** (0), then video is routed to the VIDEO module, which programs the 6845 CRT controller. If this field is set to

IO_COM1 (1), **IO_COM2** (2), **IO_COM3** (3), or **IO_COM4** (4), then the I/O is redirected to the specified port. **IO_NONE** is a value the system uses to disable INT 10h output altogether.

Other fields in the EBDA take part in redirection. Consider the fact that application INT 10h services are separated from debugger INT 10h services and Setup screen INT 10h services. This is handled by the Setup and Debugger modules by setting the **CurrIo** field to the values in **SetupIo** or **DebugIo**, respectively, before those modules do any output. Then, when the modules are finished with their processing, they restore the **CurrIo** field to its former value. The save areas for this restoration are **SetupIox** and **DebugIox**, respectively. An INT 15h function is available for handling these details; it is callable from the application program, or from the Board, CPU, or Chipset Personality Modules if a stack is available.

3.11.2 Keyboard (INT 16h) Redirection

Redirection of INT 16h requests to the SERIAL.ASM module happens in module CONIO.ASM by looking at the **CurrIo** field in the Extended BIOS Data Area (EBDA). If this field is set to **IO_CONSOLE** (0), then keyboard requests are passed to module KEYBOARD.ASM, which programs the 8042 keyboard controller on an AT, or the 8255 peripheral interface on a PC/XT compatible machine. If this field is set to **IO_COM1** (1), **IO_COM2** (2), **IO_COM3** (3), or **IO_COM4** (4), then the I/O is redirected to the specified port. **IO_NONE** is a value the system uses to disable INT 16h output altogether.

Just as with INT 10h services, other fields in the EBDA take part in input redirection. Because application INT 16h services are separated from debugger INT 16h services and Setup screen INT 16h services, the Setup and Debugger modules set the **CurrIo** field to the values in **SetupIo** or **DebugIo**, respectively, before those modules request any input. Then, when the modules are finished with their processing, they restore the **CurrIo** field to its former value. The save areas for this restoration are **SetupIox** and **DebugIox**, respectively.

Module SERIAL.ASM doesn't actually do the I/O directly for the I/O redirection. Instead, it translates the logical serial port number associated with the console redirection into a physical port number, and then calls the 8250 driver module, or the CPU personality module, depending on where the physical UART is located.

3.12 Integrated BIOS Debugger

When bringing-up new hardware, it is essential to have a debugging tool that can disassemble code, display and alter the contents of memory, write to I/O ports, breakpoint code, and test the operation of the A20 line and CMOS storage. These functions are all features of the integrated BIOS debugger that is provided with EMBEDDED BIOS.

By enabling the **OPTION_SUPPORT_DEBUGGER** configuration option in CONFIG.INC, the debugger code will be automatically assembled into the BIOS. Then, when the system boots, the debugger can be started in several ways.

First, on machines with a PC or PC/AT-compatible keyboard, the debugger can be entered through a special key chord. Just depress both the left ALT key and the left SHIFT key to break into the debugger.

Second, the debugger hooks the CPU exception vectors in case a divide by zero occurs, an invalid opcode is executed, or an INT 3 instruction is executed, for example. By placing an INT

3 in the POST mainline code (or anywhere else in the BIOS source code) after INT 10h and INT 16h services are available, the debugger will automatically be invoked. To resume, type the 'G' command to "GO", or continue on with the rest of initialization.

Third, the debugger can be entered from the Setup main menu, if the debugger Setup screen option is enabled. This allows an end-user to access the integrated BIOS debugger from within the full-screen menuing system.

Fourth, the debugger is a selectable boot action, allowing it to gain control if any of the other bootable drives are not available or are not formatted. This is controlled via the Basic SETUP screen.

Finally, the debugger can be entered if no operating system can be loaded. The system displays a message that indicates that a boot device cannot be found, and then prompts the user to press the ESC key to enter the debugger.

The debugger can be used over a serial port, in the event that the target system has no keyboard or monitor, or if those devices are being used by the application. For example, if a graphics application has drawn on the screen, the integrated BIOS debugger's output would disrupt the video display if it were not redirected. Redirection of debugger output is controlled via the **OPTION_CONIO_DEBUG** configuration option in the project file.

Use of the integrated BIOS debugger is outside the scope of this section; consult Chapter 9 for complete details.

3.13 Manufacturing Mode

EMBEDDED BIOS provides a powerful mode of operation that allows a host PC to control a target PC running EMBEDDED BIOS over an RS-232 connection at high speeds. The target can enter Manufacturing Mode from a SETUP screen, by testing OEM-defined hardware, or at the OEM's option, when critical POST errors occur. Manufacturing Mode is also available as a boot action, in the case that no actual bootable drives are available.

Manufacturing Mode is an alternative boot mode, similar to booting the operating system from a drive. Once started, Manufacturing Mode waits for incoming requests over an RS-232 port selected by the OEM. The target can enter MM permanently, or can attempt to establish a connection for up to a few seconds before continuing to boot the operating system.

When the target is in MM, the host computer can run two types of software. The first type is a device driver (`MFGDRV.SYS`) provided in the `UTIL` subdirectory of the BIOS software. This device driver provides an additional drive letter on the host machine that maps to a specified target drive, allowing the host computer's operator to remotely format and copy files to/from the device with standard DOS commands and utilities. It should be remembered that this link, including drive-level access, is provided by MM without requiring any operating system on the target side.

Another type of software that may be run on the host to communicate with a target in MM is a program that calls the MM API functions (described in Chapter 14). An example program is provided in the `UTIL` subdirectory of the BIOS software (`HOST.EXE`). This program illustrates how a C or C++ program can be written to call the MM API functions to perform functions such as Flash updating, memory testing, and booting the operating system.

To make the RS-232 connections between your target and host development machine for MM, you will need a properly-wired *null modem*; that is, with the transmit and receive lines twisted, as well as the handshaking lines twisted. Here are the pinouts for cables built with 9-pin connectors and 25-pin connectors:

Signal	9-Pin	25-Pin	25-Pin	9-Pin
Ground-Ground	pin 5	pin 7	pin 7	pin 5
Xmit-Recv	pin 3	pin 2	pin 3	pin 2
RTS-CTS	pin 7	pin 4	pin 5	pin 8
DSR-DTR	pin 6	pin 20	pin 20	pin 4
Recv-Xmit	pin 2	pin 3	pin 2	pin 3
CTS-RTS	pin 8	pin 5	pin 4	pin 7
DTR-DSR	pin 4	pin 20	pin 6	pin 6

3.14 ROM Disk

EMBEDDED BIOS also provides a solid state version of a mechanical floppy disk drive, called a ROM disk. Unlike most ROM disks, which operate as ROM Extensions and take up valuable address space, the EMBEDDED BIOS ROM disk code is integrated with the core system BIOS, and is configurable via the Setup screen system. The ROM disk driver can emulate one or several soft and/or hard drives within a system.

The ROM disk works by intercepting INT 13h requests made by an operating system or application program, and translating them into corresponding memory moves from a ROM area to the application-supplied data buffer. The EMBEDDED BIOS ROM disk takes advantage MMU support provided by the BIOS chipset or CPU personality modules to window ROM memory as needed, all transparently to the application.

ROM disk data transfers originate from a 32-bit media address supplied by the adaptation engineer at the time the BIOS is configured with the FILE_SYSTEM macro in the project file. This address can be a physical address space, or it can be a logical one associated with PC Cards or chip select lines supported by a high integration CPU or chipset.

The data stored at this address is simply a linear array of 512-byte sectors retrieved from any IBM PC-compatible floppy disk. Supplied with your Adaptation Kit is a utility called DISKIMAG.EXE (in the TOOLS directory) that can copy the contents of any DOS-formatted floppy disk or hard drive partition to a file, so that it can be burned into ROM as an image of the disk suitable for use by the ROM disk.

For complete details about the operation of the ROM disk component of EMBEDDED BIOS, consult Chapter 12.

3.15 Watchdog Timer

EMBEDDED BIOS provides a watchdog timer feature that allows operating systems and applications to instruct the hardware to automatically reset in the event that the hardware does not receive regular "check-ins" from the operating system or application.

Watchdog timer hardware comes in several different forms. Some high-integration CPUs, such as the 80C186-EC or 80386-EX processors from Intel, contain a watchdog timer that even starts out armed, so that the POST software is under watchdog control. In cases where the CPU does

not contain a watchdog timer device, an external count-down timer can be rigged-up to strobe the reset line on the CPU if desired. In some cases, this type of mechanism is already provided by high-integration chipsets.

To program the watchdog timer, the operating system or application makes calls to the general functions BIOS service interrupt, INT 15h, function C3h. A subfunction code is placed in the AL CPU register by the caller; 00h disables the watchdog timer and 01h enables it. Complete programming details can be found in Chapter 21.

3.16 Power Management and APM

EMBEDDED BIOS provides support for power-sensitive applications by being compatible with the Microsoft Advanced Power Management (APM) Specification. The BIOS power manager uses a sophisticated power management device tree to manage power-down and power-up sequences in an orderly fashion for cooperating devices within power groups.

If APM is not to be used in the system, the power manager can assign device inactivity timeouts to devices and automatically manage the system's power by transitioning devices through APM-compatible states.

If APM is enabled in the system with the **OPTION_SUPPORT_APM** configuration option, the operating system and application may access APM BIOS services that can be used to control the power-consumption state of the system. These requests are routed by module `POWER.ASM` to either the CPU Personality Module or the Chipset Personality Module to actually interact with the hardware, as power management support is being provided in most of today's chipsets and high-integration CPUs such as the Intel 80C186-EC.

APM services are requested by the operating system or application through INT 15h function 53h. The complete programming details surrounding the system BIOS power management API are presented in Chapter 15.

3.17 Cache Management

EMBEDDED BIOS provides comprehensive support of single-level and two-level caches in the target in multiple ways. First, CPUs such as Intel's i486 and Pentium processors have on-board caches that are disabled at processor reset and can be enabled for substantially improved performance. The on-board CPU cache is controlled through the manipulation of special CPU registers, and the cache status must be restored when the processor is reset during steady-state operation of the system.

Systems with external caches are also supported by EMBEDDED BIOS through the Board Personality Module (BPM) and Chipset Personality Module (CSPM). Although external cache is almost always controlled with an actual chipset, the BPM and CSPM paradigms work well even when an actual piece of VLSI is not present. External cache must be manually enabled and disabled at various points during the operation of the BIOS. For example, cache must be disabled during the POST memory test, whereas it must be enabled when applications are running.

Both internal and external caches are controlled by the `CACHE.ASM` module, and can be managed through the INT 16h function F4h BIOS service. For complete programming details, see Chapter 21.

3.18 Protected Mode Support

On 80386 and above processors, EMBEDDED BIOS uses the protected mode of the CPU to access extended memory. To configure EMBEDDED BIOS to support protected mode, the **OPTION_SUPPORT_PROTMODE** option must be enabled. Support for the 80286 CPU has been discontinued due to the CPU's architectural limitations and end-of-life.

When configured, protected mode is used during POST and during steady-state operation of the system. During POST, the BIOS determines the amount of extended memory available by performing memory tests in protected mode. During steady-state, the operating system or application program can request that the BIOS switch to protected mode with INT 15h function 89h, and move memory while in protected mode with INT 15h function 87h.

Protected mode support is complicated by the various ways in which the processor can be instructed to resume execution in real mode after a protected mode operation. In 80286-based systems, there was no "switch to real mode" CPU instruction. Therefore, these systems had an outboard hardware solution that involves any of several components: the 8042 keyboard controller, I/O port 92h, or the chipset. These methods are supported by EMBEDDED BIOS in the event that systems continue to use them even though they are not 80286-based.

On 80386 and above systems, the CPU contains a "`MOV CR0, EAX`" instruction that allows the BIOS to return to real mode without the use of external hardware. The adaptation engineer can select that this option be used when it is known that the target will be using an 80386 or better CPU.

It remains common for the outboard hardware reset techniques to be used in designs that use 80386 or better CPUs, simply because these features have been added to chipsets for full compatibility with the IBM PC/AT standard machine. Thus, the adaptation engineer should review the methods by which the target hardware can be switched back into real mode, and select the one with the lowest overhead.

Chapter 4

SETTING UP YOUR DEVELOPMENT TOOLS

This chapter will help you to set up your development environment to build the 16-bit EMBEDDED BIOS core and related programs, such as the Manufacturing Mode HOST utility program. It will also help you to build the 32-bit BIOS components, such as 32-bit PCI services and 32-bit BIOS directory services.

NOTE: This chapter documents all of the General Software build tools, even though the build process is automated with the GSMAKE utility or BIOStart for Windows. It is not necessary to learn how to use all of the tools until you have a need for them.

Be sure to read the next section on configuring for Borland or Microsoft tool sets, and setting environment variables. Then, we suggest that you begin learning about the build process in Chapter 5. Refer to this chapter as you need to learn about specific tools from time to time.

4.1 Configuring for Borland or Microsoft Tools

You may use either Borland or Microsoft development tools to build the EMBEDDED BIOS Adaptation Kit software, although you should be able to use any development tools to develop your applications that will be running on EMBEDDED BIOS.

There are three components of the BIOS software that need building in the source kit. The 16-bit core BIOS is the main part of the product, and is really the software that boots the target. This software is built using 16-bit assemblers and linkers (either TASM or MASM). The 16-bit utilities in the UTIL and COW directories are built with 16-bit assemblers, C compilers, and linkers (TASM/Borland C++ or MASM/MSVC1.52). Separate from these builds is a set of optional 32-bit BIOS components that enable certain BIOS features, such as 32-bit PCI services or 32-bit BIOS directory services. If these features are desired by the OEM, then this build must be performed with 32-bit tools (TASM32/TLINK32 or MASM and a special linker obtained from your Microsoft Developer Studio package that can support 32-bit linking).

4.1.1 Obtaining Borland 32-bit Tools

The 32-bit Borland tools are included with TASM. The assembler is named TASM32.EXE, and the linker is named TLINK32.EXE. These are automatically selected by the MAKEFILE used by GSMAKE when the 32-bit build is run. If you are using Borland tools, this should be automatically handled for you.

4.1.2 Obtaining Microsoft 32-bit Tools

For users of Microsoft toolsets, the issue of finding a 32-bit linker (not the incremental linker, but just the 32-bit segmented one) is important. The LINK.EXE that comes with MASM cannot recognize the 32-bit object records generated by the 32-bit BIOS build. You'll need to find a copy of Microsoft's 32-bit linker, and rename that copy "LINK32.EXE", and place it in your path (as a location, we suggest BIOS43\TOOLS). Then, you'll need a few DLLs that are required by that Microsoft linker. These files have been found on the Microsoft web site in the Windows 98 DDK.

If you have already installed VC++ 5.0, the necessary files are:

```
10/15/98  04:01p              359,424 LINK32.EXE
10/15/98  04:02p              117,520 MSDIS100.DLL
10/15/98  04:01p              167,424 MSPDB50.DLL
```

To get LINK32.EXE, you should find a file of the same length and date/time stamp as that shown above for LINK32.EXE, named LINK.EXE, and copy it to LINK32.EXE.

If you have not already installed VC++ 5.0, you should find all of the above files in the Windows 98 DDK, and in addition, you'll need to copy the following file from the C:\WINNT\SYSTEM32 to your path:

```
01/22/97  09:26p              565,760 MSVCP50.DLL
```

4.1.3 Build Control Environment Variables

Described later, the GSMAKE.EXE program is supplied with this Adaptation Kit to eliminate the dependency on your compiler vendor's (possibly incompatible) MAKE programs. For example, Microsoft's MAKE.EXE program shipped with its MSC V5.1 and earlier compilers does not correctly build a hierarchical dependency tree, although its NMAKE.EXE does. NMAKE.EXE, however, does not handle larger MAKEFILEs needed by this Adaptation Kit, and it supports different syntax than the Borland MAKE program does.

The General Software GSMAKE utility supports conditionals that allow it to test for the existence of symbols in your environment space. The most important variable you need to set (to anything, actually) in your environment space is the string, BORLAND. If this variable is defined, then all the MAKEFILEs in this Adaptation Kit will define macros that enable use of BCC, TASM, and TLINK. If this variable is not defined, then the builds will use Microsoft CL, MASM, and LINK.

IMPORTANT: To use Borland tools with EMBEDDED BIOS, insert the following in your development machine's AUTOEXEC.BAT file:

```
C> SET BORLAND=YES
```

To use Microsoft tools with EMBEDDED BIOS, insert the following in your development machine's AUTOEXEC.BAT file:

```
C> SET BORLAND=
```

Specifying BORLAND= without anything after the '=' causes the symbol to be undefined (removed) from your environment space.

If you are using MASM 6.1, 6.11, or 6.14 with EMBEDDED BIOS, insert the following in your development machine's AUTOEXEC.BAT file:

```
C> SET MASM61=YES
```

Another pair of important variables used by the MAKEFILE in the build process are NOPCI32 and NOGSMERGE. As with BORLAND=, these environment variables can be set to any text string value to enable the action, and to the empty string to remove them from the environment and thereby disable their action. In the case of these two variables, setting them to some value actually causes a feature in the build to be disabled; namely, 32-bit PCI build for NOPCI32, and the whole GSMERGE step for NOGSMERGE.

Disabling the 32-bit PCI services build:

If you will need to have 32-bit PCI services or 32-bit BIOS directory services as a part of the final output of the build, then you must not define NOPCI32. If you define NOPCI32, then these 32-bit components will not be built. For example, to <u>disable</u> the 32-bit PCI build:

```
C> SET NOPCI32=YES
```

To enable the 32-bit PCI build:

```
C> SET NOPCI32=
```

Specifying NOPCI32= without anything after the '=' causes the symbol to be undefined (removed) from your environment space.

Disabling the whole merge process:

The output of the core 16-bit BIOS build is an .ABS file. This file is by default merged with other ancilliary files, such as the 32-bit PCI services module (if NOPCI32 is not defined), and other files, such as VGA BIOS extensions, or external SMI code. To disable this merge process entirely, you can set the NOGSMERGE environment variable to some value (YES is recommended). If the environment variable is removed from the environment space, then the merge is run. To <u>disable</u> the merge process at the end of the build:

```
C> SET NOGSMERGE=YES
```

To enable the merge process:

```
C> SET NOGSMERGE=
```

Specifying NOGSMERGE= without anything after the '=' causes the symbol to be undefined (removed) from your environment space.

To see how the vendor selection mechanism works, review `EBIOS43\PROJECTS\MAKEFILE` and observe the `.IFDEF`, `.ELSE`, and `.ENDIF` constructs.

4.2 Standard System/Toolset Environment Variables

If you are using Microsoft tools, you will need to set the following environment variables to point to their associated directories. Borland's tools read .CFG files located in the directory from where they are loaded.

```
PATH=EBIOS43\TOOLS;...

TEMP=<scratch directory>

LIB=C800\LIB

INCLUDE=C800\INCLUDE

INIT=C800\INIT
```

Edit your `PATH` statement in `CONFIG.SYS` or `AUTOEXEC.BAT` to cause the `TOOLS` subdirectory of the EMBEDDED BIOS software directory to be searched for tools *first*. Otherwise, the GSMAKE utility might not be invoked properly.

4.3 Using Other Compilers, Assemblers, and Linkers

You may use other vendors' tools to compile, assemble, and link the EMBEDDED BIOS Adaptation Kit components; however, General Software only directly supports Borland and Microsoft development environments. The base code has been sufficiently tested with enough versions of the Borland and Microsoft tools that it should port reasonably well to other environments.

You should be able to use any development tools for writing applications to run on EMBEDDED BIOS.

4.4 GSMAKE, the Program Maintenance Utility

The General Software GSMAKE utility is used to automate the compilation, assembly, linkage, and other processes necessary to build the components of the EMBEDDED BIOS Adaptation Kit software. It is not necessary that you use GSMAKE for building your own application software.

GSMAKE accepts as input a file, called a MAKEFILE, containing a description of how the source and object files of the system components are processed to produce the finished components. GSMAKE uses the information in the MAKEFILE to produce a *dependency tree* that shows the dependence of finished or intermediate components of the build on the corresponding earlier or source components. For example, an EXE file is dependent on one or more OBJ files, and an OBJ file might be dependent on a C, CPP, or ASM source file and

associated include or header files. MAKE traverses the dependency tree, causing compilers and other tools to be invoked at each step in the tree, in order to properly build the finished product.

4.4.1 Starting GSMAKE

To build a component of the EMBEDDED BIOS Adaptation Kit software, simply move into the associated directory of the software and type:

```
C> GSMAKE
```

The GSMAKE utility automatically invokes the proper build tools in order to rebuild any pieces of the component that are out-of-date. If all the pieces are up-to-date, then GSMAKE responds:

```
GSMAKE: 'all' is up-to-date.
```

GSMAKE's dependency tree mechanism keeps its building process as short as possible; it does not re-invoke compilers, assemblers, linkers, or other utilities unless a file's date/time stamp is changed, and other pieces of the component depend on that file.

4.4.2 Command Line Options

The actual command-line syntax for GSMAKE is the following:

```
GSMAKE [targetname [targetname...]]
       [-f makefilename]
       [-d symbol=value]
       [-x [newmakefilename]]
       [-i] [-z] [-n] [-s]
```

The MAKEFILE read by GSMAKE contains one or more *target* definitions; these targets are usually interdependent. By default, the first target is the one GSMAKE assumes will be the final product (output) of the build process it is to perform. In the MAKEFILEs in the EMBEDDED BIOS Adaptation Kit software, this target is usually 'all'. By specifying an alternate (or set of alternate) target names on the command line, you can add your own targets to the MAKEFILE, or selectively rebuild only a subtree of the entire dependency tree.

GSMAKE can be used to support multiple projects within the same EMBEDDED BIOS component directory. For example, if you are working on two EMBEDDED BIOS adaptations, you may wish to have two MAKEFILEs called PROJECT1 and PROJECT2, and ask GSMAKE to read input specifically from them:

```
C> GSMAKE -f PROJECT1
```

or,

```
C> GSMAKE -f PROJECT2
```

GSMAKE will stop its build process if it encounters an error, if one of the tools it executes encounters an error (not just a warning), or if you press Control-Break. This allows you to incrementally fix a problem without rebuilding everything each time a build is attempted.

Specifying *-i* on the command line causes GSMAKE to continue processing even when it executes a tool during a step that returns an error. This allows you to continue through the build process to see what other activities are ahead.

Specifying *-n* on the command line causes GSMAKE to not execute the commands listed under each target; instead, they are simply printed to STDOUT. This allows you to build a DOS batch file to perform substantially the same thing with a command similar to the following:

```
C> GSMAKE -n > DOIT.BAT
```

Specifying *-s* on the command line disables GSMAKE's echoing of commands to STDOUT. This can help keep the screen tidy during lengthy GSMAKEs.

Specifying *-d symbol=value* on the command line defines the specified symbol to be equal to the specified value, so that they may be used in macro expansions in the MAKEFILE.

Specifying *-z* on the command line causes GSMAKE to stop if a listed dependency on a target is not listed explicitly as a target as well in the MAKEFILE. Ordinarily, a source file (say, TEST.ASM) has no file that it depends on, and should therefore be assumed to be current.

GSMAKE can also examine a basic MAKEFILE and scan the source files specified in the MAKEFILE for include (or header) files. The names of the include and header files are then inserted into the MAKEFILE as additional dependencies with a special option (-x). This option even automates this updating process with a target called "depend:", so all that is necessary to rebuild all of the MAKEFILE's dependencies is to run the command:

```
C> GSMAKE depend
```

4.4.3 Types of MAKEFILE Statements

A MAKEFILE is a simple ASCII file consisting of one or more lines of text. Blank lines are ignored by GSMAKE, as are comment lines beginning with '#'.

Lines starting with a tab are action lines; they are commands to be executed by GSMAKE when bringing a dependency node up-to-date.

Lines not starting with tabs are either intrinsic commands or targets. Targets have the following format:

```
targetname: [dependency [dependency...]]
```

If no dependencies are specified after the colon that follows the target name, then the target is assumed by GSMAKE to be up-to-date automatically.

If dependencies follow, then they name other targets in the MAKEFILE that must be brought up-to-date before the action list can be executed for that target.

If the dependencies are not explicitly defined as targets in the MAKEFILE, then they are assumed by GSMAKE to be the names of files on which the target depends. During its

dependency tree traversal, GSMAKE compares the date/time stamps of the dependency files with the date/time stamps of the targets, and if the targets are older than the dependency files, the action list is executed.

4.4.4 Intrinsic GSMAKE Commands

When GSMAKE reads a MAKEFILE, it scans it line-by-line. Lines starting with a period are assumed to be an intrinsic (built-in) GSMAKE command), as follows:

The `.IFDEF` intrinsic command is used to conditionally read a portion of a MAKEFILE (usually, a set of symbol definitions) based on whether a symbol is defined in the environment or earlier in the MAKEFILE itself. This is the mechanism used to conditionally define symbols for Borland or Microsoft environments in the EMBEDDED BIOS Adaptation Kit MAKEFILEs.

For example, to define the symbol `ASM` to point to either a Borland or Microsoft assembler, the following syntax could be used:

```
.IFDEF BORLAND
ASM=tasm
.ELSE
ASM=masm
.ENDIF
```

The `.ELSE` intrinsic command causes the case of the `.IFDEF` command to be inverted, effecting a C-like ELSE command during reading of the MAKEFILE.

The `.ENDIF` intrinsic command causes the processing of an `.IFDEF` or `.ELSE` block to be terminated, so that the lines that follow the `.ENDIF` command will be read.

The `.DISPLAY` intrinsic command causes output to be written to the display as the MAKEFILE statements are read, *not* as the dependency tree is being traversed (that happens later, after the MAKEFILE is entirely read and the dependency tree generated). Let's augment our example above to indicate clearly which tools are being used:

```
.IFDEF BORLAND
.DISPLAY Using Borland TASM for Assembly
ASM=tasm
.ELSE
.DISPLAY Using Microsoft MASM for Assembly
ASM=masm
.ENDIF
```

The `.STOP` intrinsic command can be used to terminate the reading of the MAKEFILE without MAKE itself returning an error to its caller (usually, `COMMAND.COM`, although GSMAKE can call on itself to build a step). For example, we could cause MAKE to stop immediately if using Microsoft tools:

```
.IFDEF BORLAND
.DISPLAY Using Borland TASM for Assembly
ASM=tasm
.ELSE
```

```
.DISPLAY We don't have Microsoft MASM in our shop.
.STOP
.ENDIF
```

The `.ERROR` intrinsic command can be used to terminate the reading of the MAKEFILE with an error code, so that a MAKE calling the current copy of GSMAKE can terminate its build process. For example, we could use `.ERROR` instead of `.STOP` in our example above to cause a parent GSMAKE to error as well:

```
.IFDEF BORLAND
.DISPLAY Using Borland TASM for Assembly
ASM=tasm
.ELSE
.DISPLAY We don't have Microsoft MASM in our shop.
.ERROR
.ENDIF
```

4.5 GSMERGE, the Merge Utility

The General Software GSMERGE utility is used by the build process if the NOGSMERGE environment variable is not defined. This utility combines all the pieces of a composite BIOS build, including the 16-bit core BIOS, any 32-bit components such as 32-bit PCI and 32-bit BIOS directory services, perhaps a VGA BIOS, and maybe an external SMI BIOS, and writes the resultant output to another file, suitable for programming the boot device.

The goal of this utility is to automate the merging of the Embedded BIOS core image with other option ROM images used in various boards. Such option ROM images may include VGA ROM images, network card images, or BIOS 32 extension images as well as other binaries and extensions, the full listing of which is beyond the scope of this document.

4.5.1 Overview of GSMERGE Operation

GSMERGE works by following a series of commands specified in an image definition file (IDF). This file will be raw text, listing the binaries that will be included as well as the types of the binaries and auxiliary information required to produce a correct final image. The IDF file should be located in the project directory for the specific board being built, along with the other project files and binaries involved in the build. GSMERGE will be called upon to processes the IDF file as part of the build process to produce a final binary image. Part of this process will be the generation of a special binary that describes what binaries are in the final image, and may be placed anywhere or at some defined location within the final image.

4.5.2 IDF File Syntax

The basic syntax for IDF files is simple. Blank space and line feeds are treated the same, except in the case of comments. Blank space is used as a means of separation between keywords and parameters. GSMERGE will also process the IDF file in a linear manner. There are no conditionals or loops in IDF files. All comments begin with a #, and end at the end of that line. Case is also ignored on keywords.

4.5.3 IDF Keywords

In most cases, when a keyword has a numeric field, that value may be supplied in a number of different formats. Valid formats include decimal, hexadecimal, number of kilobytes, number of megabytes or a reference to a map file. All decimal numbers will consist of the numbers 0-9. All hexadecimal numbers will consist of the numbers 0-9, the letters A-F and will have the letter 'h' appended to the end to mark them as hexadecimal. To specify a number of kilobytes, you would append a 'k' to the end of a decimal number and to specify a number of megabytes, you would append an 'm'. Unless otherwise specified, all values provided to keywords are treated as 32 bit unsigned integers.

If you have loaded a map file, you may also specify a symbol from that map file and it will be translated into an appropriate reference value. The syntax for this is:

```
[MapIdentifier]:[SymbolName]
```

MapIdentifier is specified when the map file is included, and SymbolName is the symbol to look up in the map file. See the INCLUDEMAP keyword for details on defining the MapIdentifier. Of special interest is that GSMERGE has context sensitive translation of the SymbolName. When referring to a location within the file being generated, there is an offset that is added to the address extracted from the map file. However, when referring to a location within some external file, the offset is assumed to be from the beginning of that file.

Almost anywhere within an IDF file, you may choose to make reference to a DOS environment variable by using the syntax $(Name). This allows the ability to produce generic .IDF files that may be applied with little or no modification to a large number of situations. You may also add new variables at the command line for GSMERGE. At this time, there is no plan to add the ability to define variables within an IDF file. Note that there are a few places where these variables may not be used. The exceptions will be noted.

4.5.3.1 IMAGEDEF Keyword

IMAGEDEF *FileName*, *FileSize*, *InitialValue*

Function:
This keyword is used to create and define the image that GSMERGE will produce. It may only be used once, and it must be the first meaningful keyword the GSMERGE reads from the .IDF file. The entire file starts out defined as "free". This means that from the beginning, images may be loaded to any portion of the file. As images are loaded into the binary, sections will be reserved to prevent and detect overlapping. Other keywords are provided to manage this behavior.

Options:

FileName	The path/name of the file that GSMERGE will generate.
FileSize	The size of the file that GSMERGE will generate in bytes (Constant).
InitialValue	A byte that GSMERGE will initialize the file to (Constant).

4.5.3.2 AT Keyword

AT *Location*

Function:

This keyword is used to set the location at which our next operation will take place. All normal IDF keywords work around the concept of an operational pointer. This keyword is useful for setting that pointer to a specific location. Note that in effect, this keyword acts as a modifier for operations that follow it.

Options:
Location The offset from the beginning of the file at which following operations start at.

4.5.3.3 ALIGN Keyword

ALIGN *SizeInBytes*

Function:
This keyword is used to align the location of the current file offset to a boundary that aligns from the beginning of the file. This is useful for ensuring that automatically loaded option ROMs are aligned on the correct boundaries. Note that like the AT keyword, align also acts as a modifier for operations that follow it.

Options:
SizeInBytes Used to specify the granularity of the alignment desired.

4.5.3.4 RESERVE and RESERVETO Keywords

RESERVE *Length*
RESERVETO *EndLocation*

Function:
The RESERVE keyword is used to reserve space in the image file, and is used in conjunction with a RESERVETO keyword. When you use this keyword, you signal to GSMERGE that you do not want any future loads to overlap with the specified section of the image. This prevents GSMERGE from loading compressed images and other binaries into this space. Note that once this operation is done, the output file offset will be set to the end of the reserved zone, and all further operations will continue to modify the file from that point on.

Options:
Length How much space you wish to reserve from current location pointer (Constant).
EndLocation The end of a specific section you wish to reserve.

4.5.3.5 FREE and FREETO Keywords

FREE *Length*
FREETO *EndLocation*

Function:
The FREE keyword is used to define space that may already be reserved as free space so that it may be used by GSMERGE for freely allocated binary images. It is used in conjuction with the FREETO keyword. Any section that is free may be loaded into, or may be randomly allocated for use in storing compressed images.

Options:
Length How much space to mark as free from the current location pointer (Constant).
EndLocation The end of a specific section to mark free.

4.5.3.6 SET and SETTO Keywords

SET *Length*, *Value*
SETTO *EndLocation*, *Value*

Function:
The SET keyword is used to set a portion of the image to a specific value. It is used in conjunction with the SETTO keyword to initialize portions of image that must for whatever reason be set to a specific value other then the default initial value specified by IMAGEDEF.

Options:

Length	How much space to set to Value (Constant).
EndLocation	The end of a specific section you want to set.
Value	What byte value to set each byte in the range to.

4.5.3.7 FROM Keyword

FROM *SourceFileOffset*

Function:
This keyword sets the offset from the beginning of a source file for the next image to load. This is used if you don't want to load the entire binary. For example, if you only want to load the last 96k of a BIOS image that has 32k of filler attached to the beginning, you could use FROM 8000h or FROM 32k. After that image is loaded, the offset value will be reset to 0 for the next image load, so you should use FROM for every image load that must skip a certain number of bytes. Note that if a symbol from a map file is specified, it will be assumed for this particular use that the offset is from the beginning of the file being loaded and not from the offset in the destination file defined when the map file was loaded. Note that this keyword acts as a modifier for the next file to be loaded.

Options:

SourceFileOffset	The offset of the first byte of the next file to be loaded.

4.5.3.8 TO Keyword

TO *SourceFileEnd*

Function:
This sets a specific end to a section for the next file loaded into the image. This allows you to set an end boundary to a file section so that you only have to load some of it. Note that this keyword acts as a modifier for the next file to be loaded. Note that this operation is like the FROM keyword, in respect to it's treatment of map files.

Options:

SourceFileEnd	The last byte of the source file to be loaded.

4.5.3.9 INCLUDEMAP Keyword

INCLUDEMAP *MapType*, *MapIdentifier*, *FileName*

Function:
This function includes a map file for a specific binary. It is assumed by the function that the current file offset will be the first address for the elements specified in the map. All addressing will be relative to the start of the image being generated, and address calculation will be automatic. This allows symbols from map files to be used to specify specific locations for AT, FILL or FREE keywords, or any other keyword that requires a location in the binary file. Several types of map file may be supported, specifically including 16 and 32 bit map files. If other formats for map files become available, these may be supported as well.

Options:
MapType The map file type. Current defined types are 16 bit and 32 bit.
MapIdentifier A string that will be used to identify the map file being referred to.
FileName The file name of the map file.

4.5.3.10 LOCATEPE Keyword

LOCATEPE *PhysicalAddress*]

Function:
This function sets the physical address for the base of the binary file being produced, for the purposes of locating portable executable binaries. All actual PE locations will be calculated when that PE file is loaded. All further location will happen automatically. Therefor, this option should be used once, at the top of the file. However, specific overrides can be done if required because the image being stored will later be moved to some other location before usage. Note that this address is also used to resolve what value to use for SETADDRESS.

Options:
PhysicalAddress The physical address from which the beginning of the image will be visible in memory.

4.5.3.11 LOCATERES Keyword

LOCATERES *MediaAddress*

Function:
This keyword sets up the head pointer to a list of resources at the current location. It is assumed that the current image will be made available by the media layer within the BIOS at the specified MediaAddress. In Embedded BIOS 5.0, there is an API for accessing this list, and extracting binary resources from it. In normal usage, this keyword is used to patch that interface so that it can locate the beginning of the list.

Options:
MediaAddress - The media address of the beginning of the binary image being produced.

4.5.3.12 PLACEDIR32 Keyword

PLACEDIR32 *MaxLength*

Function:
This keyword sets the actual location in the output file where the BIOS 32 directory service will be placed. If this function is used, an attempt will be made to patch any BIOS image that has

been loaded with the location of this table. This table will also be manually generated by GSMERGE, rather then patched from a .DLL file.

Options:
MaxLength The maximum amount of space the table could take up.

4.5.3.13 LOAD Keyword

LOAD *Filename*]

Function:
This function loads a binary image. If there is no compression defined, the file will get added directly to the image, and space reserved for it. Otherwise, the location pointer will be advanced, and the image will be checked to be sure that everything will fit if the image were extracted, but no reservation of space will be made for the image.

Options:
FileName The name of the file to load.

4.5.3.14 COMPRESSTO Keyword

COMPRESSTO *Filename*

Function:
This keyword alters the normal functioning of the LOAD keyword by redirecting its output to another file that will be data compressed. This is done because the final size of that file is not known in advance, so it is not possible to simply compress directly into a defined image. The next load after this keyword is used will be effected. After that load has completed, the newly compressed file will be considered finished and no further LOAD keywords will be processed until the next time COMPRESSTO is used.

Options:
FileName The name of the file to compress to.

4.5.3.15 LOADPE Keyword

LOADPE *ServiceName*, *Filename*

Function:
This keyword loads and locates a portable executable, and then adds a special header to the file that defines it as a BIOS 32 module. The BIOS 32 service will also be added to the BIOS 32 service directory, provided that ServiceName is not EXCLUDE. Only one PE module may be loaded as TABLE, and that module must be loaded and located before any other PE module included in the BIOS 32 service directory can be loaded. Any PE module may also be excluded from the service directory by using the EXCLUDE keyword. Note that no module may have a service identifier of more than 4 letters.

Options:
ServiceName The service identifier to use for the BIOS 32 service directory.
Note that ServiceName may NOT be defined in an environment variable.
FileName The name of the portable executable to add to the BIOS.

4.5.3.16 CONVERTBMP Keyword

CONVERTBMP *SourceFile*, *DestinationFile*

Function:
This keyword converts a standard Windows style bitmap file to a special data compressed format used by Embedded BIOS 5.0. It is used for converting a graphic prior to inclusion as a resource.

Options:

SourceFile	The name of the windows .BMP file to convert.
DestinationFile	The name of the final converted file.

4.5.3.17 LOADRES Keyword

LOADRES *ClassValue*, *ResourceId*, *ResourceType*, *FileName*

Function:
This keyword is used to add binary resources to an Embedded BIOS 5.0 build. These resources need not necessarily be contiguous in the final image. LOADRES is used to load resources into the image in small pieces in order to save space and make use of otherwise unusable areas of the final image map. The parameters supplied are used later for locating and extracting the image. ResourceType is especially significant in that it will determine the policy the extractor uses to get at the binary. This type field will be used to tell the extractor if data compression has been used, and if so, what type. This data compression is defined as a process that happens in addition to any data compression inherent within the actual type of binary being added. For example, an RLE image is not data compressed under this definition since RLE compression is inherent in the format and would be decoded by the graphics library. However, if the same file were compressed using the COMPRESSTO keyword and LOAD, you would need to specify the type of compression so that the resource extractor in the BIOS would know to decompress the image before handing it off to the graphics library.

Options:

ClassValue	A 16 bit value that expresses the class of the resource being inserted.
ResourceId	A 16 bit value that individually identifies the resource being inserted.
ResourceType	An 8 bit value that defines the behavior to be used by the resource extractor.
FileName	The name of the binary file that will be included as a resource for extraction.

4.5.3.18 INCLUDE Keyword

INCLUDE *FileName*

Function:
This keyword is used to include an external .IDF file. Such external .IDF files may include standard sections that are identical no matter what project is being merged, such as PCI address patches or standard areas defined as unused. The specified file is opened and processed exactly as if it were part of the current .IDF file in progress.

Options:

FileName	The name of the .IDF file to include.

4.5.3.19 SETADDRESS Keyword

SETADDRESS *OffsetValue*

Function:
This keyword is used to manually link binary files together. This is done because sometimes it is not possible to directly generate a reference to an external binary within another binary that requires such a reference. An example of this is in PCI32, where a reference to the PCIIRQ mapping table is required but generating such a reference within the assembly proved difficult if not impossible. In that circumstance, this keyword is used to patch the PCI32 image so that it is able to reference the 16 bit version of the PCIIRQ table. It places the specified OffsetValue (usually taken from a map file) at the current location within the file. This is always considered to be a 32 bit physical address, based on the value specified by LOCATEPE.

Options:
OffsetValue An address within the image that needs to be referred to.

4.5.4 Example IDF File

The following example is provided for illustrative purposes only, so that the keywords can be seen in the context of their typical use. The actual current image definition file for the named board may actually be different from what is listed here as an example.

```
#
# Binary Image definition file for the MS5169 board.
# (C) 2000 General Software, Inc.  All Rights Reserved.
#
##############################################################################
#       The following commands initialize the output binary image, and      #
#       define where in the machine it will exist.                          #
##############################################################################

        ImageDef $(PROJ).BIN, 128k, FFh # Create 128k binary image.
        LocatePe    0E0000h             # The image starts at E0000h.

##############################################################################
#       Load the actual BIOS binary file at the correct location in the     #
#       output image, as well as loading the map file and all external      #
#       binaries that will be included in the image.                        #
##############################################################################

        At 0                            # Starting at byte 0 of the image.
        IncludeMap 16bit, BIOS16, $(PROJ).MAP # Include the map file.
        load $(PROJ).ABS                # Also load the project binary.

##############################################################################
#       The following section defines all parts of the image that are       #
#       unused and are free for general usage.                              #
##############################################################################

        At BIOS16:EndOfE000             # Start at the first unused byte.
        FreeTo      00ffffh             # Free to the end of the fill area.
        At          BIOS16:FarRetF000   # Start at the last used byte.
        Reserve     1                   # Skip to the first byte of filler.
        FreeTo      01FFFFh             # Free to the end of the BIOS.
        Include ..\resource\idf\meta.idf# Reserve used sections.

##############################################################################
#       The following section defines a section of the final image that     #
#       is un-used and may contain the PCI 32 BIOS services and directory.  #
##############################################################################

#       The following block of code places the image in the F000h segment.

#       At          BIOS16:FarRetF000   # Start at the last used byte.
#       Reserve     1                   # Skip to the first byte of filler.
#       Align       16                  # Align on a paragraph.
#       PlaceDir32  128                 # Reserve 128 bytes for the directory.
#       IncludeMap  32bit, PCI32, $(GSPROJ)\PCIAPI32.MAP # Include the map.
#       LoadPe      $PCI, $(GSPROJ)\PCIAPI32.DLL # Include the service.

#       The following block of code places the image in the E000h segment.

        At BIOS16:EndOfE000             # Start at the first unused byte.
        Align       16                  # Align on a paragraph.
        PlaceDir32  128                 # Reserve 128 bytes for the directory.
        IncludeMap  32bit, PCI32, $(GSPROJ)\PCIAPI32.MAP # Include the map.
        LoadPe      $PCI, $(GSPROJ)\PCIAPI32.DLL # Include the service.

        Include ..\system32\pci\pciapi32.idf # Patch the service binary.

##############################################################################
#       Include splash screen and other graphics.                           #
```

```
##############################################################################
        At BIOS16:ExtResListHead        # Place the location at the list head.
        LocateRes 0000E0000h            # Location of starting media address.

At 0h   Include ..\resource\idf\standard.idf    # include standard graphics.

#       The following graphics are not part of the standard set, but may
#       be needed by some board modules.

#       LoadRes                         # Load a resource into the image.
#       BIOS16:RESOURCE_CLASS_GIMAGE,   # The class is a graphic image.
#       BIOS16:RESOURCE_ID_FS_FLASH,    # The ID is for Flash Drive icon.
#       0,                              # The image is not data compressed.
#       ..\RESOURCE\ICONS\RFD.RLE             # Add Flash Drive test icon.
```

4.6 DISKIMAG, the Disk Image Generator

The General Software DISKIMAG utility transfers raw sectors from any floppy disk or hard disk partition to a binary (unformatted) file suitable for use as input to a PROM programmer.

To create a ROM-based image of a hard drive partition (not a floppy disk), carefully follow the procedure below.

1. Copy the files you wish to the hard disk partition.

2. Run DEFRAG (an MS-DOS defragmentation utility program) to condense the contents of the partition to the front of the partition. Make sure from the map displayed by DEFRAG that the contents of the disk will fit into the size of image you are making. Just because you can see directory entries for files doesn't mean that the actual contents of those files (recorded in clusters elsewhere on the disk) will actually fit in the size you select.

3. Run the General Software INSTBOOT utility on the hard disk partition if it is to boot Embedded DOS. Beware Windows 95 and Windows NT users: While this utility performs its work correctly, Windows may patch the boot record again and destroy it unless this is the last thing you write to the disk before running DISKIMAG. If you copy more data to the disk, make sure you repeat this step.

4. Run DISKIMAG on the floppy to create a file that contains its image, or the first portion of the partition. Use the /P option to cause DISKIMAG to create an MBR with a partition table in it, so that the partitioning structure is created in your output file.

To create a ROM-based image of a floppy disk (not a hard drive partition), carefully follow the procedure below.

1. Format a floppy (even if it has been previously formatted and used for other things). Your desktop DOS will not always start writing files at the beginning of the floppy, and this can cause problems if you are only making an image file that is half or a quarter of the size of the floppy, as these files will be missed.

2. Copy (without any intervening deletes) the files you wish to the floppy.

3. Run the General Software INSTBOOT utility on the floppy if it is to boot Embedded DOS. Beware Windows 95 and Windows NT users: While this utility performs its work correctly, Windows may patch the boot record again and destroy it unless this is the last thing you write to the disk before running DISKIMAG. If you copy more data to the disk, make sure you repeat this step.

4. Run DISKIMAG on the floppy to create a file that contains its image, or the first portion of the floppy.

Starting DISKIMAG

DISKIMAG is run from the command line with at least two, and sometimes a third, argument, as follows:

```
C> DISKIMAG  d:  filename  [kb_to_copy]  [/P]
```

The *d:* operand specifies the drive letter from which to read raw sectors. This must be A: or B:.

The *filename* operand specifies the name of the output file to copy the raw sectors into as a contiguous byte stream.

The *kb_to_copy* operand is optional. If omitted, DISKIMAG assumes that you wish to copy 1MB of data from the floppy. Otherwise, if specified, it is a number of kilobytes (1024 byte units) of data to transfer from the floppy. Note that 1K is two sectors for 512-byte sectors.

The */P* switch is optional. If omitted, DISKIMAG will create an image of a floppy or hard disk partition by itself. If specified, DISKIMAG will create the partition table necessary for creating images of hard drive partitions, so that the ROM disk driver will be able to present this information to the operating system running on the target.

For example, to copy 1.44MB from your drive A: to a file called `OUTPUT.BIN`, you would use the following command:

```
C> DISKIMAG  B:  OUTPUT.BIN  1440
```

4.7 BIOSLOC, the ROM BIOS Extension Locator

The General Software BIOSLOC utility functions as a simple EXE relocation program, allowing you to locate your EXE-based application code and ROM extensions to any fixed segment address.

Starting BIOSLOC

BIOSLOC is run from the command line with two operands.
To convert an EXE into an ABS file, simply specify as operands the name of the EXE file (without the EXE extension) and the relocation segment address (optional). If you do not specify a relocation address, BIOSLOC chooses f000h (the normal segment used by the BIOS).

To relocate a sample application program called `TEST.EXE` to a fixed segment address at D000h, use the following command:

```
C> BIOSLOC  TEST  d000
```

To relocate your linked BIOS called `BIOS.EXE` to the default segment for a BIOS at F000h (recommended), use the following command:

```
C> BIOSLOC  BIOS
```

4.8 BIOSSUM, the ROM BIOS Extension Checksum Utility

The General Software BIOSSUM utility is used to checksum a binary file formatted as a ROM extension, and to edit a reserved byte in the header from 00h to a value that is the logical complement of the checksum of the rest of the file. In this way, all the bytes in the file are made to add to 00h so that when the image is burned into ROM, it is eligible to be recognized as a BIOS extension.

```
CODE    SEGMENT
RomSig  db      55h, 0aah
        db      4                  ; number of blocks.
        db      0eah               ; FAR JMP instruction.
        dw      OFFSET CODE:InitRomDisk
        dw      CODE

        db      'CHECKSUM.BYTE-->' ; searched for by BIOSSUM.
CheckSum db     0                  ; modified by BIOSSUM.
        ...
CODE    ENDS
```

Figure 4.1. Typical ROM BIOS Extension Header.

The code fragment shown in Figure 4.1 illustrates how a typical ROM BIOS extension module might begin with a ROM header.

The first two bytes of a ROM extension are always 55h and aah. The BIOS scans in 2KB increments through memory starting at C000h through ED00h (this is configurable depending on the underlying BIOS) and looks for these signatures on 2KB boundaries.

The third byte (4) in our example specifies the number of 512-byte blocks to scan when performing a checksum of the ROM BIOS extension. The BIOS multiplies the number by 512 and scans that many bytes starting with the first byte of the header. If all the bytes add up to zero, then the ROM BIOS extension is actually called during initialization.

In order to make this ROM BIOS extension checksum work properly, the BIOSSUM utility is needed to perform a checksum on the blocks just as though the BIOS were doing it, and calculate what needs to be done to the image to bring its checksum to 0. This value is computed (a byte), and is stored in the zero field that follows the magic string, "CHECKSUM.BYTE-->". BIOSSUM will not work unless this string is present.

Starting BIOSSUM

BIOSSUM is run from the command line with only one operand, the name of the file to be checksummed and patched.

Suppose we had a ROM BIOS extension written in `EXTEND.ASM`. We would assemble and link the module to form `EXTEND.EXE`, which is not yet relocated to the proper address. Then we would run BIOSLOC to locate it to the proper address:

```
C> BIOSLOC EXTEND C000
```

and then run BIOSSUM to compute the checksum on the relocated file. It is important to run BIOSSUM after running BIOSLOC in this procedure or the BIOSSUM program will sum an EXE file, not an ABS file. Example:

```
C> BIOSSUM EXTEND
```

4.9 BIOSMAP, the EMBEDDED BIOS Map File Analyzer

The BIOSMAP utility is used during the EMBEDDED BIOS system build to determine how to pad the BIOS image with data so that the resulting file is exactly 64KB in length. This is necessary because the BIOS is linked together from many separately-assembled modules containing multiple segments, and there is no way to ORG to an absolute location in MASM or TASM.

BIOSMAP works by closely examining the MAP file produced during the assembly of module POST. When run without operands, BIOSMAP does not look at this MAP file, but instead produces a file called `BIOSFILL.INC`, which is included in POST. After reassembly, BIOSMAP is run again with a command line argument "BIOS", specifying the MAP file to inspect. During its pass through the `BIOS.MAP` file, it looks for the symbol, OFFSET_FF00, and determines the actual offset that this symbol is associated with. Then, it modifies its `BIOSFILL.INC` file to contain DB statements that would correctly position the OFFSET_FF00 symbol to the proper offset.

4.10 PERF, the File System Performance Analyzer

The General Software PERF utility is a file system performance analysis tool that can be used to benchmark your build of EMBEDDED BIOS. This can help you to understand how various open modes, record sizes, and file sizes will constrain the overall throughput of your embedded application *before* writing any code.

4.10.1 Starting PERF

PERF is executed from the DOS command line with a carefully-selected set of switches (switches can be prefixed by the '/' or '-' characters as desired).

Fundamentally, PERF performs selected options on a specified number of files. The files may automatically be named `0.DAT`, `1.DAT`, `2.DAT`, and so on, or may include your selected prefix; i.e., `TEST0.DAT`, `TEST1.DAT`, `TEST2.DAT`, and so on. By default, only one file is used during the test. These files are called the data files.

PERF can create a new data file or open an existing one. By default, it attempts to open an existing file, unless the *-c* is specified. PERF leaves the data file on the medium unless specifically asked to delete the file at the end of its run with the *-d* switch.

PERF's main function is to perform reads, writes, or both reads and writes; either randomly or sequentially; and with varying record sizes; within a file size of your choosing. These options are all configured with command line switches. If you do not select to have PERF read or write data, then it will simply perform any creation and deletion functions you specified (see above, creating new data files).

4.10.2 Command Line Options

The actual command-line syntax for PERF is the following:

```
PERF   [-r] [-w] [-k] [-c] [-d] [-i]
       [-x[:skip]]
       [-s:recsize]
       [-l:filesize]
       [-f:nfiles]
       [-o:filename]
       [-v:ON|OFF]
       [-m:passes]
       [-n:repetitions]
```

The *-r* switch specifies that PERF should perform reads from the file into a read buffer (automatically allocated by PERF). Unless *-x* is specified, the reads will be sequential without intervening random seeks.

The *-w* switch specifies that PERF should perform writes to the file from a write buffer (automatically allocated and initialized by PERF to a known pattern). Unless *-x* is specified, the writes will be sequential without intervening ramdom seeks.

The *-k* switch specifies that PERF should apply locking around each I/O to the file. By default, no lock or unlock operations are performed. Note: You must have the `SHARE.EXE` program loaded under generic DOS to support record locking; however, Embedded DOS contains this support in the FAT FSD.

The *-c* switch specifies that PERF should create the file before operating on it on each pass (the file is destroyed if it already exists). This switch is required if the file doesn't already exist.

The *-d* switch specifies that PERF should delete the file after each pass.

The *-i* switch specifies that PERF should compare the contents of each record read from the file with the data it expects as a data integrity check. Normally, both reading and writing are enabled when this option is selected.

The *-x:skip* option specifies that random I/O should be performed. By default, sequential I/O is performed instead. The *-x* switch can be specified alone or with a colon followed by a *skip factor* that dictates how the file pointer is to be advanced after each I/O. A skip factor of zero does not move the file pointer. A skip factor of one advances the file pointer by 1. A skip factor equal to the selected record size effectively performs sequential I/O. The default skip value is 1KB.

The *-s:recsize* option specifies the number of bytes to read or write on each I/O operation. The size may be any value from 0 to 65534. The default *recsize* is 1KB.

The *-l:filesize* option specifies the total number of bytes in the file to be used for I/O. If the file is being written, then this will be the resulting size of the file. If the file is being read, then only this many bytes of the file will be processed. The *filesize* should be a multiple of the *recsize* value. The default *filesize* is 64KB.

The *-f:nfiles* option specifies the number of files to simultaneously interact with on each pass. The default number of files is one; this value may be extended to 15 (due to the handle limitation in DOS).

The *-o:filename* option specifies an optional name prefix for the file(s) to be operated on. By default, files are named `0.DAT`, `1.DAT`, and so on.

The *-v:ON|OFF* option specifies whether PERF should set the DOS VERIFY flag (and therefore enable/disable the Embedded DOS cache) before running passes. By default, PERF leaves the VERIFY setting alone. If ON is selected, then PERF sets `VERIFY=ON` (thereby disabling the cache), and then performs I/O. If OFF is selected, then PERF sets `VERIFY=OFF` (thereby enabling the cache), and then performs I/O.

4.10.3 Multiple Passes

The PERF program performs all of the selected I/O within a given *pass*. At the end of the pass, the statistics (start time, duration in milliseconds, and calculated KB/sec for that pass are displayed, along with averaged statistics for multiple passes. By default, PERF performs just one pass.

To specify multiple passes, use the *-m:nnnn* switch. For example, to perform 10 passes of the same I/O type, use the following command:

```
C> PERF [...other options here...]  -m:10
```

4.10.4 Multiple Repetitions per Pass

PERF can be programmed to perform multiple repetitions of your I/O per pass through the file. If selecting multiple repetitions (the default is just one repetition per pass), then PERF does the following:

- Opens/Creates the data file(s) for Pass #1
- Repetition #1
- Repetition #2
- Repetition #3
- Closes/Deletes the data file(s) for Pass #1
- Prints a single summary line representing all three repetitions of Pass #1

- Opens/Creates the data file(s) for Pass #2
- Repetition #1
- Repetition #2

- Repetition #3
- Closes/Deletes the data file(s) for Pass #2
- Prints a single summary line representing all three repetitions of Pass #2

This allows you to keep the data file(s) open during testing so that the operating system DosOpen, DosClose, DosCreate, and DosDelete functions are not exercised (causing them not to affect the file system cache, for example).

Multiple repetitions are specified with the *-n:nnnn* switch. For example, to program PERF to run two passes of three repetitions, use the following command:

```
C> PERF [...other options here...]  -m:2  -n:3
```

4.10.5 Some Examples

The following command creates a 256KB file called `0.DAT` with 1KB writes, and leaves it on disk:

```
C> PERF -c -l:256k -s1k
```

The following command performs 10 passes of sequential 32KB reads from the file created, above.

```
C> PERF -r -l:256k -s32k
```

The following command creates ten 1KB files called `TEST0.DAT`, `TEST1.DAT`, and so on) with 4-byte writes:

```
C> PERF -f:10 -o:test -s:4 -l:1k -m:10
```

The following command performs 10 passes of five repetitions each, creating 3 files, doing random reads and writes to each file, closing them, and deleting them, with VERIFY OFF:

```
C> PERF -v:OFF -f:3 -m:10 -n:5 -c -r -w -d
```

Chapter 5

BUILDING EMBEDDED BIOS

This chapter describes the process by which a System BIOS is made using the EMBEDDED BIOS Adaptation Kit software, often in conjunction with code written for the specific board.

5.1 Building the System BIOS

The primary product of using the EMBEDDED BIOS Adaptation Kit is a file that contains a binary image that can be burned into EPROM or programmed into a Flash ROM. The file's name is BIOS.ABS. The file's size is rounded up to the next 64KB required to support the requested build size; i.e., 64KB, 128KB, 196KB, or 256KB. Called the System BIOS or Core BIOS to distinguish it from other BIOS components such as VGA BIOS Extension, this file is built by the following process.

This system BIOS file is often combined with other components, such as 32-bit PCI services modules and VGA BIOS extensions. Called by the MAKEFILE, and operating under the direction of commands in an .IDF file in the project directory, the GSMERGE utility produces the final output file from these many sources, one of which is the output from the system BIOS build with its .ABS extension. Commonly, the final output file with all of these components combined actually has a .BIN extension. The merge step, or even the building of 32-bit components, can be disabled with environment variables. For more information, see section 5.1.8 in this Chapter. The remainder of this chapter deals with the 16-bit system BIOS build.

5.1.1 Configuring Build Options and Parameters

First, the options and `OPTIONS.INC` and `CONFIG.INC` files must be reviewed so that any changes to these defaults can be recorded in the project file. BIOStart automatically reads these files and gives the OEM point-and-click graphical access to these parameters, and changes are automatically recorded to the project file. If you're manually changing the parameters, you'll be creating a project file yourself with a text editor, and adding lines that contain redefinitions of the symbols in these two files.

5.1.2 Selecting the CPU Personality Module

Next, make sure you have a CPU Personality Module (CPM) for the CPU Class you have selected in the project file. The **NOCPU** CPM provided with EMBEDDED BIOS knows about the generic line of Intel 8086, 80286, 80386, i486, Pentium, Pentium II, Pentium III, and similar processors. It doesn't know how to use the advanced features of related processors, such as the 486-SLC, for example, but in many cases a 486-SLC can be treated like any other i486 processor. Similarly, Pentium, Pentium II, Pentium III, Celeron, and K6 Processors do not need a special CPU module, although they do require support from board and chipset modules to be supported by the BIOS adaptation.

If you have a CPU that cannot be supported with the provided CPMs, then you'll need to clone the provided TEMPLATE.ASM and TEMPLATE.INC files in the CPUS\TEMPLATE subdirectory, and create a new subdirectory under CPUS that corresponds to the new CPU name. For example, if your CPU was named ROVER, then you would create a ROVER subdirectory under CPUS, and copy CPUS\TEMPLATE\TEMPLATE.ASM to CPUS\ROVER\ROVER.ASM, and copy CPUS\TEMPLATE\TEMPLATE.INC to CPUS\ROVER\ROVER.ASM. Finally, change the INCLUDE directive in ROVER.ASM to point to the correct file (ROVER.INC), and then change the routines you need changing. After making modifications, be sure you remove all unchanged routines from this file. This keeps the file small, and ensures that the BIOS default routines are used, even in the situation where you've updated the core BIOS source code from General Software, and some of those default routines have changed for the better.

Finally, in any case, insert a line in your project file that defines the CPU CPM as follows (note this is just an illustrative example, that would need to be changed, depending on what CPU type you're actually using).:

```
CPUCLASS      EQU    <NOCPU>       ; or substitute ROVER for NOCPU.
```

5.1.3 Selecting the Chipset Personality Module

Next, make sure you have a Chipset Personality Module (CSPM) for the chipset you have selected in the project file. The **NOCHPSET** CSPM provided with EMBEDDED BIOS is an empty placeholder module that performs no special chipset programming. It is used in designs without chipsets or VLSI blocks with chipset-like qualities.

If you have a chipset that cannot be supported with the provided CSPMs, then you'll need to clone the provided TEMPLATE.ASM and TEMPLATE.INC files in the CHIPSETS\TEMPLATE subdirectory, and create a new subdirectory under CHIPSETS that corresponds to the new chipset name. For example, if your chipset was named AIRFOIL, then you would create a AIRFOIL subdirectory under CHIPSETS, and copy CHIPSETS\TEMPLATE\TEMPLATE.ASM to CHIPSETS\AIRFOIL\AIRFOIL.ASM, and copy CHIPSETS\TEMPLATE\TEMPLATE.INC to CHIPSETS\AIRFOIL\AIRFOIL.ASM. Finally, change the INCLUDE directive in AIRFOIL.ASM to point to the correct file (AIRFOIL.INC), and then change the routines you need changing. After making modifications, be sure you remove all unchanged routines from this file. This keeps the file small, and ensures that the BIOS default routines are used, even in the situation where you've updated the core BIOS source code from General Software, and some of those default routines have changed for the better.

Finally, in any case, insert a line in your project file that defines the chipset CSPM as follows (note this is just an illustrative example, that would need to be changed, depending on what chipset you're actually using).:

```
CHIPSET       EQU    <NOCHPSET>    ; or substitute AIRFOIL for NOCHPSET.
```

5.1.4 Selecting the Board Personality Module

Next, make sure you have a Board Personality Module (BPM) for the board design you have selected in the project file. The **NOBOARD** BPM provided with EMBEDDED BIOS is an empty placeholder module that performs no special board (Super I/O, etc.) programming. It is used in designs without much glue or special-purpose devices present.

If you have a design that cannot be supported with the provided BPMs, then you'll need to clone the provided `TEMPLATE.ASM` and `TEMPLATE.INC` files in the `BOARDS\TEMPLATE` subdirectory, and create a new subdirectory under `BOARDS` that corresponds to the new board design's name. For example, if your board was named `FROG`, then you would create a `FROG` subdirectory under `BOARDS`, and copy `BOARDS\TEMPLATE\TEMPLATE.ASM` to `BOARDS\FROG\FROG.ASM`, and copy `BOARDS\TEMPLATE\TEMPLATE.INC` to `BOARDS\FROG\FROG.ASM`. Finally, change the `INCLUDE` directive in `FROG.ASM` to point to the correct file (`FROG.INC`), and then change the routines you need changing. After making modifications, be sure you remove all unchanged routines from this file. This keeps the file small, and ensures that the BIOS default routines are used, even in the situation where you've updated the core BIOS source code from General Software, and some of those default routines have changed for the better.

Finally, in any case, insert a line in your project file that defines the BPM as follows (note this is just an illustrative example, that would need to be changed, depending on what board you're actually using).:

```
BOARD        EQU    <NOBOARD>     ; or substitute FROG for NOBOARD.
```

5.1.5 Type GSMAKE in DOS or in a DOS Box Under Windows

Everything is automatic—do not try to do any of the build steps by hand. Remember that if you're using Borland tools, you must use the "`SET BORLAND=YES`" DOS command prior to running GSMAKE, so that it isn't using Microsoft tools. Then, use the "`SET GSPROJ=projectfilename`" DOS command to tell GSMAKE which project to build.

Before we try it, there are actually two more set-up items, one to create the project subdirectory and project file underneath the PROJECTS subdirectory, and then again to create a subdirectory by the same project name under the SYSTEM\OBJ subdirectory. For the sake of illustration, let's use MYPROJ as the project name in the following example and text that discusses it.

Here's how to build the BIOS from the beginning:

```
C:\> CD \EBIOS43\PROJECTS
C:\EBIOS43\PROJECTS> MD MYPROJ
C:\EBIOS43\PROJECTS> MD ..\SYSTEM\OBJ\MYPROJ
C:\EBIOS43\PROJECTS> EDIT MYPROJ\MYPROJ.INC
C:\EBIOS43\PROJECTS> SET BORLAND=YES
C:\EBIOS43\PROJECTS> SET GSPROJ=MYPROJ
C:\EBIOS43\PROJECTS> GSMAKE
... all files are assembled and then linked ...
C:\EBIOS43\PROJECTS>                    [you're finished]
```

5.1.6 Inspecting the Binary 16-Bit System BIOS File

The output of the 16-bit build process described in 5.1.5 is a file called `C:\EBIOS43\PROJECTS\MYPROJ\MYPROJ.ABS`, assuming you really used MYPROJ as a project name. You may wish to inspect the file with a hex viewer such as one provided by Norton, for example.

Lacking any fancy PC tools, you can use the MS-DOS DEBUG utility to see the contents of the file. The only debugger commands you need to know are 'd' for dump out more bytes, and 'q' for quit. You will need to use a slightly different procedure depending on the size of the BIOS you have built, depending on your setting of the **OPTION_BIOS_KBSIZE** parameter.

Assuming you've created a 64KB BIOS, you can use the following DEBUG sequence to display the BIOS header at the beginning of the `MYPROJ.ABS` file:

```
C:\EBIOS43\PROJECTS> DEBUG MYPROJ\MYPROJ.ABS

- d 100

... Hex dump follows here, you should see EMBEDDED BIOS
... Copyright message.

- q                                     [exit to DOS]

C:\EBIOS43\PROJECTS>
```

If you specified an 8KB size, the `MYPROJ.ABS` image is padded at the beginning with 56KB of 0xff bytes before the real BIOS image starts. You'll need to use the following DEBUG sequence to display the BIOS header at that location:

```
C:\EBIOS43\PROJECTS> DEBUG MYPROJ\MYPROJ.ABS

- d e100

... Hex dump follows here, you should see EMBEDDED BIOS
... Copyright message.

- q                                     [exit to DOS]

C:\EBIOS43\PROJECTS>
```

If you specified a 16KB BIOS, then the `MYPROJ.ABS` image is padded at the beginning with 48KB of 0xff bytes before the real BIOS image starts. You'll need to use the following DEBUG sequence to display the BIOS header at that location:

```
C:\EBIOS43\PROJECTS> DEBUG MYPROJ\MYPROJ.ABS

- d c100

... Hex dump follows here, you should see EMBEDDED BIOS
... Copyright message.

- q                                     [exit to DOS]

C:\EBIOS43\PROJECTS>
```

If you specified a 32KB BIOS, then the `MYPROJ.ABS` image is padded at the beginning with 32KB of 0xff bytes before the real BIOS image starts. You'll need to use the following DEBUG sequence to display the BIOS header at that location:

```
C:\EBIOS43\PROJECTS> DEBUG MYPROJ\MYPROJ.ABS

- d 8100

... Hex dump follows here, you should see EMBEDDED BIOS
... Copyright message.

- q                                     [exit to DOS]

C:\EBIOS43\SYSTEM>
```

5.1.7 Programming a Boot ROM with MYPROJ.ABS

As previously mentioned, `projectname.ABS` is a binary file that contains a binary image of the 16-bit BIOS to be programmed into a boot Flash part. It does not contain "hex records" or other formats used by older ROM burners.

Take precautions when burning ROMs or programming flash. Make sure that all of the leads of your parts are clean, straight, and are handled with the proper static controls. Also make certain that the parts are plugged into their sockets in the right direction; it is easy to create a fire with EPROMs plugged in backwards in some system boards!

5.1.8 The 32-bit BIOS Build, and Composite BIOS Files

For many simple designs, a simple 16-bit system BIOS suits the requirements for pre-boot firmware. However, if your design requires 32-bit BIOS directory services, 32-bit PCI services, Microcode Update modules, or embedded VGA and/or VSA ROM extensions, you will need to create these components and merge them together into one big output file, which includes not only these components, but the 16-bit system BIOS as well. The utility that performs this function is GSMERGE (described in Chapter 4) and its operation is governed by a special .IDF script file in the project directory.

The production of a composite output file (typically, with a .BIN extension) is automatic if an IDF file is present. The MAKEFILE that directs GSMAKE to build all of the 16-bit system BIOS components will automatically invoke the build of the 32-bit BIOS components, such as the 32-bit PCI services module, and run GSMERGE to combine everything as directed by the IDF file.

To defeat the 32-bit BIOS builds, or to defeat the execution of GSMERGE as a part of the standard build process, see the instructions in Chapter 4 for setting the appropriate environment variables. It should be noted that, if GSMERGE does not find an IDF file, it will perform no action, so that a merge process will not take place. IDF files are provided for all standard project files supplied by General Software for which composite builds are needed. For example, those that have special SMI extensions (VSA) or those that have 32-bit PCI services (all standard PCI-based reference designs).

As with the .ABS files, `projectname.BIN`, the output of the GSMERGE activity, is a binary file that contains a binary image of a ROM to be burned. It already includes the .ABS file data, so

you should consider the .ABS file to be more of a temporary file used during the build. Like the .ABS file, the .BIN file has no internal structure-- it does not contain "hex records" or other formats used by older ROM burners.

5.1.9 Booting the System

Your ROMs are installed in their sockets and you are ready to apply power to the motherboard. Make sure that your work area is free of screws, tools, and bits of wire that can create shorts when you least want them to.

Also check to see that you've plugged in all the peripherals that are being supported by the BIOS. If you are working with a motherboard, connect the PC keyboard and add a VGA monitor and VGA card if possible, so that a maximum amount of status can be determined when the system boots for the first time. Then, apply power to the target.

If you are working with a target that closely resembles an ISA desktop PC, your chances of booting DOS at this point are excellent. Within a few hours, a non-booting system will become bootable by carefully reviewing the configuration parameters, and making sure you have the right Chipset, CPU, and Board Personality Modules in the build.

Finally, turn to Chapter 8 for a much more in-depth discussion of the kinds of issues that may need to be addressed in your BIOS build.

5.2 Building Auxilliary Components

All of the EMBEDDED BIOS tools come precompiled in the TOOLS directory, so there is no need to run GSMAKE in the TOOLS directory. Any additional components of EMBEDDED BIOS that are not a part of the core System BIOS however, are built in the UTIL directory.

Before building the utilities themselves, you'll need to build the Character Oriented Window package used by the HOST.EXE program that demonstrates Manufacturing Mode. Do the following <u>before</u> going into the UTIL directory to build the software that uses COW:

```
C:\> CD \EBIOS43\COW

C:\EBIOS43\COW> SET BORLAND=YES

C:\EBIOS43\COW> GSMAKE
... all files are assembled and then linked ...
C:\EBIOS43\COW>                  [you're finished]
```

To be certain that COW compiled correctly, the output of this build produced a TEST.EXE program that can be run right away. It has no functional purpose other than to paint the screen and exercise the list boxes and dialog boxes. Once this is done, you're ready to move on to building the real utility programs below.

To build the auxilliary components (such as the Remote disk server that runs host-side and test HOST program used with the Manufacturing Mode), change into the UTIL directory and run "GSMAKE". Again remember that if you're using Borland tools, you must use the "SET BORLAND=YES" DOS command prior to running GSMAKE, so that it doesn't using Microsoft tools. Here's how to build the UTIL components from the beginning:

```
C:\> CD \EBIOS43\UTIL

C:\EBIOS43\UTIL> SET BORLAND=YES

C:\EBIOS43\UTIL> GSMAKE
```

Here, all files are assembled and then linked.

```
C:\EBIOS43\UTIL>                  [you're finished]
```

To see if the HOST program was compiled correctly, run the HOST program. It will display a full screen menu that does not require Manufacturing Mode to run until an option is selected.

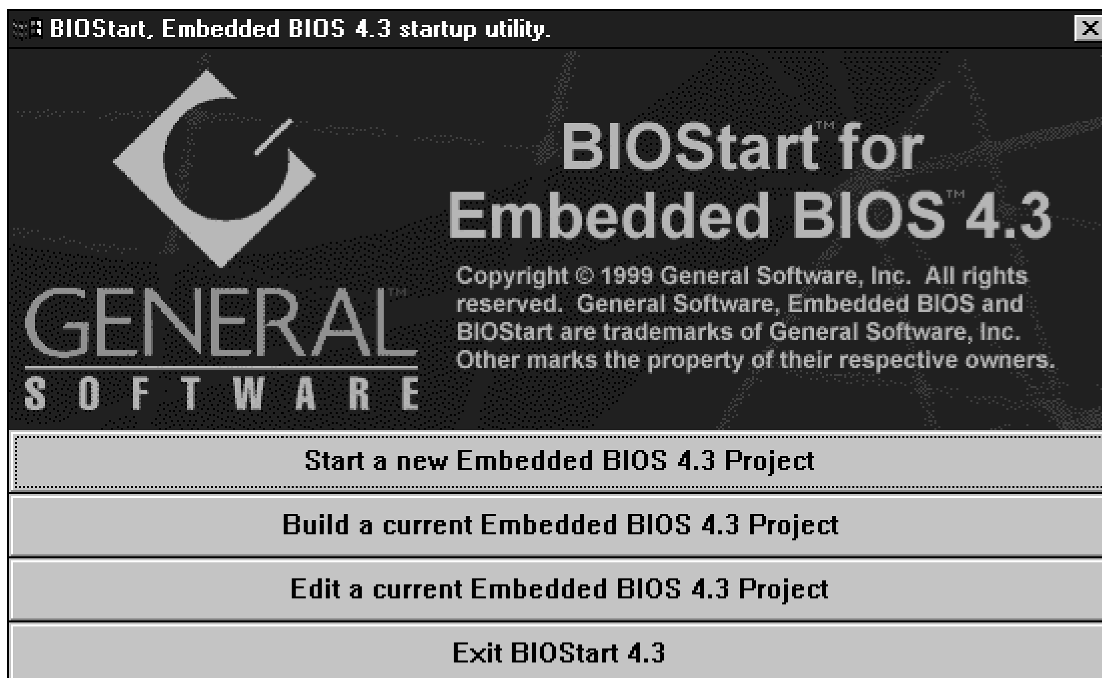
Chapter 6

CONFIGURING THE BIOS WITH BIOSTART

EMBEDDED BIOS offers so many configurable options that configuration automation is almost essential for the adaptation engineer with little BIOS adaptation experience. BIOStart provides guided access to BIOS source-level configuration options in a hierarchical manner in a Windows environment.

6.1 Overview of BIOStart

The real purpose of BIOStart is to provide the engineer with easy, high-level access to the EMBEDDED BIOS configuration process. Within the BIOStart environment, the engineer can manage projects, customize options and operating parameters, and build a binary ready-to-ROM version of the BIOS from the source code with these customized options without ever leaving Windows and without ever having to modify a .INC file by hand with a text editor.



BIOStart makes this process easy and simple for the casual user, but provides more depth to customization for the engineer with special needs. Especially useful for the experienced BIOS adaptation engineer is the on-line descriptions of options and the links to other options that interact with one another.

Prior to BIOStart's introduction, configuration of any BIOS was performed by hand, using a text editor. Actually, most BIOS adaptation kits use configuration symbols embedded directly in the source code; a handful of symbols is common. Because of the explosive growth of EMBEDDED BIOS's complexity in terms of supported features (today with over 400 configuration options), a method of managing the options and coordinating their relationships was necessary.

This condition can be likened to modern jet fighters requiring on-board computers to keep them flying; without the flight computers, the jets would quickly stop flying, but with the computers, precision flying and flying to the limits imposed by the pilot, not the equipment, are all possible. In modern fighter aircraft, the pilot doesn't control the fine details of getting there, but is assisted to a large extent by computers. BIOStart provides the management tool for this complexity in the BIOS configuration process, so that its full capabilities can be harnessed.

The BIOStart user interface therefore, has a special design; a cross between a wizard and a Windows help browser. During the customization phase, the interface has some hypertext elements found in WEB browsers that offers the user different views of associated configuration options and parameters.

BIOStart allows the BIOS adaptation engineer to hit the ground running, rather than reading thick manuals. EMBEDDED BIOS is a complex product, and while there may be time to review all of its documentation before building the first BIOS, it can provide a shortcut that allows real BIOS adaptations to be produced in a manner of minutes. It does this by eliminating common configuration problems that can occur when conflicting configuration options are used, or when parameters have not been properly selected for some options.

For example, a specific version of the BIOS may not support all of the hardware features of the board for which it was designed (i.e., cache memory control enabled but no cache memory available on the board). BIOStart will warn its user of such potential problems, and although the engineer will be allowed to enable such features, a warning is generated. BIOStart will fix all settings that generate a warning, setting them to what engineers at General Software have coded as the best value for that board.

Despite its power, BIOStart is not a substitute for good planning, or for actually writing the customization code that may be necessary as a part of the CPU Personality Module, the Chipset Personality Module, or more commonly, the Board Personality Module.

BIOStart cannot help someone who knows nothing about the hardware design a working BIOS. Basic knowledge of BIOS and hardware terminology is required to use it. However, provided the engineer has this basic knowledge, BIOStart is powerful enough to make the source code configuration process an easy task. No knowledge of assembly language or the actual structure of the source code is required to use BIOStart. Nor is it necessary to know about all of the 400 parameters, because BIOStart will guide you to the settings that you need while shielding you from the settings that may cause problems if set by a novice, or are irrelevant to the hardware you are designing.

6.2 Installing the Adaptation Kit with BIOStart

BIOStart also doubles as the Windows installation utility for the Adaptation Kit itself. It lets you specify which directory you wish to expand the source tree in and it executes a batch file that installs the BIOS. It then allows you to specify if you are using Borland tools or not. At the time of this writing, if you are using MASM 6.1, 6.11, or 6.14, you will need to manually set the MASM61 environment variable in your `AUTOEXEC.BAT` file. If you specify that you use Borland tools, then BIOStart will ask if you wish to update your `AUTOEXEC.BAT` file. On reboot, the `AUTOEXEC.BAT` will set the appropriate environment variable for you. BIOStart currently requires that that variable be set. Finally, BIOStart will create a General Software program group in the program manager, or on the Start menu.

When setup has finished installing everything, it will attempt to find the source directory again. If it can, BIOStart starts, and if you didn't have to modify the `AUTOEXEC.BAT`, you can proceed with the BIOS design process. If the `AUTOEXEC.BAT` was modified, you will need to exit BIOStart, and reset the computer.

If you are re-installing Embedded BIOS, or you need to run BIOStart as setup again, go into the Windows directory and delete the file `BIOStart.INI`. This is where BIOStart stores the location of the source code tree. If you delete this file, BIOStart will be unable to find the source code, and will enter into install mode. If you wish to specify a pre-existing installation of Embedded BIOS 4.3, tell BIOStart that you wish to install Embedded BIOS and then specify the path of the source tree for Embedded BIOS. Then ignore or answer no to all of the other installation questions. You can also modify the `BIOStart.INI` file directly with a text editor.

6.3 Creating and Editing a Project

BIOStart is a configuration utility designed to allow the adaptation engineer to modify existing EMBEDDED BIOS projects or even create new projects from scratch. We recommend that whenever possible, engineers start with a pre-existing project. When extensive modifications are likely to be made to the board, the chipset or the CPU, the engineer should clone pre-existing modules using BIOStart. Here is how you do this:

1. Open BIOStart. Windows 3.1 users should click on the BIOStart icon in the General Software program group in program manager. Windows 95 and Windows NT users can click on the **[Start]** button, and select BIOStart from the General Software list under applications.

2. Start a new Embedded BIOS 4.3 Project. Click on the Start a new Project button, select and clone a board module. Click on the listed board module that most closely resembles your project (see Figure 6.1). Click on the button labeled **[Clone Selected Module]**. This will open up the Clone New Module dialog box.

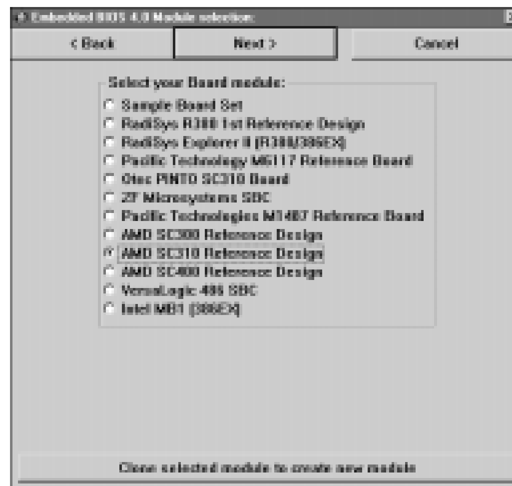


Figure 6.1. Board Personality Module selection.

From this project screen, all of the main components will be selected. Enter the title of the new board module in the upper text box. This is the text that will appear on the button every time you wish to create a new project, so enter something that will distinguish the module from the other entries in the list (see Figure 6.2).

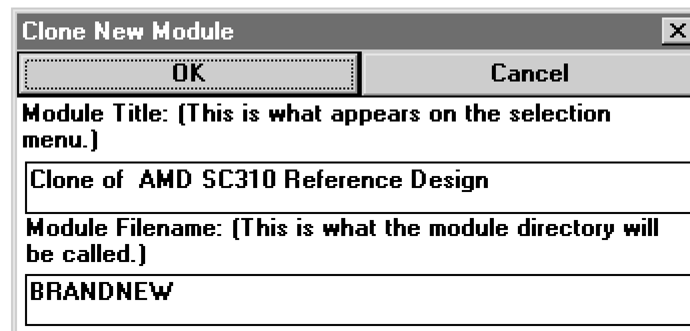


Figure 6.2. Module cloning.

Enter an 8 character module name in the lower text box. This file name will be used to name the sub-directory under the BOARDS directory, as well as the names of any .ASM or .INC files that will be copied from the original module. You should enter a unique directory identifier.

Finally, click on **[OK]** at the top of the Clone New Module dialog box. BIOStart will generate and execute a batch file that will clone the modules. A `File not found` message is normal if there are no .ASM files for BIOStart to clone, so ignore that message. If there are .ASM files for BIOStart to clone, the message will not appear. When the batch file is finished, press any key, and be sure the DOS box closes. BIOStart will not continue as long as the DOS box remains open (see Figure 6.3).

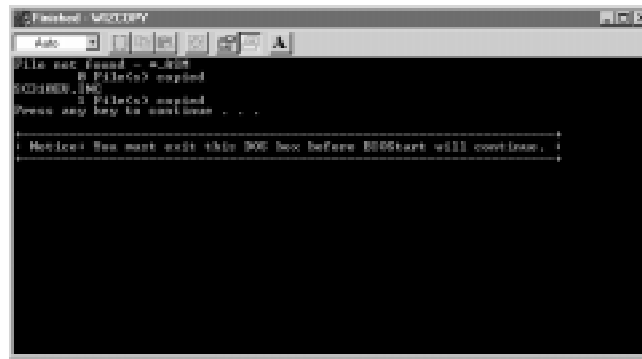


Figure 6.3. DOS box launched by BIOStart.

Select the newly cloned module from the list and press **[Next]** at the top of the window.

3. Repeat the above Board selection procedure for the Chipset and CPU modules.

4. Specify a title and filename for the BIOS adaptation. The title of the BIOS will appear at the top of the screen when the completed BIOS boots on your target hardware. The file name is used to determine the name of the project directory where the .INC file and the final .ABS file will go as well as the name of those files. To preserve compatibility with DOS and with GSMAKE, BIOStart limits the project file name to 8 characters. The default settings as well as the screen that you set them on is displayed in Figure 6.4.

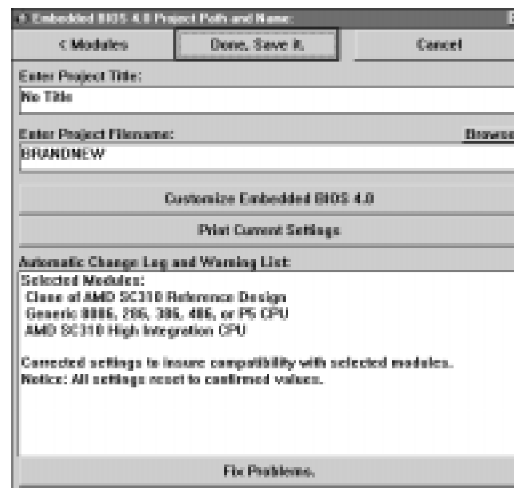


Figure 6.4. Main project control screen.

6.4 Customizing a Project

When you are actually creating a project from scratch, there are some additional steps you need to take. BIOStart cannot anticipate how the hardware will be arranged and what parts of the BIOS you will want enabled, so you will have to specify this yourself.

To begin the customization process, click on Customize Embedded BIOS 4.3. This will allow you access to all of the source code configuration parameters. You should click on the button labeled **[Basic Configuration]** (Figure 6.5) and click on each sub-category in turn, insuring that all the settings are correct for the hardware you are designing. Be aware that if you choose new

modules, or return to the module selection process, the settings will be completely reset, and you will have to go through this process again.

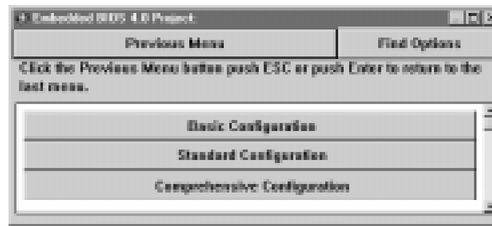


Figure 6.5. Top-level BIOS customization screen.

Also, unlike the initial BIOStart windows, the configuration windows may be re-sized or maximized. There may also be more than one open at once. The previous menu is usually hidden behind the one you are currently working with, so if you wish to see options set in the previous menu without closing the current one, you can move the current window, or re-size it. If you find the menu to be cramped, or you wish to see as many options as possible at the same time you can also maximize the window.

The **[Find Options]** button allows you to open a custom menu with all the options relating to a specific topic. If you are searching for one of the configuration options listed in this manual, you can enter it as well, and BIOStart will display its equivalent data field in the menu (see Figure 6.6). You could, for example, enter **FLASH**, and only the top menu item in Figure 6.6 would be displayed.

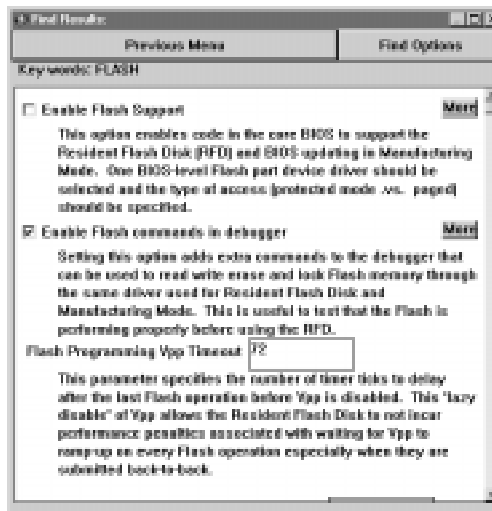


Figure 6.6. Special screen produced with Find command.

Also shown in Figure 6.6 are several **[More]** buttons. These buttons allow access to more configurations options that are related to that option.

Clicking on the **[Previous Menu]** button will take you back to the last menu. If you move all of the top menus aside, until you see the bottom menu, you could click on **[Previous Menu]** and BIOStart would take you back to a screen similar to the one in Figure 6.4.

As you set these values, BIOStart may generate warnings. If you really want to try using the feature BIOStart warns you about, you can enable it anyway. The BIOS may or may not work. You may also need to modify the source code for that feature to work. BIOStart will display all

options that generate a warning in the Automatic Change Log and Warning List shown in Figure 6.7. If any option was changed by BIOStart in an attempt to repair potential problems, this will be listed there as well. The changes are displayed in the order they were made. You can clear this list, as well as resetting possible problem values, by clicking on the **[Fix Problems]** button located directly bellow the list.

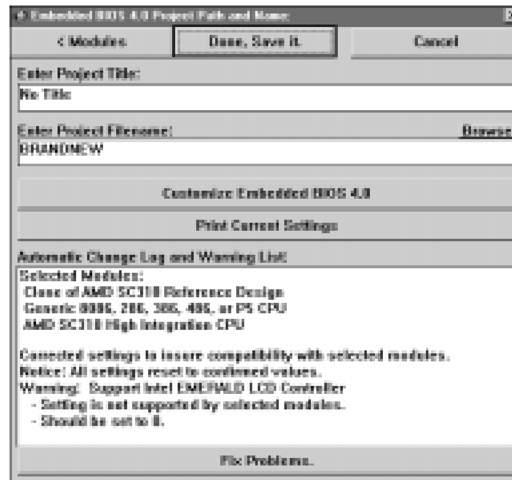


Figure 6.7. Main project control screen with logged conflicts.

At the top of the list, the selected modules are also displayed, thereby insuring that you always know which modules are being used with this project.

6.5 Printing Project Customization Settings

The **[Print Settings]** button is used primarily for debugging purposes. It prints out every EMBEDDED BIOS 4.3 option, and what BIOStart thinks it is set to. This can take a fair amount of paper, and is intended for tech support. However, it can be used to double check your work, as a means of insuring that you know what all the options are set to. It also provides a hard copy of the option settings.

The **[Browse]** button allows you to see what other projects have been created, and select one of them as the name of this project. Be aware that if you chose to save the project after having done so, the old project will be overwritten, and its .INC file will be lost. BIOStart will warn you if you attempt to overwrite a pre-existing project. Note that changing the Filename does NOT save the file. You must still click on the **[Done, Save it]** button at the top of the window.

6.6 Saving the Project and Settings

When you have finished modifying the options, and you wish to try and build the project, click on the **[Done, Save it]** button at the top of the Window. If the project name is unique, and the project has not been saved before, BIOStart will create a new project directory under the Projects directory in the source code tree with the same name as the project filename. It will also create a new .INC file.

<u>Do not click on the **[Cancel]** button unless you wish to lose all the changes you have made</u>. The **[Cancel]** button will not delete cloned modules from the disk, so don't worry about losing that kind of change; the project simply won't be saved.

6.7 Building the Project

Once you have created a project, you can use BIOStart to build it. You do this by clicking on the [Build a current Embedded BIOS 4.3 Project].

Then select a project from the list and click on [OK]. This list is shown in Figure 6.7, and is similar to the list displayed when you click on [Edit a current Embedded BIOS 4.3 Project] or [Browse] (Figure 6.7). Just click on the project you wish to build or type its name.

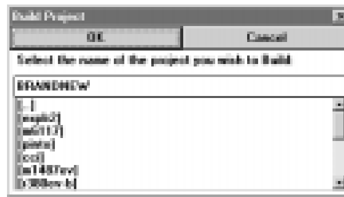


Figure 6.8. Selection of project to be built.

Next BIOStart will ask you if you wish to delete the old object files. This is a good idea if you have built the project before and you wish to insure that all the source code files will be re-compiled to reflect the new settings. BIOStart will then generate and execute a batch file that will run GSMAKE to build your project.

When the batch file has finished, be sure the DOS box opened for the batch file is closed. BIOStart will not continue until it is closed, so that the user can inspect the final results of the build and observe any errors that occurred when invoking the assembler, linker, GSMAKE, or other tools.

Once the project has been built, and BIOStart is certain the DOS box has been closed, BIOStart will display a message acknowledging that the build has been completed.

6.8 Patching Binary System BIOS Files

Once a binary (.ABS) file has been built, BIOStart can be used to directly modify that file. Binary configurable options are modified by the same process standard options are modified. You click on [Edit a current Embedded BIOS 4.3 Project], select the project from the list, and BIOStart loads the project. If there is a .ABS file, BIOStart also loads the binary configuration portion of that file.

You then click on the [Customize Embedded BIOS 4.3] button shown in the opening screen shot at the beginning of this Chapter. Find the options you wish to modify. If they are binary configurable, then BIOStart will tell you (see Figure 6.9). All binary configurable options have the text “This is a Binary configurable option” added to the bottom of its help field.

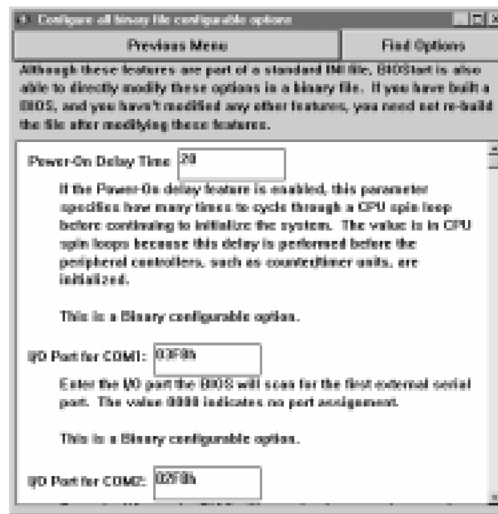


Figure 6.9. Configuration of Binary-Only parameters.

If you are an OEM, and you wish to release a binary patch only version of BIOStart, you will need to make modifications to the .WIZ files located in TOOLS subdirectory of your source code tree. Contact General Software for information describing the format and language used in these files.

6.9 Upgrading BIOStart

Most upgrades to BIOStart will be to the .WIZ files that control it. You can obtain the latest version of the .WIZ files by contacting General Software. As time goes on, new .INC and new .WIZ files will be produced that will fix potential configuration problems. Just copy the new files over the old ones. Do not attempt to modify these files yourself unless you know how they work. They are integral to BIOStart, and improper changes to them may damage the interface. Instructions on customizing the BIOStart interface can be obtained from General Software, if it becomes necessary for your project.

Chapter 7

BIOS BUILD OPTIONS

This chapter presents all of the BIOS source-level build options, which can be configured in the *project file*, either with BIOStart or with a text editor.

The defaults for these configuration options, parameters, and in some cases tables, are defined in the INC\OPTIONS.INC and INC\CONFIG.INC files. **Do not modify these files**; instead create a project file of your own, and modify it by copying lines from these other files to your project file, and then change the copied lines. Editing INC\OPTIONS.INC and INC\CONFIG.INC will make it difficult for you to upgrade the core BIOS software to new releases, and can make debugging new adaptations very diffcult. Please note: we cannot support customers who modify these files.

You can imagine that it would be quite difficult to scan through over 100,000 lines of source code and make changes to it just to be sure that you have made all the changes that are necessary to make the BIOS work on your target. Fortunately, this is not necessary with EMBEDDED BIOS.

Configuration of the BIOS takes five steps:

1. Create a project file to define the configuration.

2. Add options to the project file that override defaults found in INC\OPTIONS.INC and INC\CONFIG.INC. Do not edit INC\OPTIONS.INC and INC\CONFIG.INC directly. Three important options specify the CPM, CSPM, and BPM for the build.

3. Supply a CPU Personality Module (CPM) if necessary (see Chapter 19).

4. Supply a Chipset Personality Module (CSPM) if necessary (see Chapter 20).

5. Supply a Board Personality Module (BPM) if necessary (see Chapter 21).

This chapter describes the options and parameters that may be specified in the project file. Please take some time to review these options. General Software has already set the defaults to standard values that make sense for IBM PC/AT-compatible systems.

7.1 Options Found in OPTIONS.INC

This section explains the purpose of the options defined in the `INC\OPTIONS.INC` file. Remember, do not modify `INC\OPTIONS.INC` directly. Instead, copy the lines you want to change from this file into your project file, and change them in the project file.

Note that some configuration options are closely related. Turning on the **OPTION_SUPPORT_SETUP** option, for example, makes the build sensitive to the **OPTION_SETUP_CUSTOM** and **OPTION_SETUP_DIAGNOSTICS** options to enable specific components of Setup in the system.

7.1.1 BIOS_MAJOR_VERSION Constant

The **BIOS_MAJOR_VERSION** constant is set by General Software to identify the release. Do not modify this constant.

Values:

4 - Indicates EMBEDDED BIOS 4.x architecture.

Related Parameters:

BIOS_MINOR_VERSION.

7.1.2 BIOS_MINOR_VERSION Constant

The **BIOS_MINOR_VERSION** constant is set by General Software to identify the release. Do not modify this constant.

Values:

3 - Indicates EMBEDDED BIOS 4.3 architecture.

Related Parameters:

BIOS_MAJOR_VERSION.

7.1.3 OPTION_BIOS_KBSIZE Option

The **OPTION_BIOS_KBSIZE** option determines the size of the BIOS image itself. The highest value for this option is 256; the lowest is determined by how many features and options are enabled in the BIOS build.

All builds of the core BIOS create a file called `BIOS.ABS` that is a multiple of 64KB in size. The actual size of the file produced by the build is rounded up to the next 64KB based on the value of this parameter. This, if 29, 48, 63, or 64 were specified, the build would produce a 64KB file. Similarly, if 65, 100, 127, or 128 were specified, the build would produce a 128KB file. The

build pads the bottom portion of each 64KB "group" within the BIOS image with FFh, allowing other software to be merged-in with a BIOS build in a PROM programmer. By increasing this parameter to 64, no filler will be generated for the top 64KB group. By reducing the parameter to lower values, such as 32, 20, or less, it is possible to pack the BIOS into a smaller area in the top of BIOS.ABS, and therefore the BIOS may require a smaller area of ROM to run from.

A suggested value to start with is 64 for simpler targets with fewer features enabled, or 128 for higher-end targets or those builds with lots of features enabled. Then, the value should be reduced after building a BIOS with the intended features.

After the BIOS is built, take a look at the file called PROJECTS*myproj*\BIOSFILL.INC; this file will contain symbol definitions used by the 16-bit BIOS build to create padding in the system. In the sample BIOSFILL.INC file below, note that the BIOS has two full segments for a 128KB build; the E000h and F000h segment. The F000h segment is commonly called the Last segment. Segment E000h has 35935 bytes free that have been padded by this build. The Last segment at F000h has only 3920 bytes free that have been padded. The total amount of space unused in the 128KB raw file is 39855, as indicated by a comment at the bottom of the file.

Clearly, this file indicates that it does not use all of the first segment in the BIOS image at E000h. The size of this segment could be reduced by 35935 bytes, but must be done on a 1KB basis. Therefore, **OPTION_BIOS_KBSIZE** might be set to (128-(35935/1024))=(128-35)=93 to achieve a perfect footprint for this build. This would pad the beginning of the E000h segment with FFh bytes, making it available for other uses.

```
;***    BIOSFILL.INC -- Embedded BIOS Padding Include File.
;
;1.     Functional Description.
;       This include file is built with the BIOSMAP.EXE utility to define
;       padding bytes that position the symbol OFFSET_FF00 to offset ff00h.
;       Doing this allows us to correctly position the bootstrap code within
;       the assembly, because the assembly ORG statement can't do this.
;
;2.     Modification History.
;       S. E. Jones     92/06/02.       Manufactured by BIOSMAP.
;       S. E. Jones     93/09/05.       Moved to C 8.0.
;       K. C. Taylor    98/08/12.       Upgraded for multi-segment.
;
;3.     NOTICE: Copyright (C) 1992-2000 General Software, Inc.

SEGE000_FILL_SIZE = 35935 ; Pad E000 to 64k.
LASTSEG_FILL_SIZE = 3920 ; locate OFFSET_FF00 to ff00h.

;       Total unused space: 39855
```

It is necessary to specify which components of the BIOS should be moved into segments C000h, D000h, or E000h from the standard F000h segment, in order to make use of BIOS build sizes greater than 64. This is accomplished with the **RELOCATE_FEATURE** table in the project file.

Values:

n - A number between 1 and 256, inclusive. Start with 64 or 128.

Related Parameters:

RELOCATE_FEATURE – Moves features to segments E000h, D000h, and C000h.

7.1.4 OPTION_SUPPORT_PCODE Option

The **OPTION_SUPPORT_PCODE** option enables or disables code that implements the Pseudo Code (PCODE) Interpreter in the core BIOS. This option causes some common sequences of machine code in the BIOS to be converted to special Intel opcodes which generate an invalid instruction trap. The PCODE emulator handles these exceptions on the fly and executes the intended operation. The result is a small space savings at some expense in compatibility and performance.

Do not enable this option and the corresponding **OPTION_SUPPORT_PCODE** option in the Embedded DOS-ROM build. If you are running application software or other system software that uses a similar technique, then the EMBEDDED BIOS PCODE interpreter may not be given the chance to properly handle the exceptions, which could lead to wrong results. It is important to use this option only to save space in completely closed systems where all the software to be run in the system is known and tested in advance.

Values:

1 - Enable PCODE interpreter, saving code space wherever possible.
0 - Disable PCODE interpreter.

Related Parameters:

None.

7.1.5 OPTION_SUPPORT_SETUP Option

The **OPTION_SUPPORT_SETUP** option enables or disables code that implements the SETUP menu and related screens.

Enabling this option does not enable specific screens in SETUP. These must be enabled with the **OPTION_SETUP_xxx** parameters.

SETUP works with or without an actual CMOS part. If no CMOS is present in a design, then the factory default values for SETUP are used, as built-up from build parameters.

SETUP can perform other things besides CMOS configuration. For example, it allows access to the Integrated BIOS Debugger, Manufacturing Mode, Standard Diagnostics suite, Power Management functions, and Flash disk formatter.

Values:

1 - Enable SETUP menu.
0 - Disable SETUP menu.

Related Parameters:

OPTION_SETUP_DEMO - Enable GS demo Setup screen.
OPTION_SETUP_CUSTOM - Enable chipset Setup screen.
OPTION_SETUP_PASSWORD - Enable password Setup screen.
OPTION_SETUP_DIAGNOSTICS - Enable user diagnostics Setup screen.
OPTION_SETUP_DEBUGGER - Enable debugger entry in Setup menu.

OPTION_SETUP_IDE - Enable OEM IDE utilities Setup screen.
OPTION_SETUP_SHADOWCACHE - Enable ROM shadowing Setup screen.
OPTION_SETUP_PWR_FEATURES - Enable power mgt features Setup screen.
OPTION_SETUP_PWR_TIMEOUTS - Enable power mgt timeouts Setup screen.
OPTION_SETUP_MFGMODE - Enable Manufacturing Mode Setup access.
OPTION_SETUP_RAMDISK - Enable RAM disk formatting Setup screen.
OPTION_SETUP_RFDDISK - Enable low-level RFD formatting Setup screen.

7.1.6 OPTION_SUPPORT_CONFIGBOX Option

The **OPTION_SUPPORT_CONFIGBOX** option enables or disables code that implements the configuration box displayed during POST right before booting the operating system.

Values:

1 - Enable configuration box.
0 - Disable configuration box.

Related Parameters:

CONFIG_CFGBOX_MONO_ATTRIB - Monochrome attribute used for box.
CONFIG_CFGBOX_COLOR_ATTRIB - Color attribute used for box.

7.1.7 OPTION_SUPPORT_POSTCODES Option

The **OPTION_SUPPORT_POSTCODES** option enables or disables code in POST that writes progress codes to the manufacturing port (normally, I/O port 80h). This is a useful feature that is used during development to debug the hardware, and is also used during Q/A of production units.

The I/O port address can be changed by changing the **CONFIG_POST_PROGRESS_PORT** parameter, for systems that do not have a progress port in the traditional sense, or that have a progress port that is not wired to the default address.

Sometimes it may be useful to employ another register, such as a scratch register on a 16550 UART, as the POST progress port. If a read/write port such as this (2ffh, 3ffh, etc.) is selected, then the Manufacturing Mode can be used to remotely determine what the last POST code was.

Some chipsets, such as the RadiSys R380, provide support for "snooping" I/O bus accesses. The EMBEDDED BIOS R380EX Chipset Personality Module can be programmed to route these POST codes to a 7-segment display, for example.

Values:

1 - Enable POST codes written to manufacturing port.
0 - Disable POST codes written to manufacturing port.

Related Parameters:

CONFIG_POST_PROGRESS_PORT - Select I/O port for POST codes.

7.1.8 OPTION_SUPPORT_POSTCODES_COM Option

The **OPTION_SUPPORT_POSTCODES_COM** option enables or disables code in POST that writes a special set of progress codes over an RS232 link via an 8250-compatible UART. This allows debugging of POST on targets that do not have an I/O port 80h monitor, or when no logic analyzer is available to record the *sequence* of POST activities that occurs.

The actual output is routed through the BPM's **BoardPostCodeCom** routine, so that the OEM can use any hardware to display or capture the codes. The device is initialized by the core BIOS by calling the BPM's **PostCodeComInit** routine, so the OEM can initialize any custom hardware required by this function. The default BPM routines support standard 8250-compatible UARTs, for the convenience of most designers who already have those parts on their targets.

The base I/O port address of the UART can be changed by changing the **CONFIG_POST_PROGRESS_COM** parameter. Values such as 3f8h and 2f8h can be used to access standard UART addresses, but alternates can also be chosen.

By default, output over this port occurs at 9600 baud, no parity, and 1 stop bit. The baud rate can be adjusted with the **CONFIG_POST_PROGRESS_BAUD** parameter.

COM port progress codes are simple alphanumeric characters that are generated with the **POSTCODECOM** macro calls in module `SYSTEM\POST.ASM`. If you need to debug other modules, simply add **POSTCODECOM** statements as needed; keeping in mind that (a) the **POSTCODECOM** macro destroys some registers (see `INC\MACROS.INC`) and (b) the **POSTCODECOM** macro cannot be called until after the UART has been made available via Super I/O programming or Chipset programming to enable the UART.

Values:

1 - Enable alphanumeric progress codes written to COM port.
0 - Disable alphanumeric progress codes written to COM port.

Related Parameters:

CONFIG_POST_PROGRESS_COM - Select base I/O port for UART.
CONFIG_POST_PROGRESS_BAUD - Select baud rate for UART.

7.1.9 OPTION_SUPPORT_MFGCODES Option

The **OPTION_SUPPORT_MFGCODES** option enables or disables code in POST that copies incoming Manufacturing Mode command codes to an I/O port so that it can be viewed on a 7-segment hex LED display.

Commonly, this is enabled during debugging of a system, and the standard I/O port 80h is used for both this purpose and for the purpose of displaying BIOS POST codes.

Values:

1 - Enable Manufacturing Mode progress codes.
0 - Disable Manufacturing Mode progress codes.

Related Parameters:

CONFIG_MFG_PROGRESS_PORT - I/O port to which the command codes will be written. Usually, this is 80h, but can be modified to support any I/O port.

7.1.10 OPTION_SUPPORT_POSTMSGS Option

The **OPTION_SUPPORT_POSTMSGS** option enables or disables code in POST that displays progress or error messages during system initialization. Desktop PC targets should have this option enabled, while embedded targets without a display should have this option disabled during production.

It is possible for embedded hardware that doesn't have a display to route POST messages over a serial port. This is not the same thing as routing POST *codes* over a COM port; here I/O refers to actual messages such as the sign-on banner, memory count-up display, and so on.

Keyboard input for prompts during POST is also conditionalized with this option. When the option is enabled, the prompts are enabled. When the option is disabled, the code continues as though the operator supplied the answer most likely to allow POST to continue to boot the operating system.

To redirect POST's messages over an RS232 line, choose a serial port assignment for **CONFIG_CON_REDIR_STD** other than 0, where its value indicates the COM port number.

Values:

1 - Enable POST messages.
0 - Disable POST messages.

Related Parameters:

CONFIG_CON_REDIR_STD - Standard video I/O redirection.

7.1.11 OPTION_SUPPORT_POWERON_DELAY Option

The **OPTION_SUPPORT_POWERON_DELAY** option enables or disables code in POST that pauses after a power-on condition to wait for the power supply to come up to the required voltage. While the CPU may be executing properly, peripherals being programmed by POST may require extra time immediately after power-on to reset and come up to operating condition.

This can especially be a problem with targets that have lots of components to power-up, that are powered by light-duty power supplies. If it seems that it takes a couple of pushes of the RESET button on a board to get it to boot, inadequate supply may be the problem.

Often the main component that does not receive enough power quickly enough to begin servicing CPU requests is the 8042 keyboard controller. If you have an 8042-compatible keyboard controller, this parameter should be enabled, and you should set the **CONFIG_POWER_ON_DELAY** parameter to something around 20. This parameter can be adjusted after the entire BIOS is running on the *final hardware* with its *final power supply*.

Values:

1 - Enable power-on delay.
0 - Disable power-on delay.

Related Parameters:

CONFIG POWER_ON_DELAY - Delay length (in units of "CPU loops").

7.1.12 OPTION_SUPPORT_DEBUGGER Option

The **OPTION_SUPPORT_DEBUGGER** option enables or disables code in the BIOS that implements the integrated BIOS debugger. If this option is enabled, then the BIOS will automatically route CPU traps and faults to the debugger.

If you want the debugger to intercept the CTRL-SHIFT key chord as a request to enter the debugger asynchronously, then you must explicitly set **OPTION_DEBUG_HOTKEY** to 1.

You may also want to enable **OPTION_DEBUG_FLASH** if you will be using the EFL, RFL, WFL, SFL, or UFL debugger commands to manipulate Flash devices interactively.

The **OPTION_DEBUG_WATCHINT** option can be enabled to support software interrupt watchpoints at all of the common BIOS service routines when traces of BIOS interrupt requests are needed to debug a new operating system.

The **OPTION_DEBUG_NMI** option can be enabled to allow the NMI interrupt to break into the debugger with a hardware request. This is useful when supporting breakout switches on ISA designs.

The debugger's PCMCIA CIS decoding commands are enabled with the **OPTION_DEBUG_PCMCIA** option. This allows debugging of custom BIOS code to enable certain PCMCIA cards for embedded applications.

The **OPTION_DEBUG_ASSEMBLY** option controls the support for disassembly of CPU instructions in the debugger. Normally, this is enabled, but it can be disabled to save space.

The **OPTION_DEBUG_EDOSROM** option enables a special back-door debugging service that permits internal debugging statements in Embedded DOS-ROM to be conditionally executed and their output routed through the BIOS. This is normally used at General Software for debugging system software that runs with Embedded DOS-ROM.

The **OPTION_DEBUG_CHIPSET** option controls the support for reading and writing chipset-specific registers with CSR and CSW commands. Normally, this is enabled, but it can be disabled to save space.

Values:

1 - Enable integrated BIOS debugger.
0 - Disable integrated BIOS debugger.

Related Parameters:

OPTION_DEBUG_HOTKEY - Enable Ctl-Left-Shift debugger entry.
OPTION_DEBUG_FLASH - Enable Flash debugging commands.
OPTION_DEBUG_WATCHINT - Enable BIOS interrupt watchpoints.
OPTION_DEBUG_NMI - Enable NMI debugger entry.
OPTION_DEBUG_PCMCIA - Enable PCMCIA debugger commands.
OPTION_DEBUG_ASSEMBLY - Enable opcode disassembler in debugger.
OPTION_DEBUG_EDOSROM - Enable Embedded DOS-ROM backdoor I/O.
OPTION_DEBUG_CHIPSET - Enable chipset read/write register commands.

7.1.13 OPTION_SUPPORT_SHADOW Option

The **OPTION_SUPPORT_SHADOW** option enables or disables code in the BIOS that supports the shadowing of slow ROMs with fast DRAM or SRAM.

The support for ROM shadowing must be provided by the Board Personality Module (BPM) or Chipset Personality Module (CSPM) for this option to function properly. Additionally, the Shadowing Configuration Setup screen must be enabled so that the user can specify which areas to shadow.

If you intend for the system to support PCI properly, then this option must be enabled, since PCI uses shadow RAM to map PCI ROM extensions.

Values:

1 - Enable ROM shadowing.
0 - Disable ROM shadowing.

Related Parameters:

OPTION_SUPPORT_CHIPSET - Enable CSPM code to provide shadowing support. The actual shadowing function is implemented in the Chipset Personality Module, and the CSPM code is only enabled by this option.

OPTION_HARDERR_DISSHADOW - Cause critical POST error if shadow disabling doesn’t work.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing in CMOS.
OPTION_CMOS_SHADOW_C000 - Enable shadowing of segment C000h.
OPTION_CMOS_SHADOW_C400 - Enable shadowing of segment C400h.
OPTION_CMOS_SHADOW_C800 - Enable shadowing of segment C800h.
OPTION_CMOS_SHADOW_CC00 - Enable shadowing of segment CC00h.
OPTION_CMOS_SHADOW_D000 - Enable shadowing of segment D000h.
OPTION_CMOS_SHADOW_D400 - Enable shadowing of segment D400h.
OPTION_CMOS_SHADOW_D800 - Enable shadowing of segment D800h.
OPTION_CMOS_SHADOW_DC00 - Enable shadowing of segment DC00h.
OPTION_CMOS_SHADOW_E000 - Enable shadowing of segment E000h.
OPTION_CMOS_SHADOW_E400 - Enable shadowing of segment E400h.
OPTION_CMOS_SHADOW_E800 - Enable shadowing of segment E800h.
OPTION_CMOS_SHADOW_EC00 - Enable shadowing of segment EC00h.
OPTION_CMOS_SHADOW_F000 - Enable shadowing of segment F000h.

7.1.14 OPTION_SUPPORT_CACHE Option

The **OPTION_SUPPORT_CACHE** option enables or disables code in the BIOS that supports a level 2 (L2) cache that is external to the processor.

This option does not specify how the cache is supported, but does enable the code to initialize the cache and route requests to control it during normal system operation.

To control the L2 cache with the Chipset Personality Module, use the **OPTION_CACHE_CHIPSET** option. To control the cache with special hardware on the board itself, use **OPTION_CACHE_BOARD**.

The above methods are L2 cache controls. The level 1 (L1) cache controller, if present, resides in the CPU. This feature is enabled for CPUs with internal caches by enabling **OPTION_CACHE_CPU**. This option may be used in conjunction with one of the L2 cache enablers. The L1 cache control logic is not affected by the **OPTION_SUPPORT_CACHE** option.

Values:

1 - Enable L2 cache controls.
0 - Disable L2 cache controls.

Related Parameters:

OPTION_SUPPORT_CHIPSET - Enable Chipset Personality Module.

OPTION_CACHE_CHIPSET - Enable chipset cache controls.

OPTION_CACHE_BOARD - Enable Board Personality Module cache controls.

OPTION_CACHE_CPU - Enable L1 cache, independent of this L2 support.

7.1.15 OPTION_SUPPORT_8250 Option

The **OPTION_SUPPORT_8250** option enables or disables code in the BIOS that supports PC-compatible 8250, 8251, 16450, or 16550 UARTs in the serial port BIOS.

This option provides generic PC-compatible UART support, even when the UARTs are actually PC-compatible UARTs implemented in a chipset or on-board a high-integration CPU.

Values:

1 - Enable 8250 UART support.
0 - Disable 8250 UART support.

Related Parameters:

OPTION_SERIAL_8250 - Enable 8250 serial ports.

OPTION_SERIAL_CPU - Enable on-board CPU serial ports.

OPTION_SERIAL_WAIT_DSR - Wait for DSR before receiving.

OPTION_SERIAL_WAIT_DSRCTS - Wait for DSR & CTS before transmitting.

OPTION_SUPPORT_8250 - Enable 8250-compatible UARTs.

OPTION_SERIAL_FIFO - Enable 8250-compatible FIFO.

OPTION_SERIAL_HALT - Enable HLT in spin-wait for character available.

OPTION_SERIAL_9600_BAUD - Force all INT 14h initialization requests to always initialize the UART at 9600 baud, no parity, and 1 stop bit.

CONFIG_SERIAL_TIMEOUT - COM port timeout in seconds. This timeout is used in INT 14h requests.

COM1_BASE - I/O port address for COM1 UART.
COM2_BASE - I/O port address for COM2 UART.
COM3_BASE - I/O port address for COM3 UART.
COM4_BASE - I/O port address for COM4 UART.

COM1_INIT - Initialization setting for COM1 UART.
COM2_INIT - Initialization setting for COM2 UART.
COM3_INIT - Initialization setting for COM3 UART.
COM4_INIT - Initialization setting for COM4 UART.

7.1.16 OPTION_SUPPORT_8254 Option

The **OPTION_SUPPORT_8254** option enables or disables code in the BIOS that supports the PC/AT compatible 8253/ 8254 programmable interval timer chip. This part contains three timers that are used by the BIOS to maintain the time of day, to manage DRAM refresh when used in conjunction with an 8237A, and to beep the speaker.

Some high-integration CPUs and chipsets provide this timer system, and usually, the replicas operate the same as the original 8253/8254. Therefore, even though the timer silicon may reside in the CPU or the chipset, the **OPTION_SUPPORT_8254** is enabled, and **OPTION_TIMER_8254** is enabled instead of enabling **OPTION_TIMER_CPU**.

The only time when **OPTION_TIMER_CPU** is used is on targets based on nonstandard processors such as the Intel 80C186-EC (that CPU's timer is not compatible with the 8254).

Values:

1 - Enable 8254 timer.
0 - Disable 8254 timer.

Related Parameters:

OPTION_TIMER_8254 - Use 8254 as primary timer source. If you have an 8254, this should be *enabled* except for unusual circumstances.

OPTION_TIMER_CPU - Use CPU integrated timer as the primary timer source. If you have an 8254, this should be *disabled* except for unusual circumstances.

OPTION_TIMER_BOARD - Use timer hardware on the board itself as the primary timer source. If you have an 8254, this should be *disabled* except for unusual circumstances.

7.1.17 OPTION_SUPPORT_8255 Option

The **OPTION_SUPPORT_8255** option enables or disables code in the BIOS that supports the PC and PC/XT compatible 8255 peripheral interface controller, used to interface with the PC configuration switches, the NMI controls, the PC speaker, and the PC keyboard.

There is a distinction between raw 8255 support used to control the PC/XT keyboard and PORT B, which is not present on all designs. If the target has a PC or PC/XT keyboard controller (i.e., an 8255), then you must enable **OPTION_SUPPORT_8255**.

As a separate consideration, whether or not your target has an 8255 keyboard controller interface, you should enable **OPTION_SUPPORT_PORT_B** if your target supports PORT B. Almost all AT-class platforms have a PORT B, and most PC/XT-class platforms have this port as well.

Values:

1 - Enable 8255 peripheral interface controller.
0 - Disable 8255 peripheral interface controller.

Related Parameters:

OPTION_SUPPORT_PORT_B - Enable PORT B support.

OPTION_SUPPORT_KEYBOARD - Enable INT 16h keyboard support.

OPTION_KEYBOARD_PCAT - Enable PC, PC/XT, or PC/AT keyboard support logic.

7.1.18 OPTION_SUPPORT_PORT_B Option

The **OPTION_SUPPORT_PORT_B** option enables or disables code in the BIOS that supports I/O PORT B. This port can be implemented by the 8042 keyboard controller in PC/AT-class machines, the 8255 peripheral controller in PC/XT-class machines, the CPU itself in V51-class machines, or in many cases, the chipset.

Port B is actually the name for the byte-wide I/O port at address 61h. Its bits are defined as follows:

Bit 7 r = 1 RAM parity error.

w	= 1	Clear IRQ timer latch (MCA only).
6 r	= 1	I/O parity error.
5 r	= x	Output of Timer 2 (8254 or equivalent).
4 r	= x	Refresh request clock divided by 2.
3 r/w	= 0	Enable I/O parity check.
2 r/w	= 0	Enable RAM parity check.
1 r/w	= 1	Speaker data enabled.
0 r/w	= 1	Gate Timer 2 enabled.

As can be seen, PORT B is tied to a number of features in PC/XT and PC/AT-class designs. PORT B is involved in RAM parity support, DRAM refresh detection, and speaker control.

Values:

1 - Enable PORT B support.
0 - Disable PORT B support.

Related Parameters:

OPTION_SUPPORT_8255 - Enable PC & PC/XT keyboard controller support (PORT B can be implemented with an 8255).

OPTION_SUPPORT_8042 - Enable PC/AT keyboard controller support (PORT B can be implemented with an 8042).

7.1.19 OPTION_SUPPORT_8259 Option

The **OPTION_SUPPORT_8259** option enables or disables code in the BIOS that supports the PC compatible 8259 programmable interrupt controller.

PC/AT systems have two 8259s, so this option and the **OPTION_SUPPORT_8259_2** should be enabled for ISA-type targets.

Your target might not have a real, discrete, 8259 interrupt controller. Instead, the same functionality could be implemented in the chipset, or may reside in the CPU itself.

When implemented in the chipset, interrupt controllers are almost always identical with the 8259, and so the BIOS should be told that an 8259 (or two as the case may be) exist(s).

When the interrupt controller is implemented in a high-integration CPU, it may or may not emulate an 8259. If it does, then **OPTION_SUPPORT_8259** should be enabled, and then no code needs to be placed in the CPU Personality Module. If the interrupt controller is not 8259-compatible, then **OPTION_SUPPORT_8259** should be disabled, along with **OPTION_INT_8259**, and then **OPTION_INT_CPU** should be enabled.

In the rare case where an external 8259 interrupt controller is cascaded to a CPU interrupt controller, then the following three parameters should be enabled: **OPTION_SUPPORT_8259**, **OPTION_INT_8259**, and **OPTION_CPU_8259**.

Values:

1 - Enable primary 8259 interrupt controller.
0 - Disable primary 8259 interrupt controller.

Related Parameters:

OPTION_SUPPORT_8259_2 - Enable secondary interrupt controller.

OPTION_INT_8259 - Use 8259's for BIOS interrupt control. If **OPTION_SUPPORT_8259** is enabled, this should also be enabled, except in rare circumstances.

OPTION_INT_CPU - Use CPU integrated interrupt controller (such as the CIC on an Intel 80C186-EC). This can be used in conjunction with **OPTION_INT_8259** and **OPTION_SUPPORT_8259** in the event that the external 8259 is cascaded to the CPU CIC.

OPTION_INT_BOARD - Use external hardware on the board itself to manage interrupts.

7.1.20 OPTION_SUPPORT_8259_2 Option

The **OPTION_SUPPORT_8259_2** option enables or disables code in the BIOS that supports the PC/AT compatible secondary 8259 programmable interrupt controller. PC/AT systems have two 8259s, so this option must be enabled for ISA-type targets.

See additional comments about interrupt controller options with the **OPTION_SUPPORT_8259** option.

Values:

1 - Enable secondary 8259 interrupt controller.
0 - Disable secondary 8259 interrupt controller.

Related Parameters:

OPTION_SUPPORT_8259 - Enable primary interrupt controller.

OPTION_INT_8259 - Use 8259's for BIOS interrupt control. If **OPTION_SUPPORT_8259_2** is enabled, this should also be enabled, except in rare circumstances.

OPTION_INT_CPU - Use CPU integrated interrupt controller (such as the CIC on an Intel 80C186-EC). This can be used in conjunction with **OPTION_INT_8259**, **OPTION_SUPPORT_8259**, and **OPTION_SUPPORT_8259_2** in the event that there are *two* external 8259s cascaded to the CPU CIC.

OPTION_INT_BOARD - Use external hardware on the board itself to manage interrupts.

7.1.21 OPTION_SUPPORT_8237 Option

The **OPTION_SUPPORT_8237** option enables or disables code in the BIOS that supports the PC compatible primary 8237A DMA controller. PC/AT systems have two 8237As, so the **OPTION_SUPPORT_8237_2** option must be enabled for ISA-type targets.

In many cases, high-integration CPUs will contain 8237A-compatible DMA controller(s). Chipsets may also contain these components. If either the CPU or the chipset contains an 8237A, then **OPTION_SUPPORT_8237** should be enabled, so that no code in the CPU or Chipset Personality Modules need be written. Note this is the case for the Intel 80C386-EX and AMD SC300- and SC400-series Elan CPUs.

Values:

1 - Enable primary 8237A DMA controller.
0 - Disable primary 8237A DMA controller.

Related Parameters:

OPTION_SUPPORT_8237_2 - Enable secondary interrupt controller.

OPTION_DMA_8237 - Use 8237A in core BIOS functionality. If you have enabled **OPTION_SUPPORT_8237**, then this parameter should also be enabled, except in rare circumstances.

OPTION_DMA_CPU - Use the CPU's integrated DMA controller in core BIOS functionality. If you have enabled **OPTION_SUPPORT_8237**, then this parameter should be disabled, except in rare circumstances.

OPTION_DMA_BOARD - Use external DMA hardware on the board to handle DMA requests from the BIOS. This is sometimes necessary if the DMA controllers in the CPU are nonstandard, and some reordering of the DMA channel numbers are necessary (as might be the case with a RadiSys R380 and Intel 386-EX combination). If you have enabled **OPTION_SUPPORT_8237**, then this parameter should be disabled, except in rare circumstances.

7.1.22 OPTION_SUPPORT_8237_2 Option

The **OPTION_SUPPORT_8237_2** option enables or disables code in the BIOS that supports the PC/AT compatible secondary 8237A DMA controller. PC/AT systems have two 8237A parts, so this option must be enabled for ISA-type targets.

See additional comments about DMA controller options with the **OPTION_SUPPORT_8237** option.

Values:

1 - Enable secondary 8237A DMA controller.
0 - Disable secondary 8237A DMA controller.

Related Parameters:

OPTION_SUPPORT_8237 - Enable primary interrupt controller.

OPTION_DMA_8237 - Use 8237A in core BIOS functionality. If you have enabled **OPTION_SUPPORT_8237_2**, then this parameter should also be enabled, except in rare circumstances.

OPTION_DMA_CPU - Use the CPU's integrated DMA controller in core BIOS functionality. If you have enabled **OPTION_SUPPORT_8237_2**, then this parameter should be disabled, except in rare circumstances.

OPTION_DMA_BOARD - Use external DMA hardware on the board to handle DMA requests from the BIOS. This is sometimes necessary if the DMA controllers in the CPU are nonstandard, and some reordering of the DMA channel numbers are necessary (as might be the case with a RadiSys R380 and Intel 386-EX combination). If you have enabled **OPTION_SUPPORT_8237**, then this parameter should be disabled, except in rare circumstances.

7.1.23 OPTION_SUPPORT_8042 Option

The **OPTION_SUPPORT_8042** option enables or disables code in the BIOS that supports the PC/AT compatible 8042 keyboard controller. This controller is actually a general-purpose microcontroller part that not only interacts with the keyboard, but it also provides access to cache, A20, and CPU speed controls.

Although the 8042 nomenclature is still used today, the actual 8042 microcontroller and its control program are now implemented in many different ways, including hardware state machines on high-integration CPUs, and other packages such as the 8051.

The 8042 external architecture is often emulated by chipsets, so it is important to enable this option if your chipset emulates the 8042 so that EMBEDDED BIOS will program it properly.

It is also important that the 8255 support not be enabled if the 8042 option is enabled. The 8042 emulates much of what the 8255 does, so these modules would conflict if enabled together.

If your target has an 8042, then it is likely to also have a PORT B defined. If so, you should enable **OPTION_SUPPORT_PORT_B**.

There are a few 8042 control parameters in CONFIG.INC that deal with 8042 timing issues, since it operates asynchronously to the CPU's clocking. The parameters are **CONFIG_WAIT_8042**, **CONFIG_WAIT_8042_INIT,** and **CONFIG_SETTLE_8042**.

More 8042 parameters deal with the functionality of the 8042 itself, not its timing. The parameters are **OPTION_8042_TESTP22P23**, **OPTION_8042_READPWRSTAT**, **OPTION_8042_CHECKBAT**, **OPTION_8042_PS2**, and **OPTION_8042_WAIT_BEFORE_BAT**.

Values:

1 - Enable 8042 keyboard controller support.

0 - Disable 8042 keyboard controller support.

Related Parameters:

OPTION_SUPPORT_8255 - Enable 8255 keyboard controller.

OPTION_8042_TESTP22P23 - Test 8042 ports 2.2 and 2.3 during POST.

OPTION_8042_READPWRSTAT - Read 8042 status after power on.

OPTION_8042_CHECKBAT - Fail POST if BAT code is erroneous.

OPTION_8042_PS2 - Insert appropriate delays for PS/2-compatible 8042 keyboard controller. Note this is for the controller, not the keyboard.

OPTION_8042_WAIT_BEFORE_BAT - Delay during POST right before BAT is read from 8042 to allow the keyboard extra time to boot.

CONFIG_WAIT_8042 – Retry time for 8042 to accept command.

CONFIG_WAIT_8042_INIT – Retry time for 8042 to initialize during POST.

CONFIG_SETTLE_8042– Retry time for 8042 to execute command.

7.1.24 OPTION_SUPPORT_CMOS Option

The **OPTION_SUPPORT_CMOS** option enables or disables code in the BIOS that supports the PC/AT compatible CMOS Configuration RAM in the battery-backed Real-Time Clock device.

The battery-backed CMOS RAM makes it ideal for storing system configuration data; this is how PC/AT-compatible machines maintain their state after power-down. CMOS is edited by the Setup screen system, and may be initialized to factory default values in the project file.

CMOS is not actually necessary in systems that use SETUP. Remember that the whole point of CMOS RAM is to maintain a battery-backed copy of the system's configuration. If this is gone, then the machine will be forced to use factory defaults unless SETUP is entered, and the parameters modified during that session. Thus, SETUP can be used to adjust factory defaults for one bootstrap operation. Additionally, SETUP can be used to enter Manufacturing Mode, Standard Diagnostics, and the Integrated BIOS Debugger.

If your target does not have CMOS, but you would like the system to maintain its state using another device as though it did have the equivalent CMOS RAM, review the routines in the Board Personality Module (BPM) specification in Chapter 20 to learn how to intercept internal CMOS read/write requests in the BPM and handle them in other ways.

Values:

1 - Enable CMOS RAM support.
0 - Disable CMOS RAM support.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable Setup screen system.

CONFIG_MAX_CMOS_LOCATIONS - Number of CMOS cells.

CONFIG_START_BOARD_CMOS - 1st CMOS cell assigned to board extensions.

CONFIG_START_CMOS_CACHE - 1st CMOS cell not RTC-related.

CONFIG_CMOS_INDEX - I/O port used to read/write CMOS RAM index register.

CONFIG_CMOS_DATA - I/O port used to read/write CMOS RAM data.

OPTION_CMOS_SHADOW_ENABLE - Enable shadowing in CMOS.

OPTION_CMOS_SHADOW_C000 - Enable shadowing of segment C000h.

OPTION_CMOS_SHADOW_C400 - Enable shadowing of segment C400h.

OPTION_CMOS_SHADOW_C800 - Enable shadowing of segment C800h.

OPTION_CMOS_SHADOW_CC00 - Enable shadowing of segment CC00h.

OPTION_CMOS_SHADOW_D000 - Enable shadowing of segment D000h.

OPTION_CMOS_SHADOW_D400 - Enable shadowing of segment D400h.

OPTION_CMOS_SHADOW_D800 - Enable shadowing of segment D800h.

OPTION_CMOS_SHADOW_DC00 - Enable shadowing of segment DC00h.

OPTION_CMOS_SHADOW_E000 - Enable shadowing of segment E000h.

OPTION_CMOS_SHADOW_E400 - Enable shadowing of segment E400h.

OPTION_CMOS_SHADOW_E800 - Enable shadowing of segment E800h.

OPTION_CMOS_SHADOW_EC00 - Enable shadowing of segment EC00h.

OPTION_CMOS_SHADOW_F000 - Enable shadowing of segment F000h.

OPTION_CMOS_MOUSE - Factory default for CMOS enabling support for the PS/2 mouse.

OPTION_CMOS_TEST1MB - Factory default for CMOS enabling POST's memory test above 1MB.

OPTION_CMOS_TESTCLICK - Factory default for CMOS enabling POST's speaker clicks between tested blocks during its memory tests.

OPTION_CMOS_PARITY - Factory default for CMOS POST parity enable.

OPTION_CMOS_DELETE - Factory default for CMOS POST display allowing "*press <DEL> to enter Setup*" message to appear.

OPTION_CMOS_HEXLOWER - Factory default for CMOS for lower-case hex number displays during all BIOS-level I/O through its PRINTF package.

OPTION_CMOS_F1ERROR - Factory default for CMOS enabling the prompt to press F1 to continue when soft errors occur during POST.

OPTION_CMOS_NUMLOCK - Factory default for CMOS enabling NUMLOCK key.

OPTION_CMOS_TYPEMATIC - Factory default for CMOS enabling typematic keyboard programming.

OPTION_CMOS_WEITEK - Factory default for CMOS enabling Weitek support.

OPTION_CMOS_FLOPPYSEEK - Factory default for CMOS enabling floppy seek during POST.

OPTION_CMOS_EXTCACHE - Factory default for CMOS enabling external cache.

OPTION_CMOS_INTCACHE - Factory default for CMOS enabling internal cache.

OPTION_CMOS_FASTA20 - Factory default for CMOS enabling fast A20 gate.

OPTION_CMOS_HDSEEK - Factory default for CMOS enabling hard disk seek during POST.

OPTION_CMOS_CONFIGBOX - Factory default for CMOS enabling display of configuration box after POST.

OPTION_CMOS_EXHMEMTEST - Factory default for CMOS enabling exhaustive memory tests during POST.

OPTION_CMOS_PASSWORD - Factory default for CMOS enabling password checking during POST.

OPTION_CMOS_KEYBOARD - Factory default for CMOS enabling keyboard support.

OPTION_CMOS_ROMDISK - Factory default for CMOS enabling ROM disk support.

OPTION_CMOS_SPEED - Factory default for CMOS initial CPU speed.

OPTION_CMOS_REFRESH - Factory default for CMOS DRAM refresh.

OPTION_CMOS_POWER - Factory default for CMOS power management enable.

OPTION_CMOS_ATA - Factory default for CMOS ATA support enable.

OPTION_CMOS_RFD - Factory default for CMOS RFD support enable.

OPTION_CMOS_LOAD_WINCE - Factory default for CMOS Windows CE boot enable.

CONFIG_CMOS_BOOT_0 - Factory default value for CMOS 1st boot action.

CONFIG_CMOS_BOOT_1 - Factory default value for CMOS 2nd boot action.

CONFIG_CMOS_BOOT_2 - Factory default value for CMOS 3rd boot action.

CONFIG_CMOS_BOOT_3 - Factory default value for CMOS 4th boot action.

CONFIG_CMOS_BOOT_4 - Factory default value for CMOS 5th boot action.

CONFIG_CMOS_BOOT_5 - Factory default value for CMOS 6th boot action.

CONFIG_CMOS_FLOPPY_0 - Factory default device assignment for 1st floppy.

CONFIG_CMOS_FLOPPY_1 - Factory default device assignment for 2nd floppy.

CONFIG_CMOS_FLOPPY_2 - Factory default device assignment for 3rd floppy.

CONFIG_CMOS_FLOPPY_3 - Factory default device assignment for 4th floppy.

CONFIG_CMOS_IDE_0 - Factory default value for CMOS hard drive type for 1st hard drive, when high nibble of **OEM_INIT_CMOS_HARD** is f0h.

CONFIG_CMOS_IDE_1 - Factory default for CMOS hard drive type for 2nd hard drive, when low nibble of **OEM_INIT_CMOS_HARD** is 0fh.

CONFIG_CMOS_IDE_2 - Factory default value for CMOS hard drive type for 3rd hard drive.

CONFIG_CMOS_IDE_3 - Factory default for CMOS hard drive type for 4th hard drive.

CONFIG_CMOS_IDE0_CYL - Factory default for CMOS fixed disk 0 cylinders (16 bits).

CONFIG_CMOS_IDE0_HDS - Factory default for CMOS fixed disk 0 heads (8 bits).

CONFIG_CMOS_IDE0_SPT - Factory default for CMOS fixed disk 0 sectors per track (8 bits).

CONFIG_CMOS_IDE1_CYL - Factory default for CMOS fixed disk 1 cylinders (16 bits).

CONFIG_CMOS_IDE1_HDS - Factory default for CMOS fixed disk 1 heads (8 bits).

CONFIG_CMOS_IDE1_SPT - Factory default for CMOS fixed disk 1 sectors per track (8 bits).

CONFIG_CMOS_IDE2_CYL - Factory default for CMOS fixed disk 2 cylinders (16 bits).

CONFIG_CMOS_IDE2_HDS - Factory default for CMOS fixed disk 2 heads (8 bits).

CONFIG_CMOS_IDE2_SPT - Factory default for CMOS fixed disk 2 sectors per track (8 bits).

CONFIG_CMOS_IDE3_CYL - Factory default for CMOS fixed disk 3 cylinders (16 bits).

CONFIG_CMOS_IDE3_HDS - Factory default for CMOS fixed disk 3 heads (8 bits).

CONFIG_CMOS_IDE3_SPT - Factory default for CMOS fixed disk 3 sectors per track (8 bits).

CONFIG_CMOS_TYPEMATIC_DELAY - Factory default for CMOS keyboard typematic delay.

CONFIG_CMOS_TYPEMATIC_RATE - Factory default for CMOS keyboard typematic repeat rate.

CONFIG_CMOS_FLOPPY_RETRY - Factory default for CMOS floppy disk I/O retry.

CONFIG_CMOS_EQUIP - Factory default for CMOS equipment byte.

7.1.25 OPTION_SUPPORT_NPX Option

The **OPTION_SUPPORT_NPX** option enables or disables code in the BIOS that supports on-board (i486-Pentium III) or outboard (287 or 387) numeric coprocessors.

If you have numeric coprocessor hardware in your target, it is necessary to enable this option to ensure that the hardware is initialized to a well-defined state and that the status bits in the CR0 register (i486-Pentium III only) are set appropriately for floating point emulators to work properly.

Values:

1 - Enable numeric coprocessor support.
0 - Disable numeric coprocessor support.

Related Parameters:

None.

7.1.26 OPTION_SUPPORT_V25 Option

The **OPTION_SUPPORT_V25** option enables or disables code in the time BIOS changes the way the INT 8h and INT 1ch interrupt service routines call one another. In PC-compatible systems, INT 8h calls INT 1ch when a timer tick occurs. In V25 systems, a CPU timer is hard-wired to INT 1ch, which in turn calls INT 8h instead.

This option does not provide instant support for V25 processors across all features of the BIOS; that is a function of the CPU Personality Module architecture.

Values:

1 - Enable V25 support in time BIOS.
0 - Disable V25 support in time BIOS.

Related Parameters:

None.

7.1.27 OPTION_SUPPORT_XT_NMI Option

The **OPTION_SUPPORT_XT_NMI** option enables or disables code in the BIOS that clears outstanding NMI interrupts during POST on PC-compatible machines. Do not set this option on non IBM-PC platforms without thoroughly understanding the ramifications.

The XT NMI I/O port is at address a0h. This same location is used on PC/AT platforms as the second 8259 interrupt controller's base I/O port. Thus, **OPTION_SUPPORT_8259_2** and **OPTION_SUPPORT_XT_NMI** are mutually exclusive.

Values:

1 - Enable XT NMI clearing during POST.
0 - Disable XT NMI clearing during POST.

Related Parameters:

OPTION_SUPPORT_8255 - Required for this option to work.
OPTION_SUPPORT_8259_2 - Must be disabled for this option to work.

7.1.28 OPTION_SUPPORT_VIDEO Option

The **OPTION_SUPPORT_VIDEO** option enables or disables code in the BIOS that provides support for video CRT controllers (such as MDA, CGA, and VGA) and LCD controllers.

This option does not specifically enable support for a certain video monitor or LCD controller; that is handled with the **OPTION_VIDEO_xxx** options. Thus, if **OPTION_SUPPORT_VIDEO** is enabled, one of these other options, such as **OPTION_VIDEO_6845** or **OPTION_VIDEO_AMDELAN** must be enabled.

This option operates independently of the console redirection feature, enabled with **OPTION_SUPPORT_CON_REDIRECTOR**. If you are using I/O redirection to COM ports, then you do not need to set **OPTION_SUPPORT_VIDEO**, except if you want to use *both* the video subsystem *and* redirected I/O. In this case, you should also enable **OPTION_VIDEO_DUPLICATE**, which will cause all I/O sent to the standard INT 10h video display to also be sent to the COM port of your choosing.

For a complete video system that includes the standard 6845 video controller found in PC, PC/XT, and PC/AT-compatible monochrome and color adapters, you must also enable **OPTION_VIDEO_6845**.

Video boards, such as VGA and Super VGA cards, actually contain a ROM BIOS extension that must be scanned. When scanned, the ROM on the card actually takes over for the EMBEDDED BIOS video services, and only calls EMBEDDED BIOS to do simple functions. The complex ones are all handled in the VGA ROM. To enabled this, you'll also need to enable **OPTION_SUPPORT_VIDEO_BOARDS**, and set **CONFIG_VIDEO_ROM_SCAN** to either 0c000h or 0e000h, depending on where the VGA ROM BIOS is located in your target. On ISA desktop PCs, this value is 0c000h.

In addition, if you are using memory-mapped video such as that found in 6845-based designs, you should also enable **OPTION_VIDEO_VIDEOMEM** so that POST can test it, and automatically make an automatic determination about which video adapter is being used in the system.

If you have an AMD SC300 or AMD SC400 CPU, then EMBEDDED BIOS can support its LCD controller when you enable **OPTION_VIDEO_AMDELAN** in conjunction with this option.

An alternate to 6845 CRT controller support is the Hitachi's HD61830 LCD controller, enabled with **OPTION_VIDEO_HD61830**. Please note that this code was donated to General Software by a German customer, and the code has German comments. We regret that we are unable to speak German well enough to support the code, but it is provided in the event that you speak German well enough to maintain it. This code is working on the HD61830 in actual applications.

Another customer-provided driver is `HDMLCD.ASM`, enabled with the **INC\OPTIONS.INC OPTION_VIDEO_HDMLCD**. This driver supports a set of LCD panels with different row/column geometries, and is known to work with EMBEDDED BIOS. Please note that this code was also donated to General Software, and we cannot directly support it.

Customers requiring LCD support for the CPU codenamed EMERALD may obtain the code by having the silicon vendor contact General Software in writing. Then, **OPTION_VIDEO_EMERALD** enables the code. Details about EMERALD are confidential and provided only when the silicon vendor approves release of details in writing.

If you have a special CRT or LCD controller that you plan on using in your design, enable **OPTION_VIDEO_CUSTOMER**, and add your code to the already-started `SYSTEM\CUSTVID.ASM`. This allows you to add your own code to just one module, instead of editing the many modules that support the 6845 throughout the BIOS.

Other video options may have been added to EMBEDDED BIOS since this documentation was printed; therefore, consult `INC\OPTIONS.INC` for a list of your video options.

If video is memory-mapped (as is the case with the 6845 controller), then you'll need to make sure that **CONFIG_VIDEO_SEG_GRAPHIC**, **CONFIG_VIDEO_SEG_MONO**, and **CONFIG_VIDEO_SEG_COLOR**, are all set to the proper segement addresses where video memory is to be found for the relevant video modes. On desktop PCs, these values are 0a000h, 0b000h, and 0b800h, respectively.

Values:

1 - Enable video controller support.
0 - Disable video controller support.

Related Parameters:

OPTION_VIDEO_6845 - Support PC-compatible 6845 CRT controller (monochrome, color, Hercules, EGA, VGA, and SVGA designs).

OPTION_VIDEO_AMDELAN - Support SC300 and SC400 LCD controllers.

OPTION_VIDEO_EMERALD - Support EMERALD LCD controller.

OPTION_VIDEO_HD61830 - Support Hitachi HD61830 LCD controller.

OPTION_VIDEO_HDMLCD - Support another family of LCD controllers.

OPTION_VIDEO_CUSTOMER - Support OEM-written custom video driver SYSTEM\CUSTVID.ASM.

OPTION_VIDEO_DUPLICATE - Send video output to both serial port and primary video device.

OPTION_VIDEO_VIDEOMEM - Test video RAM during POST, and automatically determine video board type (mono or color).

OPTION_SUPPORT_CON_REDIRECTOR - Enable support for console redirection over RS-232 link to host's terminal program.

CONFIG_VIDEO_SEG_GRAPHIC - Segment address of video memory when in graphics mode.

CONFIG_VIDEO_SEG_MONO - Segment address of video memory when in monochrome mode.

CONFIG_VIDEO_SEG_COLOR - Segment address of video memory when in color mode.

OPTION_SUPPORT_VIDEO_BOARDS - Scan for EGA/VGA/SVGA ROM BIOS extensions.

CONFIG_VIDEO_ROM_SCAN - Segment address of EGA/VGA/SVGA ROM BIOS extensions.

7.1.29 OPTION_SUPPORT_KEYBOARD Option

The **OPTION_SUPPORT_KEYBOARD** option enables or disables code in the BIOS that provides support for a PC, PC/XT, or PC/AT keyboard controller.

This option does not specifically enable support for a certain type of keyboard controller; that is handled with the **OPTION_KEYBOARD_xxx** options. Thus, if **OPTION_SUPPORT_KEYBOARD** is enabled, one of these other options, such as **OPTION_KEYBOARD_PCAT** or **OPTION_KEYBOARD_CUSTOMER** must be enabled.

This option operates independently of the console redirection feature, enabled with **OPTION_SUPPORT_CON_REDIRECTOR**. If you are using I/O redirection to COM ports, then you do not need to set **OPTION_SUPPORT_KEYBOARD**, except if you want to use *both* the keyboard subsystem *and* redirected I/O.

For a complete keyboard system that includes the standard PC/AT 8042 keyboard controller found in PC, PC/XT, and PC/AT-compatible systems, you must also enable **OPTION_KEYBOARD_PCAT** and **OPTION_SUPPORT_8042**. This is the default setting of these options. Note that if you are using the 8042, you will also need to configure options relating to the 8042 (see that section for details).

For a complete keyboard system that includes the standard PC/XT 8255 keyboard controller (note: this is not PC/AT compatible, it is PC/XT compatible), you must enable **OPTION_SUPPORT_8255** and **OPTION_KEYBOARD_PCAT**.

If you have a custom keyboard (not just a keypad that will be driven by your application for a few proprietary functions), then you can add support for it directly in the core BIOS by enabling **OPTION_SUPPORT_KEYBOARD** and **OPTION_KEYBOARD_CUSTOMER**. Then, edit SYSTEM\CUSTKBD.ASM, and add the required code to drive the keyboard device.

Values:

1 - Enable keyboard support (not console redirection).
0 - Disable keyboard support (not console redirection).

Related Parameters:

OPTION_SUPPORT_8255 - Use 8255 as keyboard interface.

OPTION_SUPPORT_8042 - Use 8042 keyboard controller.

OPTION_KEYBOARD_PCAT - Use PC/AT keyboard.

OPTION_KEYBOARD_PCXT - Use PC/XT keyboard.

OPTION_KEYBOARD_CHIPSET - Use keyboard driver defined in CSPM.

OPTION_KEYBOARD_CUSTOMER - Use custom controller/keyboard.

OPTION_KEYBOARD_MATRIX - Enable special key translation on matrix keyboards.

OPTION_SUPPORT_CON_REDIRECTOR - Enable support for console redirection over RS-232 link to host’s terminal program.

7.1.30 OPTION_SUPPORT_TESTBASEMEM Option

The **OPTION_SUPPORT_TESTBASEMEM** option enables or disables code in POST that tests the lower 64KB region of memory before proceeding with system initialization. This testing takes time and is usually not desired in embedded systems that must boot as quickly as possible.

While desktop PCs normally test a whole 64KB at the bottom of lower memory, EMBEDDED BIOS can be configured with the **CONFIG_TESTBASE_SIZE** parameter to test any size from 16KB to 64KB. For optimal boot times, it is recommended that this value be reduced to a lower value such as 4, because this test is performed before wait states are reduced and caches are enabled. Caution: Do not try to reduce this value below 16, since that will result in mismanagement of the boot-time stack, causing erratic pre-boot behavior.

Values:

1 - Enable testing of bottom memory block on power-on.
0 - Disable testing of bottom memory block on power-on.

Related Parameters:

CONFIG_TESTBASE_SIZE - Specifies size of block to test.

7.1.31 OPTION_SUPPORT_PAGEREG Option

The **OPTION_SUPPORT_PAGEREG** option enables or disables code in POST that supports the PC/AT-compatible page register file. All ISA-compatible motherboards support this register file.

This option must be enabled for ISA-class targets to successfully perform DRAM refresh via the 8237A and the 8254; and to provide DMA-based floppy I/O. If neither of these functions need to be present in your target, you should disable this option.

The page register file is used in conjunction with the two 8237A DMA controllers to extend their addressability to the entire lower 1MB. In some systems, the page register file contains 8-bit values instead of 4-bit values; therefore, the address range is extended by 4 bits to 16MB.

Values:

1 - Enable page register support.
0 - Disable page register support.

Related Parameters:

None.

7.1.32 OPTION_SUPPORT_XTEXPANSION Option

The **OPTION_SUPPORT_XTEXPANSION** option enables or disables code in POST that supports the PC/XT-compatible expansion box. The XT expansion box is an architectural relic that is no longer used in ISA-class systems, but may be necessary in some PC/XT-compatible embedded Single Board Computers.

Unless your hardware documentation explictly states that this programming is required, you should disable this option.

Values:

1 - Enable XT expansion box support.
0 - Disable XT expansion box support.

Related Parameters:

None.

7.1.33 OPTION_SUPPORT_SCT Option

The **OPTION_SUPPORT_SCT** option enables or disables code in the BIOS that builds a System Configuration Table and makes it available through the INT 15h BIOS service. The SCT is inspected by DOS and by some application programs to determine what features are supported by the BIOS.

Values:

1 - Enable SCT support.
0 - Disable SCT support.

Related Parameters:

None.

7.1.34 OPTION_SUPPORT_PROTECT_MODE Option

The **OPTION_SUPPORT_PROTECT_MODE** option enables or disables code in the BIOS that supports switching between real mode and protected mode (80386 and above processors only), including the memory move functions provided by the INT 15h general services BIOS interrupt that generally support extended memory at the BIOS level.

This support is also required for POST to test extended memory during system initialization.

Do not enable this option if the target processor is not capable of operating in protected mode. The 8088, 8086, V20, V25, 80188, and 80186 processors are not capable of supporting protected mode programming. EMBEDDED BIOS supports protected mode on 386, 486, Pentium, and P6-class CPUs. Support for the 80286 has been discontinued because the part’s life has ended.

Issues related to protected mode support are A20 gating and mode switching from protected mode back to real mode. If you enable **OPTION_SUPPORT_PROTECT_MODE**, then you must enable an A20 gating option, and you must enable a "to-real" option.

Selecting A20 Gate Controls

In PC/AT-class targets, there exists an A20 gate that controls whether access to addresses above 1MB simply wrap around to physical address 0 or address the actual physical memory above 1MB. This mechanism is called the A20 gate, and the control over the gate is handled differently depending on what A20 gate hardware is present in the system.

If you have an 8042 keyboard controller and the keyboard controller is providing A20 gate control, then **OPTION_A20_8042** must be enabled.

If you have a PS/2-compatible I/O port 92h, then **OPTION_A20_PORT92** must be enabled. Sometimes targets (such as the AMD SC300 Elan evaluation board) provide a wired-OR configuration that requires _both_ the 8042 and port 92h to control the A20 gate. In this case, set both **OPTION_A20_8042** and **OPTION_A20_PORT92**.

If you are using a chipset with a "fast A20 gate", and if it is _not_ just an implementation of the PS/2-compatible port 92h, then enable **OPTION_A20_CHIPSET**, enable **OPTION_SUPPORT_CHIPSET**, and edit the **CsEnableA20** and **CsDisableA20** routines in your Chipset Personality Module to provide the necessary manipulation of the chipset hardware to toggle the A20 line. For more information about these chipset routines, see Chapter 19. Warning: The SC300 and SC310 have errata regarding the A20 gating. You must have a full understanding of these errata before proceeding along these lines.

If your specialty CPU includes a "fast A20 gate" that is _not_ just an implementation of the PS/2-compatible port 92h, then enable **OPTION_A20_CPU**, set CPUCLASS to a particular CPU type, and edit the **CpuEnableA20** and **CpuDisableA20** routines in your CPU Personality Module to provide the necessary manipulation of the CPU hardware to toggle the A20 line. For more information about these CPU routines, see Chapter 18.

If you have a board design that employs a special discrete A20 gate, or if the A20 gating logic is not present in your Chipset or CPU Personality Modules, then you can enable **OPTION_A20_BOARD**, and add code to the board module in the **BoardEnableA20** and **BoardDisableA20** routines.

Make sure that initially, you disable **OPTION_A20_FAILPOST**, until your BIOS is fully running. The reason for this is that POST has an A20 test that uses a memory wraparound test to see if the A20 gate is working. This requires memory above 1MB to be available. If this memory is not available, the test fails, even though the A20 gate may be working. You should enable this option only if your target will be using memory at 1MB, and you require that the A20 gate be tested.

Selecting Mode Switching Controls

On 80286 CPUs, there is no software-only procedure for the BIOS to switch back to real mode after performing a protected mode operation. Instead, hardware must assist by saving the state of the executing program, rebooting, and then restoring the state of the executing program so that it can continue just as though no CPU reset had occurred. On 386 and above CPUs, this is not a problem because a "switch to real-mode" CPU instruction is available (it is MOV CR0, EAX).

On 80286 platforms, there exist three different ways to solve this problem, depending on the supporting hardware.

If **OPTION_TOREAL_PORT92** is enabled, then the BIOS reboots the machine by manipulating the PS/2-compatible I/O port 92h. IBM PS/2 models 60 and 70 require this approach to rebooting.

If **OPTION_TOREAL_8042** is enabled, then the BIOS reboots the machine by sending the 8042 keyboard controller a reboot command. The keyboard controller in turn enables the reset line on the CPU, causing a reset of the CPU to occur. IBM PC/AT targets require this method.

If your target has a 386 or later CPU, then it can switch to real mode with a software instruction alone. While you could use one of the above techniques, it would prove much slower than to simply enable **OPTION_TOREAL_CPU**.

Selecting Reboot Methods

Closely related to switching to real mode is the method used to reboot the target. Essentially, the process of rebooting is the same as switching to real mode for the 80286, except that the state of the BIOS is not saved so that a protected mode operation returns control to the BIOS. You must enable one of the rebooting options to support the reboot operation, depending on the mode switching analysis you did above.

Select **OPTION_REBOOT_JUMP** for real-mode only targets such as the 8086, V20, or 80186.

Select **OPTION_REBOOT_PORT92** if you have a PS/2-compatible I/O port 92h.

Select **OPTION_REBOOT_8042** if you have an 8042 keyboard controller controlling the CPU reset line.

Select **OPTION_REBOOT_CHIPSET** if your chipset provides a fast way to reboot the CPU (and also perhaps the memory controller).

Select **OPTION_REBOOT_BOARD** if your board’s design provides a fast way to reboot the CPU.

Selecting Extended Memory Limit

If you will be using Flash memory above 1MB, it is important to limit POST's extended memory scan so that it does not attempt to write to the Flash in its test. Make sure **CONFIG_MAX_EXT_MEMORY** is set properly, or POST could hang during its extended memory test.

Values:

1 - Enable protected mode and extended memory support.
0 - Disable protected mode and extended memory support.

Related Parameters:

OPTION_A20_8042 - Use 8042 to gate A20 line.

OPTION_A20_CHIPSET - Use chipset to gate A20 line.

OPTION_A20_BOARD - Use board to gate A20 line.

OPTION_A20_CPU - Use CPU to gate A20 line.

OPTION_A20_PORT92 - Use PS/2 compatible port 92h to gate A20 line.

OPTION_TOREAL_PORT92 - Use port 92h to switch to real mode.

OPTION_TOREAL_8042 - Use 8042 to switch to real mode.

OPTION_TOREAL_CPU - Use CPU instruction to switch to real mode.

OPTION_REBOOT_JUMP - Jump to reset vector to reset machine.

OPTION_REBOOT_PORT92 - Use port 92h to reset machine.

OPTION_REBOOT_8042 - Use 8042 to reset machine.

OPTION_REBOOT_CHIPSET - Use chipset to reset machine.

OPTION_REBOOT_BOARD - Use board to reset machine.

CONFIG_MAX_EXT_MEMORY - Set upper limit of extended memory.

7.1.35 OPTION_SUPPORT_SERIAL Option

The **OPTION_SUPPORT_SERIAL** option enables or disables code in the BIOS that supports the serial I/O services of INT 14h.

Once INT 14h services are enabled, the specific hardware that provides serial I/O must be selected through sub-options.

OPTION_SERIAL_8250 causes 8250-compatible UARTs (includes 16450 and 16550 UARTs as well) to be used. If are using external standard PC-compatible UARTs, select this option. If your CPU or chipset supports 8250-compatible UARTs, this option should also be enabled (for example, the Intel 386-EX CPU contains 8250-compatible UARTs).

OPTION_SERIAL_CPU enables codepaths that support special UARTs that are integrated into the CPU itself. The 80C186-EC is an example of a CPU that has nonstandard UARTs onboard the CPU. If this option is enabled, then the **CPUCLASS** parameter must be configured for the correct CPU type, and the CPU routines in the CPU Personality Module must contain the necessary code to program the UARTs.

OPTION_SERIAL_WAIT_DSR causes INT 14h to wait for Data Set Ready to become active before data are received. For 3-wire serial I/O cables, this option should be disabled.

OPTION_SERIAL_WAIT_DSRCTS causes INT 14h to wait for Data Set Ready and Clear to Send to become active before data are transmitted. This is also intended for more fully-featured cables.

OPTION_SERIAL_FIFO causes the 8250-compatible driver code to enable the FIFO on 8250-compatible UARTs that support FIFOs. This reduces losses due to the receive buffer being full when another character is received. Note that BIOS does not use interrupt-driven I/O for INT 14h, although it does use interrupt-driven receive paths for Manufacturing Mode.

OPTION_SERIAL_HALT causes the 8250-compatible driver code to execute a HLT instruction when it must spin-wait for an incoming character over a serial port on a read with wait. This allows designs that must reduce power consumption to a minimum to switch to a very low power mode when polling for input over the serial port.

OPTION_SERIAL_9600_BAUD causes the 8250-compatible driver code to always set the communications parameters to 9600 baud, no parity, and one stop bit, whenever commanded to change the parameters via INT 14h. This allows console redirection to be employed at 9600 baud even when MS-DOS attempts to reset the serial port baud rates to 2400 baud, even parity, and one stop bit.

Values:

1 - Enable INT 14h services.
0 - Disable INT 14h services.

Related Parameters:

OPTION_SERIAL_8250 - Enable 8250 serial ports.

OPTION_SERIAL_CPU - Enable on-board CPU serial ports.

OPTION_SERIAL_WAIT_DSR - Wait for DSR before receiving.

OPTION_SERIAL_WAIT_DSRCTS - Wait for DSR & CTS before transmitting.

OPTION_SUPPORT_8250 - Enable 8250-compatible UARTs.

OPTION_SERIAL_FIFO - Enable 8250-compatible FIFO.

OPTION_SERIAL_HALT - Issue HLT if spinwait on read becomes necessary.

OPTION_SERIAL_9600_BAUD - Always use 9600 baud.

CONFIG_SERIAL_TIMEOUT - COM port timeout in seconds. This timeout is used in INT 14h requests.

COM1_BASE - I/O port address for COM1 UART.
COM2_BASE - I/O port address for COM2 UART.
COM3_BASE - I/O port address for COM3 UART.
COM4_BASE - I/O port address for COM4 UART.

COM1_INIT - Initialization setting for COM1 UART.
COM2_INIT - Initialization setting for COM2 UART.
COM3_INIT - Initialization setting for COM3 UART.
COM4_INIT - Initialization setting for COM4 UART.

7.1.36 OPTION_SUPPORT_PARALLEL Option

The **OPTION_SUPPORT_PARALLEL** option enables or disables code in the BIOS that supports the parallel I/O services of INT 17h.

If you are using PC/XT or PC/AT-compatible parallel ports, then you should enable **OPTION_PARALLEL_EXTERNAL**. Enable **OPTION_PARALLEL_CPU** if you're using CPU-integrated parallel ports that are not compatible with standard PC-compatible parallel ports.

INT 17h services provide timeouts to account for equipment failures or problems such as "printer out of paper" conditions. The timeouts are specified in seconds with the configuration parameter, **CONFIG_PARALLEL_TIMEOUT**. The recommended value to start with on this parameter is 1 second.

During initialization, POST may require some delay in order to determine if a parallel port is functioning properly. This delay is specified with the **CONFIG_WAIT_LPT** parameter. The suggested value to start with is 1000h and is CPU speed-dependent (value given is for a 386-25 class machine).

Values:

1 - Enable INT 17h services.
0 - Disable INT 17h services.

Related Parameters:

OPTION_PARALLEL_EXTERNAL - Enable external parallel ports.
OPTION_PARALLEL_CPU - Enable on-board CPU parallel ports.
CONFIG_PARALLEL_TIMEOUT - Specifies timeout for INT 17h.
CONFIG_WAIT_LPT - Specifies initialization delay for POST.

7.1.37 OPTION_SUPPORT_ROM_EXTENSIONS Option

The **OPTION_SUPPORT_ROM_EXTENSIONS** option enables or disables code in the BIOS that scans for user-defined ROM extensions in the adapter area. This option does not enable the scan for Embedded DOS-ROM or for external VGA ROM BIOS extensions.

This option should be enabled by most adaptations unless special memory maps are being used that would be interfered with by the ROM scan.

The ROM scan is executed by POST after basic keyboard and video I/O services are available, so that ROM extensions can use these services to display messages and otherwise interact with the user.

There are several configuration parameters that govern the scope of the ROM scan.

CONFIG_LOW_ROM_SCAN specifies the first segment address of the scan. For desktop PC's, this value is C800h.

CONFIG_HIGH_ROM_SCAN specifies the last segment address of the scan; this address is actually the first one beyond the scan and it is not scanned. The desktop PC standard is EE00h.

CONFIG_ROM_SCAN_INTERVAL specifies the number of bytes between scan addresses. The desktop PC standard is 2048 (there is the possibility for a ROM extension every 2KB in the address space). This can be adjusted to values such as 1KB for embedded designs to maximize the use of the ROM scan address space.

Additional configuration parameters affect this ROM scan, because there are additional ROM BIOS extensions that are called at different times than this general purpose scan. Because they cannot be called twice, *they are excluded from the scan.*

CONFIG_VIDEO_ROM_SCAN specifies the segment address of the VGA video BIOS, if **OPTION_SUPPORT_VIDEO_BOARDS** is enabled. The desktop PC standard for this segment address is C000h, but in some embedded designs this value is E000h.

CONFIG_MINI_DOS_SCAN specifies the segment address of the last-chance boot ROM, which traditionally held ROM BASIC in the IBM PC. Today, EMBEDDED BIOS uses Embedded DOS-ROM as the operating system it boots from ROM, and this parameter specifies its segment address in ROM.

Values:

1 - Enable general ROM scan.
0 - Disable general ROM scan.

Related Parameters:

CONFIG_LOW_ROM_SCAN - First segment to scan for extensions.

CONFIG_HIGH_ROM_SCAN - Last segment to scan for extensions.

CONFIG_ROM_SCAN_INTERVAL - Number of bytes to skip betwen ROM extensions.

CONFIG_MINI_DOS_SCAN - Segment address of Embedded DOS-ROM operating system.

CONFIG_VIDEO_ROM_SCAN - Segment address of video ROM BIOS extension.

7.1.38 OPTION_SUPPORT_VIDEO_BOARDS Option

The **OPTION_SUPPORT_VIDEO_BOARDS** option enables or disables code in the BIOS that scans for an EGA or VGA ROM extension in the adapter area to supplement or take-over the video BIOS services provided in the core system BIOS.

This option should be enabled by most adaptations unless special memory maps are being used that would be interfered with by the ROM scan.

This option does not apply to PCI-based designs. In these systems, PCI video adapters are supported with a PCI bus scan. PCI device option ROMs are always mapped into memory at a dynamic location, and are never found to be at a predetermined location in the address space. If the target is PCI-based, and also supports ISA slots, then this option should be enabled so that both PCI and ISA VGA cards can be used.

If this option is enabled, you must also specify the segment address of the ROM BIOS extension to be scanned by setting **CONFIG_VIDEO_ROM_SCAN** appropriately. The desktop PC standard for this value is C000h, but it can be set to other addresses such as E000h, if required.

The address chosen for **CONFIG_VIDEO_ROM_SCAN** is automatically excluded from the general ROM BIOS extension scan.

If this option is enabled, the ROM BIOS extension that receives control will almost certainly require that you have enabled support for a 6845 video controller, and INT 10h support. This requires that **OPTION_SUPPORT_VIDEO**, **OPTION_VIDEO_6845**, and **OPTION_VIDEO_VIDEOMEM** be enabled. Additionally, **CONFIG_VIDEO_SEG_GRAPHIC**, **CONFIG_VIDEO_SEG_MONO**, and **CONFIG_VIDEO_SEG_COLOR** must also be set to the desktop standard addresses-- A000h, B000h, and B800h, respectively.

Values:

1 - Enable Video ROM scan.
0 - Disable Video ROM scan.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable INT 10h BIOS service.

OPTION_VIDEO_6845 - Enable 6845 video controller support.

OPTION_VIDEO_VIDEOMEM - Enable scan of video memory and autodetection of video monitor type.

CONFIG_VIDEO_SEG_GRAPHIC - Segment of video RAM for graphic modes.

CONFIG_VIDEO_SEG_MONO - Segment of video RAM for monochrome mode.

CONFIG_VIDEO_SEG_COLOR - Segment of video RAM for color modes.

CONFIG_VIDEO_ROM_SCAN - Segment to scan for video ROM BIOS extension.

7.1.39 OPTION_SUPPORT_SOUND Option

The **OPTION_SUPPORT_SOUND** option enables or disables code in the BIOS that supports the programming of the speaker for clicks, beeps, and tones.

This feature is used during POST to signal errors before video services are available, and also during steady state of the system to indicate that the keyboard typeahead buffer is full.

To enable the beep that occurs after POST has completed and is ready to transfer control to the operating system, enable **OPTION_BOOT_BEEP**.

This feature requires the **OPTION_SUPPORT_PORT_B** be enabled so that the hardware can be properly controlled.

It also requires that a timer be available (typically, **OPTION_SUPPORT_8254** is enabled to satisfy this requirement).

If you are supporting sound with a CPU timer, then you should enable **OPTION_SOUND_CPU** and disable **OPTION_SOUND_8254_8255**, so that EMBEDDED BIOS can route sound requests to the right hardware.

If you are using either an 8254 or the PC/XT-compatible 8255 PIO support to create sound, then **OPTION_SOUND_8254_8255** should be enabled.

If your platform has custom sound circuitry that requires special programming, the board module's sound functions can be called by enabling **OPTION_SOUND_BOARD**.

The frequency and duration of the beep sounds are controlled with three configuration options, as follows. We recommend that you use the default values and modify them to suit your taste once you have actually heard what tones the default values produce.

The **CONFIG_BEEP_LENGTH** parameter is a value that indicates how long the beep should last, in "CPU loops." This CPU-speed-dependent value is necessarily so because the operation of a timer in the system cannot be assumed.

The **CONFIG_BEEP_CYCLE** parameter is a value that is used to delay between toggling the speaker's position when no timer is available to manually control the speaker's oscillation. This is effectively an inverse frequency control for use at points in POST before the timer has been initialized, or in systems without an 8254 timer.

The **CONFIG_BEEP_8254_TONE** parameter is a value that is loaded into the 8254 to provide a different beep frequency, not in terms of CPU loops, but values related to the independently-clocked 8254.

Values:

1 - Enable speaker support.
0 - Disable speaker support.

Related Parameters:

OPTION_SUPPORT_PORT_B - Support PORT B sound architecture.

OPTION_SUPPORT_8255 - Support XT-compatible peripherals.

OPTION_SUPPORT_8042 - Support AT-compatible keyboard controller.

OPTION_SUPPORT_8254 - Support AT-compatible timer controller.

OPTION_BOOT_BEEP - Enable beep upon POST completion.

OPTION_SOUND_CPU - Use CPU for sound support.

OPTION_SOUND_8254_8255 - Use 8254 or 8255 for sound support.

OPTION_SOUND_BOARD - Use board module routine for sound support.

OPTION_TIMER_CPU - Enable CPU integrated timer.

OPTION_TIMER_8254 - Enable AT-compatible timer controller.

CONFIG_BEEP_LENGTH - Duration of beep.

CONFIG_BEEP_CYCLE - Inverse beep frequency control for 8255 only.

CONFIG_BEEP_8254_TONE - Beep frequency control for 8254 only.

7.1.40 OPTION_SUPPORT_DEVICECALLS Option

The **OPTION_SUPPORT_DEVICECALLS** option enables or disables code in the BIOS that supports the BIOS up-calls that tell the operating system or application software that BIOS managed devices are waiting, or that certain keys, such as the **SysReq** key, are being pressed on the keyboard.

Values:

1 - Enable BIOS device up-calls.
0 - Disable BIOS device up-calls.

Related Parameters:

None.

7.1.41 OPTION_SUPPORT_TIMEBIOS Option

The **OPTION_SUPPORT_TIMEBIOS** option enables or disables code in the BIOS that supports the date/time services of INT 1ah.

The INT 1ah services can use either a CMOS Real Time Clock (RTC) part or a counter-timer to keep time. If an RTC is used, then the date is also maintained. If no RTC is used, then the counter-timer can be either a standard 8254 counter-timer unit or a proprietary counter-timer unit in an integrated CPU.

There are two types of RTC parts supported in this version of EMBEDDED BIOS. You should enable **OPTION_RTC_CMOS** if you have the Dallas equivalent part with CMOS RAM, and also enable **OPTION_SUPPORT_CMOS** at the same time to enable its support in SETUP.

If you have the 72421 RTC instead of the Dallas part, enable **OPTION_RTC_72421** instead, and set **OPTION_SUPPORT_CMOS** if you have CMOS RAM.

POST initializes the RTC with a default mode byte that is used to configure the RTC. This is configured with the **CONFIG_DEFAULT_RTC** parameter. The default value is 26h. Other values may be obtained by studying the documentation for the RTC part you are using.

Values:

1 - Enable date/time services.
0 - Disable date/time services.

Related Parameters:

OPTION_SUPPORT_CMOS - Enables CMOS RAM support.
OPTION_SUPPORT_8254 - Enables 8254 counter-timer support.
OPTION_TIMER_8254 - Use 8254 for counter-timer support.
OPTION_TIMER_CPU - Use CPU for counter-timer support.
OPTION_TIMER_BOARD - Use board for counter-timer support.
OPTION_RTC_CMOS - Enables RTC support with Dallas part.
OPTION_RTC_72421 - Enables RTC support with 72421 part.
CONFIG_DEFAULT_RTC - Initialization byte for RTC.

7.1.42 OPTION_SUPPORT_APM Option

The **OPTION_SUPPORT_APM** option enables or disables code in the BIOS that supports the Advanced Power Management API. The APM services (called through INT 15h function 53h) rely on the underlying power management support provided in the CHIPSET or CPU Personality Modules.

This option does not enable power management in the BIOS; that is done by enabling **OPTION_SUPPORT_POWERMAN**. The APM option only augments the power management subsystem by providing a standard API for operating systems and applications to communicate requests to the BIOS.

If you intend to use the chipset's power management functions, then enable **OPTION_POWERMAN_CHIPSET**. If the CPU's power management functions are to be used to support APM, then enable **OPTION_POWERMAN_CPU**. If special platforms require custom programming beyond simple CPU or chipset programming, then **OPTION_POWRMAN_BOARD** should be enabled, and the code placed in the board module. Please note that not all chipsets or CPUs are capable of implementing power management.

Values:

1 - Enable APM support.
0 - Disable APM support.

Related Parameters:

OPTION_POWERMAN_CPU - Use CPU Personality Module to implement power controls.

OPTION_POWERMAN_CHIPSET - Use Chipset Personality Module to implement power controls.

OPTION_POWERMAN_BOARD - Use Board Personality Module to implement power controls.

OPTION_SUPPORT_POWERMAN - Enable BIOS-level power management.

OPTION_SETUP_PWR_FEATURES - Enable device features power management Setup screen.

OPTION_SETUP_PWR_TIMEOUTS - Enable device timeouts power management Setup screen.

7.1.43 OPTION_SUPPORT_POWERMAN Option

The **OPTION_SUPPORT_POWERMAN** option enables or disables code in the BIOS that supports actual device-level power management. The power management system in the BIOS uses a power management tree created with a table in the project file to introduce an APM state machine for each device in the system, and to sequence state transitions of various system components in the proper order.

This option does not enable the APM API; that is done by enabling **OPTION_SUPPORT_APM**. The APM option only augments the power management subsystem by providing a standard API for operating systems and applications to communicate requests to the BIOS.

If you intend to use the chipset's power management functions, then enable **OPTION_POWERMAN_CHIPSET**. If the CPU's power management functions are to be used to support APM, then enable **OPTION_POWERMAN_CPU**. If special platforms require custom programming beyond simple CPU or chipset programming, then **OPTION_POWRMAN_BOARD** should be enabled, and the code placed in the board module. Please note that not all chipsets or CPUs are capable of implementing power management.

Values:

1 - Enable power management support.
0 - Disable power management support.

Related Parameters:

OPTION_POWERMAN_CPU - Use CPU Personality Module to implement power controls.

OPTION_POWERMAN_CHIPSET - Use Chipset Personality Module to implement power controls.

OPTION_POWERMAN_BOARD - Use Board Personality Module to implement power controls.

OPTION_SUPPORT_POWERMAN - Enable BIOS-level power management.

OPTION_SETUP_PWR_FEATURES - Enable device features power management Setup screen.

OPTION_SETUP_PWR_TIMEOUTS - Enable device timeouts power management Setup screen.

7.1.44 OPTION_SUPPORT_PCI Option

The **OPTION_SUPPORT_PCI** option enables or disables code in the BIOS that supports the PCI API and PCI bus initialization in PCI-based systems.

Several PCI-related parameters in the project file specify how PCI devices will be treated during POST; see the related parameters section for the lengthy list.

Values:

1 - Enable PCI support.
0 - Disable PCI support.

Related Parameters:

OPTION_SUPPORT_PCI_POSTMSGS – Enable display of PCI messages during POST, such as the device display.

CONFIG_PCI_ROM_SHADOW_START - First segment of available shadow RAM to be used for storing copies of PCI device option ROMs.

CONFIG_PCI_ROM_MAP - High 16 bits of device ROM extension mapping area.

CONFIG_PCI_MEM_AVAIL - High 16 bits of 1st memory address space assignable to PCI devices.

CONFIG_PCI_IO_PORT_BASE - Lower 10 bits of 1st I/O addresses offered to PCI devices.

CONFIG_PCI_IO_ALLOC specifies the size of the ISA I/O address space in bytes, to allow the PCI system to skip past the replicated ISA I/O ports during its allocation of PCI I/O resources.

CONFIG_PCI_IO_LENGTH specifies the number of I/O locations at the base I/O address.

CONFIG_PCI_TMP_TBL_SEG specifies the segment address of low RAM to be used by PCI during the preboot environment for scratch purposes.

CONFIG_PCI_BM_OFFSET specifies the offset into the scratch RAM segment where the PCI bus map will be built during the preboot PCI bus scan.

CONFIG_PCI_LATENCY specifies the value to be stored into each PCI device's latency field during initialization.

CONFIG_PCI_IRQ_BITMAP specifies a 16-bit mask containing bits that correspond with IRQs to be assigned to PCI instead of the rest of the system.

7.1.45 OPTION_SUPPORT_PCI_POSTMSGS Option

The **OPTION_SUPPORT_PCI_POSTMSGS** option enables or disables code in the BIOS that displays messages during POST when PCI is initializing, such as the PCI configuration table.

Values:

1 - Enable PCI messages in POST.
0 - Disable PCI messages in POST.

Related Parameters:

OPTION_SUPPORT_PCI - Enable PCI support.
OPTION_SUPPORT_POSTMSGS - Enable POST messages in general.

7.1.46 OPTION_SUPPORT_MCA Option

The **OPTION_SUPPORT_MCA** option enables or disables the assembly of MCA-compatible identifying data structures in the core BIOS.

If this option is set, then certain system software, such as OS/2, HIMEM.SYS, and DOS extenders will make decisions about how to gate the A20 line and access other system functions differently than they would otherwise. Do not set this option without fully understanding its ramifications.

Values:

1 - Enable MCA support.
0 - Disable MCA support.

Related Parameters:

CONFIG_PS2_MOUSE_IRQ – Specify IRQ used to support PS/2 mouse.

CONFIG_PS2_MOUSE_WAIT – Specify retry limit for driver to wait for PS/2 commands to be accepted by 8042.

7.1.47 OPTION_SUPPORT_PS2MOUSE Option

The **OPTION_SUPPORT_PS2MOUSE** option enables or disables code in the BIOS that supports the PS/2-compatible mouse through the 8042 keyboard controller.

Several PS/2 mouse-related parameters in the project file specify how to fine-tune the interaction between the mouse and the keyboard controller, and the keyboard controller and the CPU.

CONFIG_PS2_MOUSE_IRQ specifies the system interrupt level that the keyboard controller associates with mouse-related activities.

CONFIG_PS2_MOUSE_LOOP specifies the maximum number of loops in a timeout loop to wait for the 8042 to report the status of the mouse before a device timeout is declared.

Values:

1 - Enable PS/2 mouse support.
0 - Disable PS/2 mouse support.

Related Parameters:

CONFIG_PS2_MOUSE_IRQ - System interrupt level associated with mouse hardware events.

CONFIG_PS2_MOUSE_LOOP - Timeout value for information from keyboard controller.

7.1.48 OPTION_SUPPORT_WATCHDOG Option

The **OPTION_SUPPORT_WATCHDOG** option enables or disables code in the BIOS that supports the Watchdog Timer API. The watchdog timer services rely on underlying watchdog timer support provided in the Chipset, CPU, or Board Personality Modules.

To enable chipset support for the watchdog timer, enable **OPTION_WATCHDOG_CHIPSET**. To enable CPU Personality Module support for the watchdog timer, enable **OPTION_WATCHDOG_CPU**. To enable Board Personality Module support for the watchdog timer, enable **OPTION_WATCHDOG_BOARD**.

The watchdog timer can be configured to operate in two different ways. First, the BIOS can automatically "kick" the dog every timer tick in its INT 8h handler, so that the application program and operating system are relieved of this responsibility. Only in the event that interrupt latency becomes larger than the watchdog timer's limit, does the watchdog timer expire. In this case, **OPTION_WATCHDOG_TIMER_KICK** must be enabled.

The second method for employing the watchdog timer is for the operating system or application to assume full responsibility for kicking the dog. In this case, **OPTION_WATCHDOG_TIMER_KICK** must be disabled.

Values:

1 - Enable watchdog timer support.
0 - Disable watchdog timer support.

Related Parameters:

OPTION_WATCHDOG_CPU - Use CPU Personality Module to implement watchdog controls.

OPTION_WATCHDOG_CHIPSET - Use Chipset Personality Module to implement watchdog controls.

OPTION_WATCHDOG_BOARD - Use Board Personality Module to implement watchdog controls.

OPTION_WATCHDOG_TIMER_KICK - Enable automatic kick of the dog on each timer tick inside the BIOS.

7.1.49 OPTION_SUPPORT_SOFT_ERR Option

The **OPTION_SUPPORT_SOFT_ERR** option enables or disables code in the BIOS that causes the POST to display errors on the screen that indicate correctable problems (these are not critical errors that result in beep codes or that cause Manufacturing Mode to take over). If this option is disabled, then the BIOS doesn't report these problems and simply corrects them without warning.

An example of a soft error encountered during POST would be a memory size mismatch; a difference between the amount of memory detected in physical memory scan, and the amount of memory as recorded in CMOS.

Soft errors are not displayed on the screen if **OPTION_SUPPORT_POSTMSGS** is disabled, because this option's purpose is to remove all messages from the display. Be certain that you enable **OPTION_SUPPORT_POSTMSGS** if you wish to see soft errors.

Soft errors may cause the Setup screen system to be invoked by enabling **OPTION_SOFTERR_SETUP**. If this option is disabled, then the Setup screen will not be invoked on soft errors.

One soft error, the CMOS memory size mismatch, can be enabled or disabled with the **OPTION_SOFTERR_MEMMIS** option. On earlier PC/AT clones, the actual size of memory was compared against the prerecorded size in CMOS, and if these sizes didn't match, a soft error occurred. This was primarily intended to catch cases where the machine was reconfigured, to allow the user to edit the configuration with the Setup system. Rarely is this behavior actually needed in modern systems.

Soft errors are subject to the same I/O redirection as standard INT 10h I/O; therefore, you can change **CONFIG_CON_REDIR_STD** to a COM port number to redirect the I/O over a serial line. If this is done, then all further application I/O (and that from Embedded DOS-ROM) will be routed over the same I/O port.

Values:

1 - Enable soft errors during POST.
0 - Disable soft errors during POST.

Related Parameters:

OPTION_SUPPORT_POSTMSGS - Enable POST messages.
OPTION_SUPPORT_CMOS - Enable CMOS RAM support.
OPTION_SOFTERR_MEMMIS - Cause soft error if memory size mismatch.
OPTION_SOFTERR_SETUP - Enter Setup system if soft error occurs.
CONFIG_CON_REDIR_STANDARD - Console I/O redirection for standard I/O.

7.1.50 OPTION_SUPPORT_MINI_DOS Option

The **OPTION_SUPPORT_MINI_DOS** option enables or disables code in the BIOS that causes the BIOS to be aware that Embedded DOS-ROM is available in ROM, and therefore to initialize it. (The term, Mini-DOS, is sometimes used to refer to Embedded DOS-ROM because of its small footprint: in some cases, less than 32KB).

The Embedded DOS-ROM segment address is specified with the **CONFIG_MINI_DOS_SCAN** parameter; normally, this value is E000h to be compatible with its distant ancestor, ROM BASIC. The selection of this ROM scan address causes it to be excluded from the general ROM BIOS extension scan so that it is only called once during POST.

Embedded DOS-ROM determines, when initialized, whether to hook the INT 19h vector or the INT 18h vector. If it hooks INT 19h, then it will become the primary operating system. If it hooks INT 18h, then it becomes a backup operating system in case the BIOS is unable to load an operating system from the default boot drive.

Values:

1 - Enable Embedded DOS-ROM ROM scan.
0 - Disable Embedded DOS-ROM ROM scan.

Related Parameters:

CONFIG_MINI_DOS_SCAN - Specify address of Embedded DOS-ROM ROM BIOS extension.

7.1.51 OPTION_SUPPORT_EXHMEMTEST Option

The **OPTION_SUPPORT_EXTMEMTEST** option enables or disables code in the BIOS that provides an exhaustive memory test that can be called during POST, during Manufacturing Mode, or from the Standard Diagnostics in the Setup system.

Exhaustive memory tests basically perform an analysis of every word in the tested range of RAM, and for each word, every bit is tested. Thus, the exhaustive memory test takes much longer than the standard memory test, but it finds problems that the standard memory test can't find, such as data lines wired together or address aliasing.

Enabling the **OPTION_SUPPORT_EXHMEMTEST** does not instruct the BIOS to start using the tests over the standard ones; this is accomplished with the following additional options.

OPTION_MEMTEST_LOW_POST is enabled to exhaustively scan low memory during POST, instead of the standard (quicker) scan.

OPTION_MEMTEST_HIGH_POST is enabled to exhaustively scan extended memory during POST, instead of the standard (very quick) scan.

OPTION_MEMTEST_WAIT is enabled to cause POST to pause between block tests, so that the user has time to press the <ESC> key to bypass memory tests.

OPTION_MEMTEST_CLEAR is enabled to cause POST to rewrite low memory with a pattern of all 00h's, so that bugs in MS-DOS do not surface.

OPTION_MEMTEST_CLICK is enabled to cause POST to click the speaker between block tests, so that the user has an aural indication that progress is being made.

In addition to the above feature selectors, the exhaustive memory tests (and the standard memory tests) do not test beyond set limits, so that they do not begin accidently manipulating devices or Flash memory that immediately follows RAM areas to be tested. The following two parameters control these limits:

CONFIG_MAX_LOW_MEMORY defines the number of kilobytes (often, 640) that are to be tested for valid RAM to be used as low memory. This value can be raised or lowered, depending on how far you wish the memory scan to reach. A caution: raising it beyond the 640k limit will cause the memory tests to test video memory on a VGA card in graphics mode successfully, which will result in a system crash when the video board is actually used.

CONFIG_MAX_EXT_MEMORY defines the number of kilobytes of extended memory to be tested for valid RAM, for the same reasons.

Values:

1 - Enable exhaustive memory tests.
0 - Disable exhaustive memory tests.

Related Parameters:

OPTION_CMOS_EXHMEMTEST – Factory default for enabling/disabling exhaustive memory test in CMOS Setup.

OPTION_MEMTEST_LOW_POST - Exhaustively test low memory during POST.

OPTION_MEMTEST_HIGH_POST - Exhaustively test high memory during POST.

OPTION_MEMTEST_WAIT - Pause between testing blocks.

OPTION_MEMTEST_CLEAR - Clear low memory to a field of 00h's for MS-DOS.

OPTION_MEMTEST_CLICK - Click the speaker after testing each block.

OPTION_MEMTEST_QUICK – Test only the first word of each 1KB block.

CONFIG_MAX_LOW_MEMORY - Maximum low memory size in KB.

CONFIG_MAX_EXT_MEMORY - Maximum extended memory size in KB.

7.1.52 OPTION_SUPPORT_KNOWN_ENTRYPOINTS Option

The **OPTION_SUPPORT_KNOWN_ENTRYPOINTS** option enables or disables the special entrypoints at specific hard-coded addresses in the BIOS so that older VGA BIOS extensions, application programs, and other software, can call the BIOS service routines directly without using an INT instruction to do the work.

These entrypoints span a range of 8KB of ROM in the top 64KB of the BIOS, so enabling this option causes the size of the BIOS to grow by roughly 8KB without a large functional benefit.

Related to this option is **OPTION_SUPPORT_IBM_COMPAT**, which causes a special compatibility string to be inserted at F000:E000, so that certain utility programs can detect the BIOS as IBM-compatible. This option also wastes space.

Values:

1 - Enable hard-coded entrypoints.
0 - Disable hard-coded entrypoints.

Related Parameters:

OPTION_SUPPORT_IBM_COMPAT - Enable IBM-compatibility string.

7.1.53 OPTION_SUPPORT_IBM_COMPAT Option

The **OPTION_SUPPORT_IBM_COMPAT** option enables or disables the special "IBM IS A REGISTERED TRADEMARK OF INTERNATIONAL BUSINESS MACHINES CORPORATION" string at offset E000h in the BIOS. This string is examined by utility programs to determine if a desktop BIOS is IBM-compatible.

Certainly, a BIOS need not contain this string in order to provide work-alike functionality to its distant IBM relative.

Values:

1 - Enable IBM string.
0 - Disable IBM string.

Related Parameters:

OPTION_SUPPORT_KNOWN_ENTRYPOINTS - Enable backdoor entrypoints into BIOS.

7.1.54 OPTION_SUPPORT_MFGMODE Option

The **OPTION_SUPPORT_MFGMODE** option enables or disables the Manufacturing Mode protocol engine that provides a host with remote access to EMBEDDED BIOS facilities on the target.

Manufacturing Mode can be entered from the SETUP system by enabling **OPTION_SETUP_MFGMODE**.

Manufacturing Mode can also be entered during POST if a critical error is encountered by enabling both **OPTION_CRITICAL_BOARD** and **OPTION_MFGMODE_CRITICAL**. The default code in the Board Personality Module invokes the Manufacturing Mode routine.

A third way to enter Manufacturing Mode is to test a special hardware device called a test mode pin. This test mode pin's actual physical assignment may be a line on an unused UART, such as Carrier Detect, for example. By providing code other than the default code in the Board Personality Module's **BoardTestMode** routine, a special OEM-defined hardware circuit can be interrogated to determine if Manufacturing Mode should be entered, or if the operating system should continue to boot.

You can cause Manufacturing Mode to constantly check the status of your test mode pin so that when the pin goes low, Manufacturing Mode exits and boots the operating system. This also enables the target to wait for a timeout period (a couple seconds) until the test mode pin goes active. To enable the timeout option, enable **OPTION_MFGMODE_TIMEOUT**.

If Flash is to be programmed during Manufacturing Mode, then **CONFIG_FLASH_DATASEG** must be set to a segment address of a 64KB buffer that will be used as a temporary staging buffer for data coming over the RS232 link under the host program's control.

During Flash programming, the Manufacturing Mode code automatically copies the entire BIOS into RAM at the segment address specified by **CONFIG_FLASH_CODESEG**, so that it can update the Flash containing the BIOS, if necessary.

For debugging the Manufacturing Mode on hardware that provides 7-segment hex readouts of POST codes, or other similar hardware, you can enable **OPTION_SUPPORT_MFGCODES**, and then set **CONFIG_MFG_PROGRESS_PORT** to the 8-bit I/O port address that Manufacturing Mode commands should be copied to for visual inspection. This port is written with the value 0ffh when no commands are being executed.

Values:

1 - Enable support for Manufacturing Mode.
0 - Disable support for Manufacturing Mode.

Related Parameters:

OPTION_SETUP_MFGMODE - Enable SETUP screen option to enter Manufacturing Mode.

OPTION_MFGMODE_CRITICAL - Call the Board Personality Module's critical error handler if a POST critical error occurs, such as a RAM parity error, or other hardware fault.

OPTION_MFGMODE_TIMEOUT - Time-out the Manufacturing Mode if the test-mode pin goes inactive.

OPTION_CRITICAL_BOARD - Enable path to Manufacturing Mode from critical errors passed to Chipset Personality Module.

OPTION_SUPPORT_MFGCODES - Enable Manufacturing Mode progress codes to be written to the port defined by **CONFIG_MFG_PROGRESS_PORT**.

CONFIG_FLASH_DATASEG - Segment address of Manufacturing Mode staging buffer.

CONFIG_FLASH_CODESEG - Segment address of Manufacturing Mode scratch code buffer.

CONFIG_MFG_PROGRESS_PORT - I/O port used to write Manufacturing Mode progress codes to.

7.1.55 OPTION_SUPPORT_PARITY Option

The **OPTION_SUPPORT_PARITY** option enables or disables code in the BIOS that supports parity checking.

Parity support is used to generate an NMI interrupt when a RAM or I/O parity errors. This requires **OPTION_SUPPORT_PORT_B** to be enabled, as the parity control bits are defined in PORT B.

If RAM parity is to be supported, then **OPTION_MEMTEST_CLEAR** must be enabled. This is required because memory will have indeterminate contents (and therefore indeterminate parity) if it boots without being initialized to some value. Note that both low memory and extended memory must be initialized so that memory parity errors do not occur in uninitialized extended memory.

Values:

1 - Enable parity checking support.
0 - Disable parity checking support.

Related Parameters:

OPTION_SUPPORT_PORT_B - Support PORT B architecture.

OPTION_MEMTEST_CLEAR - Initialize all of low memory to a field of 00h's.

7.1.56 OPTION_SUPPORT_PASSWORD Option

The **OPTION_SUPPORT_PASSWORD** option enables or disables code in the BIOS that supports password checking during POST.

If this option is enabled, and a password has been entered into CMOS via the SETUP screen system's SET PASSWORD main menu item, then the target will require the user to enter a password before allowing the target to boot an operating system.

Password checking happens after SETUP runs, and therefore does not affect Manufacturing Mode or the integrated BIOS debugger.

In order for this option to be useful, the **OPTION_SETUP_PASSWORD** option must be enabled, so that the user can enter a new password.

Values:

1 - Enable password checking support.
0 - Disable password checking support.

Related Parameters:

OPTION_SETUP_PASSWORD - Support password entry in SETUP screen system.

7.1.57 OPTION_SUPPORT_DEMO Option

The **OPTION_SUPPORT_DEMO** option enables or disables a timeout in the BIOS that causes the BIOS to stop running the system. The demo timeout happens approximately one hour after cold boot.

Values:

1 - Enable demo timeout.
0 - Disable demo timeout.

Related Parameters:

None.

7.1.58 OPTION_SUPPORT_DEMO_MSG Option

The **OPTION_SUPPORT_DEMO_MSG** option enables or disables code in the BIOS that displays messages during POST, indicating that the BIOS is an unlicensed demonstration version. Additional messages are displayed which tell the user how to contact General Software, Inc., for licensing details.

Values:

1 - Enable demonstration BIOS messages in POST.
0 - Disable demonstration BIOS messages in POST.

Related Parameters:

OPTION_SUPPORT_DEMO - Enable demonstration timeout in BIOS build.

7.1.59 OPTION_SUPPORT_ATA Option

The **OPTION_SUPPORT_ATA** option enables or disables the special dedicated ATA mode for a Cirrus Logic 6710 or 6720 controller. This mode programs the controller into a special mode whereby ATA PC Cards inserted into the socket are treated as IDE drives by the EMBEDDED BIOS IDE software.

Values:

1 - Enable dedicated ATA mode.
0 - Disable dedicated ATA mode.

Related Parameters:

CONFIG_PCMCIA_IOBASE - Specifies I/O base address for CL 67x0 controller.

7.1.60 OPTION_SUPPORT_CON_REDIRECTOR Option

The **OPTION_SUPPORT_CON_REDIRECTOR** option enables or disables the internal core BIOS support for routing video output from INT 10h and keyboard input from INT 16h over RS-232 ports. Three different channels of console I/O are supported: POST and DOS, BIOS debugger, and Setup screen system. Each channel can have a separate routing assignment.

The actual assignment of each channel is governed by a separate configuration parameter. The values for each of these parameters specifies the COM port number to route the I/O over, or zero (0) if I/O should be routed over the traditional keyboard and screen devices. The **CONFIG_CON_REDIR_STD** parameter governs BIOS POST and DOS I/O. The **CONFIG_CON_REDIR_DEBUG** parameter governs BIOS debugger I/O. The **CONFIG_CON_REDIR_SETUP** parameter governs BIOS Setup screen I/O.

The **OPTION_SERIAL_9600_BAUD** parameter can be set when running MS-DOS or another operating system that insists on resetting the COM port baud rates to its own value. For example, MS-DOS initializes all INT 14h serial ports to 2400 baud, even parity, and 1 stop bit. When run over a serial port, MS-DOS appears to start booting, and then crash, when in fact all it has done is change the serial port's baud rate so that it cannot properly communicate with the host terminal software. The solution is to set this parameter so that when MS-DOS attempts to change the baud rate, the BIOS ignores the request and sets it to 9600 baud.

Note that if you are using a VGA BIOS extension in your system at the same time as console redirection, the VGA BIOS will probably hook INT 10h and prevent the core BIOS's console redirection code from being called at all. Therefore, **OPTION_SUPPORT_VIDEO_BOARDS** may need to be disabled for these systems.

If you want the system to use a VGA BIOS in the system if present, and use console redirection if the VGA BIOS is not present, you'll need to do a couple things. First, you need to disable **OPTION_HARDERR_VIDEO** so that if the core BIOS can't find the video RAM on a nonexistant card, it will continue through POST. Second, you'll need to place some custom OEM code in your board module, perhaps routine **BoardInit4**, to check for the existence of the 55h/aah VGA BIOS signature, and if not present, perform an INT 15h that redirects console I/O to the serial port of your choice.

EMBEDDED BIOS also has an autoredirection feature built into the core; it may be enabled with the **OPTION_CON_REDIR_AUTO** configuration option. See the BPM routine **BoardAutoRedirect** for details about how to change the autoredirection policy.

Values:

1 - Enable console redirection.
0 - Disable console redirection.

Related Parameters:

OPTION_SERIAL_9600_BAUD - Force 9600 baud when MS-DOS resets UARTs.

OPTION_CON_REDIR_WAIT – Wait for TBE before outputting characters.

OPTION_CON_REDIR_DISABLE – Disable redirection on output timeout.

OPTION_CON_REDIR_CANCEL – Cancel redirection if main console keypress detected.

OPTION_CON_REDIR_AUTO – Enable autoredirection of console I/O.

CONFIG_CON_REDIR_STD - I/O assignment for BIOS POST and DOS.

CONFIG_CON_REDIR_DEBUG - I/O assignment for BIOS debugger.

CONFIG_CON_REDIR_SETUP - I/O assignment for BIOS Setup system.

7.1.61 OPTION_SUPPORT_MCL Option

The **OPTION_SUPPORT_MCL** option enables or disables the Media Control Layer (MCL) component of the BIOS when it is built. Normally, MCL is enabled automatically by the **MEDIA_REGION** macro as table entries are defined. One special entry, covering the entire media address space, is defined by default, and the MCL is therefore normally assembled even when no OEM-specified **MEDIA_REGION** table entries are defined. Disabling this option forces the MCL to be not included in the system.

Values:

1 - Enable Media Control Layer.
0 - Disable Media Control Layer.

Related Parameters:

MEDIA_REGION - Defines entries in the media region table.

7.1.62 OPTION_SUPPORT_DISKIO Option

The **OPTION_SUPPORT_DISKIO** option enables or disables the File System Control Layer (FSCL) component of the BIOS when it is built. Normally, FSCL is enabled automatically by the **FILE_SYSTEM** macro as table entries are defined. Disabling this option causes the entire INT 13h disk services subsystem, including the drivers, to be removed from the system.

Values:

1 - Enable Disk I/O Support.
0 - Disable Disk I/O Support.

Related Parameters:

FILE_SYSTEM - Defines entries in the file system table.

7.1.63 OPTION_SUPPORT_WINCE Option

The **OPTION_SUPPORT_WINCE** option enables or disables support in the BIOS for booting Windows CE binaries, either from disk or directly from ROM or Flash.

If this option is enabled, and then if **OPTION_CMOS_LOAD_WINCE** is enabled, the BIOS will attempt to load the Windows CE binary (NK.BIN) from the root directory of devices instead of the boot record that loads a traditional operating system such as DOS. If the Windows CE binary is not present on the disk, then the boot record is loaded and executed.

If this option is disabled, then the BIOS will simply load the boot record into memory and transfer control to it. This feature is called "CE Ready."

This is not the same feature as the boot option to run Windows CE directly from ROM or Flash in the extended memory address space. That is accomplished by setting **CONFIG_CMOS_BOOT_*n*** to **BOOT_WINCE**.

Values:

1 - Enable CE Ready Support.
0 - Disable CE Ready Support.

Related Parameters:

OPTION_CMOS_LOAD_WINCE - Define default action when booting from a drive.

7.1.64 OPTION_SUPPORT_BOOT_FAR Option

The **OPTION_SUPPORT_BOOT_FAR** option controls the format of the first instruction in the BIOS executed by the CPU. If this parameter is enabled, then the first instruction will be a FAR jump instruction, causing the CS register to be reloaded according to real-mode rules (this is the default). Some CPUs require that registers be programmed before the first reload of CS (for example, chip selects). If this is the case, then this option should be disabled, causing the first instruction to be executed to be a NEAR jump instead.

If this option is disabled, then CS remains untouched until after the **BOARD_PREPOST** macro runs, but before the mainline POST runs. This allows the BPM's macro to get control to perform any special processing before CS gets reloaded.

Values:

1 - Enable FAR jump as first instruction.
0 - Disable FAR jump as first instruction.

Related Parameters:

None.

7.1.65 OPTION_SUPPORT_BIOS32 Option

The **OPTION_SUPPORT_BIOS32** option enables or disables the 32-bit components of the chipset and board personality modules, so that they can support the 32-bit BIOS build.

Values:

1 - Enable 32-bit code in board and chipset modules.
0 – Disable 32-bit code in board and chipset modules.

Related Parameters:

None.

7.1.66 OPTION_SUPPORT_SPLASHSCR Option

The **OPTION_SUPPORT_SPLASHSCR** option enables or disables the graphical front-end of the BIOS, including splash screen and graphic icon management and display code.

This option needs to be enabled, along with **OPTION_SUPPORT_EXTRES**, for any **SPLASH_TABLE** entries to be meaningful.

Values:

1 - Enable graphical front-end support.
0 – Disable graphical front-end support.

Related Parameters:

OPTION_SUPPORT_EXTRES - Enable External Resource Manager support.
CONFIG_SPLASH_VMODE – Specify video mode for graphica front-end.
CONFIG_SPLASH_WIDTH – Specify display device width in pixels.
CONFIG_SPLASH_WBYTES – Specify video frame buffer width in bytes.
CONFIG_SPLASH_HEIGHT – Specify video display height in raster lines.
CONFIG_SPLASH_COLORS – Specify number of colors supported by video mode.
CONFIG_SPLASH_SEG – Specify segment address for graphics workspace.
CONFIG_SPLASH_BOOTS – Specify limit for booting with disabled splash screen.

SPLASH_TABLE – Specify graphic resources to be used.

7.1.67 OPTION_SUPPORT_EXTRES Option

The **OPTION_SUPPORT_EXTRES** option enables or disables the external resource manager, which provides APIs for finding and extracting external resources bound with the final BIOS image by the GSMERGE utility.

Resources can be any separately-prepared binary component, including relocatable code, bitmaps, application data, and so on.

Values:

1 - Enable external resource manager.
0 –Disable external resource manager.

Related Parameters:

None.

7.1.68 OPTION_SUPPORT_INT13_EXTENSIONS Option

The **OPTION_SUPPORT_INT13_EXTENSIONS** option enables or disables the support in the File System Control Layer and IDE/ATA file system drivers for handling the industry-standard, extended INT 13h functions.

These functions are used by some operating systems, such as Windows NT, that require access to data on a hard drive that is greater than 8GB in size.

Values:

1 - Enable INT 13h extensions.
0 –Disable INT 13h extensions.

Related Parameters:

None.

7.1.69 OPTION_SETUP_CUSTOM Option

The **OPTION_SETUP_CUSTOM** option enables or disables code in the BIOS to display the Custom Configuration SETUP screen, which is supported by the Board Personality Module (BPM).

In order for this option to be effective, **OPTION_SUPPORT_SETUP** must be enabled (see that section for more options and details).

This feature should only be selected if the adaptation engineer has selected a Board Personality Module (BPM) and has defined setup fields within the module that allow the end-user to configure the chipset during Setup. See Chapter 20 for more details.

Values:

1 - Enable Custom SETUP screen.
0 - Disable Custom SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.

7.1.70 OPTION_SETUP_DEMO Option

The **OPTION_SETUP_DEMO** option enables or disables code in the BIOS to display the Demonstration SETUP screen, normally only used by General Software for adaptations of EMBEDDED BIOS for reference designs distributed by silicon vendors.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** must be enabled (see that section for more options and details).

Values:

1 - Enable Demo SETUP screen.
0 - Disable Demo SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.

7.1.71 OPTION_SETUP_PASSWORD Option

The **OPTION_SETUP_PASSWORD** option enables or disables code in the BIOS to display the Password Configuration SETUP screen.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** must be enabled (see that section for more options and details).

OPTION_SUPPORT_PASSWORD must also be enabled in order for this option to be useful. This option controls whether POST will check the password that is entered from the SETUP screen.

Values:

1 - Enable password configuration SETUP screen.
0 - Disable password configuration SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_PASSWORD - Enable password checking in POST.

7.1.72 OPTION_SETUP_DIAGNOSTICS Option

The **OPTION_SETUP_DIAGNOSTICS** option enables or disables code in the BIOS to support the Standard Diagnostics SETUP screen.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** must be enabled (see that section for more options and details).

Some portions of the Standard Diagnostics suite may be enabled or disabled, depending on various other options you have enabled in the adaptation. There are so many of these options, that they cannot be listed individually here. If "No Hdwr" is present on a particular test that you wish to perform, its corresponding option may need to be configured properly in the project file.

The diagnostics suite is extensive, and considerable code space is used for its implementation. If your adaptation is running short on space, disabling this option can save space that can be used for other functions.

Values:

1 - Enable standard diagnostics SETUP screen.
0 - Disable standard diagnostics SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.

7.1.73 OPTION_SETUP_DEBUGGER Option

The **OPTION_SETUP_DEBUGGER** option enables or disables code in the BIOS to support the Debugger SETUP option. This allows the user to enter the integrated BIOS debugger from SETUP's main menu without having to press the special Ctl-Left-Shift keys together.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_DEBUGGER** must be enabled (see those sections for more options and details).

Values:

1 - Enable Debugger SETUP option.
0 - Disable Debugger SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

7.1.74 OPTION_SETUP_IDE Option

The **OPTION_SETUP_IDE** option enables or disables code in the BIOS to support the OEM-extensible IDE Utility SETUP screen. This screen is intended for OEM expansion, so that special setup and diagnostics can be performed on special drives.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_DISKIO** must be enabled, and a **FILE_SYSTEM** table entry must be specified for the IDE driver.

Values:

1 - Enable IDE Utility SETUP screen.
0 - Disable IDE Utility SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_DISKIO - Enable disk support.
FILE_SYSTEM - Enable specific disk driver.

7.1.75 OPTION_SETUP_SHADOW Option

The **OPTION_SETUP_SHADOW** option enables or disables code in the BIOS to support the ROM Shadowing SETUP screen.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** must be enabled (see that section for more options and details).

ROM shadowing is enabled with the **OPTION_SUPPORT_SHADOW** option, but the shadowing work is performed in the Chipset Personality Module. All adaptations requiring ROM shadowing require that shadowing code be implemented by the OEM in the Board Personality Module or the Chipset Personality Module. Therefore, both **OPTION_SUPPORT_SHADOW** and **OPTION_SUPPORT_CHIPSET** must be enabled to support shadowing.

Values:

1 - Enable ROM Shadowing SETUP screen.
0 - Disable ROM Shadowing SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_SUPPORT_CHIPSET - Enable chipset support.

7.1.76 OPTION_SETUP_PWR_FEATURES Option

The **OPTION_SETUP_PWR_FEATURES** option enables or disables code in the BIOS to support the Power Management Features SETUP screen.

The power management features supported by the setup screen are dictated by the power management device tree built in the project file by the OEM using the **POWER_DEVID** macro.

This screen does not address timeouts for devices, which are specified on the Power Management Timeouts SETUP screen (see **OPTION_SETUP_PWR_TIMEOUTS**).

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_POWERMAN** must be enabled (see those sections for more options and details).

Power management is provided by the Chipset Personality Module, the CPU Personality Module, and/or the Board Personality Module. Thus, these modules must contain power management code in order for the core BIOS to perform actual power management with the available hardware.

Values:

1 - Enable Power Management Features SETUP screen.
0 - Disable Power Management Features SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_POWERMAN - Enable power management support.
OPTION_SETUP_PWR_TIMEOUTS - Enable Power Management timeouts screen.

7.1.77 OPTION_SETUP_PWR_TIMEOUTS Option

The **OPTION_SETUP_PWR_TIMEOUTS** option enables or disables code in the BIOS to support the Power Management Timeouts SETUP screen.

The power management timeouts supported by the setup screen are dictated by the power management device tree built in the project file by the OEM using the **POWER_DEVID** macro.

This screen does not address device features, which are specified on the Power Management Features SETUP screen (see **OPTION_SETUP_PWR_FEATURES**).

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_POWERMAN** must be enabled (see those sections for more options and details).

Power management is provided by the Chipset Personality Module, the CPU Personality Module, and/or the Board Personality Module. Thus, these modules must contain power management code in order for the core BIOS to perform actual power management with the available hardware.

Values:

1 - Enable Power Management Timeouts SETUP screen.
0 - Disable Power Management Timeouts SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_POWERMAN - Enable power management support.
OPTION_SETUP_PWR_FEATURES - Enable Power Management features screen.

7.1.78 OPTION_SETUP_MFGMODE Option

The **OPTION_SETUP_MFGMODE** option enables or disables code in the BIOS to support the SETUP option that allows the user to enter Manufacturing Mode.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_MFGMODE** must be enabled (see those sections for further details).

Values:

1 - Enable Manufacturing Mode SETUP option.
0 - Disable Manufacturing Mode SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

7.1.79 OPTION_SETUP_RAMDISK Option

The **OPTION_SETUP_RAMDISK** option enables or disables code in the BIOS to support the SETUP option that can reformat the core BIOS RAM disk.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_DISKIO** must be enabled, and the RAM disk defined with a **FILE_SYSTEM** table entry.

Values:

1 - Enable RAM disk formatting SETUP option.
0 - Disable RAM disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_DISKIO - Enable disk support.
FILE_SYSTEM - Enable specific disk driver.

7.1.80 OPTION_SETUP_RFDDISK Option

The **OPTION_SETUP_RFDDISK** option enables or disables code in the BIOS to support the SETUP option that can low-level format the core BIOS RFD disk.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_DISKIO** must be enabled, and the RFD disk defined with a **FILE_SYSTEM** table entry.

Values:

1 - Enable RFD disk formatting SETUP option.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_DISKIO - Enable disk support.
FILE_SYSTEM - Enable specific disk driver.

7.1.81 OPTION_SETUP_SHAD_C000 Option

The **OPTION_SETUP_SHAD_C000** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment C000h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.82 OPTION_SETUP_SHAD_C400 Option

The **OPTION_SETUP_SHAD_C400** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment C400h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.83 OPTION_SETUP_SHAD_C800 Option

The **OPTION_SETUP_SHAD_C800** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment C800h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.84 OPTION_SETUP_SHAD_CC00 Option

The **OPTION_SETUP_SHAD_CC00** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment CC00h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.85 OPTION_SETUP_SHAD_D000 Option

The **OPTION_SETUP_SHAD_D000** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment D000h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.86 OPTION_SETUP_SHAD_D400 Option

The **OPTION_SETUP_SHAD_D400** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment D400h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.87 OPTION_SETUP_SHAD_D800 Option

The **OPTION_SETUP_SHAD_D800** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment D800h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.88 OPTION_SETUP_SHAD_DC00 Option

The **OPTION_SETUP_SHAD_DC00** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment DC00h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.89 OPTION_SETUP_SHAD_E000 Option

The **OPTION_SETUP_SHAD_E000** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment E000h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.90 OPTION_SETUP_SHAD_E400 Option

The **OPTION_SETUP_SHAD_D400** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment E400h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.91 OPTION_SETUP_SHAD_E800 Option

The **OPTION_SETUP_SHAD_E800** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment E800h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.92 OPTION_SETUP_SHAD_EC00 Option

The **OPTION_SETUP_SHAD_EC00** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment EC00h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.93 OPTION_SETUP_SHAD_F000 Option

The **OPTION_SETUP_SHAD_F000** option enables or disables support for a shadowable region in the SETUP screen. EMBEDDED BIOS's SETUP screen provides for 16KB granularity from segment C000h through EFFFh, and then a 64KB segment at F000h. Some chipsets may not be capable of shadowing at this resolution (perhaps at 32KB or 64KB increments instead).

If this option is enabled, then the Shadowing SETUP screen's entry for the specified segment will allow the user to enable or disable shadowing for that segment. If this option is disabled, then the SETUP screen will not allow the user to enable or disable shadowing for that region.

In order for this option to be effective, **OPTION_SUPPORT_SETUP** and **OPTION_SUPPORT_SHADOW** must be enabled.

Values:

1 - Enable shadow option at segment F000h.
0 - Disable RFD disk formatting SETUP option.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP menu.
OPTION_SUPPORT_SHADOW - Enable shadow memory support.

7.1.94 OPTION_REFRESH_8237 Option

The **OPTION_REFRESH_8237** option enables or disables code in the BIOS to support DRAM refresh by linking the primary 8237A DMA controller together with the 8254 programmable interrupt timer. This is the standard method used on the IBM PC, PC/XT, and PC/AT systems.

To use **OPTION_REFRESH_8237**, you must also enable **OPTION_SUPPORT_8237** and **OPTION_SUPPORT_8254** to specifically enable the drivers for the 8237 and 8254.

In most newer system designs, this refresh mechanism has been replaced with a DRAM refresh controller either on the CPU or in the chipset. If you are using CPU refresh, you need to select the **CPUCLASS** for the CPU type, and then enable **OPTION_REFRESH_CPU** instead. If you are using chipset refresh, enable **OPTION_SUPPORT_CHIPSET** and **OPTION_REFRESH_CHIPSET**. If you have a board that requires any refreshing mechanism that is not included in the standard 8237A, CPU, or chipset methods, then the **OPTION_REFRESH_BOARD** option should be enabled.

If the chipset, CPU, and/or board refreshing methods are selected, then code must be provided in the associated personality module's refresh control routines. If all the code is already supplied by General Software, then additional coding is not necessary.

When this option is selected, the other refresh options must be disabled. When refreshing DRAM, you may also enable **OPTION_REFRESH_CHARGE**, which causes low memory to be alternately written with 1's and 0's after refresh has been enabled. This was required for the IBM PC and PC/XT, but is not required for most modern hardware.

Values:

1 - Enable DRAM refresh using the 8237A and 8254.
0 - Disable DRAM refresh using the 8237A and 8254.

Related Parameters:

OPTION_SUPPORT_REFRESH - Enable refresh logic.

OPTION_SUPPORT_8237 - Enable 8237 DMA support.

OPTION_SUPPORT_8254 - Enable 8254 timer support.

OPTION_REFRESH_CHIPSET - Support refresh through chipset controller.

OPTION_REFRESH_CPU - Support refresh through CPU integrated refresh controller.

OPTION_REFRESH_BOARD - Support refresh though board module.

OPTION_REFRESH_CHARGE - Optionally charge the DRAMs in an IBM PC so that refresh works properly.

7.1.95 OPTION_REFRESH_CHIPSET Option

The **OPTION_REFRESH_CHIPSET** option enables or disables code in the BIOS to support DRAM refresh by programming the chipset to use its own decoupled DRAM refresh mechanism.

When this option is selected, the other refresh options must be disabled. This option does not make use of the 8237A or 8254 hardware.

To use this option, you must enable **OPTION_SUPPORT_CHIPSET**, and set **CHIPSET** to a chipset identifier that supports refreshing.

Values:

1 - Enable DRAM refresh via chipset.
0 - Disable DRAM refresh via chipset.

Related Parameters:

OPTION_SUPPORT_REFRESH - Enable refresh logic.

OPTION_SUPPORT_CHIPSET - Enable Chipset Personality Module.

OPTION_REFRESH_8237A - Support refresh through chipset controller.

OPTION_REFRESH_CPU - Support refresh through CPU integrated refresh controller.

OPTION_REFRESH_BOARD - Support refresh through board module.

OPTION_REFRESH_CHARGE - Optionally charge the DRAMs in an IBM PC so that refresh works properly.

CHIPSET - Select Chipset Personality Module.

7.1.96 OPTION_REFRESH_CPU Option

The **OPTION_REFRESH_CPU** option enables or disables code in the BIOS to support DRAM refresh by programming the CPU to use its own internal DRAM refresh controller.

When this option is selected, the other refresh options must be disabled. This option does not make use of the 8237A or 8254 hardware, nor does it use a chipset for support.

To use this option, you must also set **CPUCLASS** to the identifier associated with the CPU that you are using, and ensure that the CPU Personality Module supports refreshing via the CPU DRAM refresh controller.

Values:

1 - Enable DRAM refresh via CPU.
0 - Disable DRAM refresh via CPU.

Related Parameters:

OPTION_SUPPORT_REFRESH - Enable refresh logic.

OPTION_REFRESH_8237A - Support refresh through chipset controller.

OPTION_REFRESH_CHIPSET - Support refresh through chipset integrated refresh controller.

OPTION_REFRESH_BOARD - Support refresh through board module.

OPTION_REFRESH_CHARGE - Optionally charge the DRAMs in an IBM PC so that refresh works properly.

CPUCLASS - Select CPU Personality Module.

7.1.97 OPTION_REFRESH_BOARD Option

The **OPTION_REFRESH_BOARD** option enables or disables code in the BIOS to support DRAM refresh by calling the Board Personality Module's DRAM refresh support routines (which may in turn call CPU or chipset routines, or perform OEM-proprietary actions.

When this option is selected, the other refresh options must be disabled. This option does not make use of the 8237A or 8254 hardware, nor does it use a chipset for support.

To use this option, you must also set **BOARD** to the identifier associated with the Board Personality Module that you are using, and ensure that the CPU Personality Module supports refreshin.

Values:

1 - Enable DRAM refresh via board module.
0 - Disable DRAM refresh via board module.

Related Parameters:

OPTION_SUPPORT_REFRESH - Enable refresh logic.

OPTION_REFRESH_8237A - Support refresh through chipset controller.

OPTION_REFRESH_CHIPSET - Support refresh through chipset integrated refresh controller.

OPTION_REFRESH_CPU - Support refresh through CPU module.

OPTION_REFRESH_CHARGE - Optionally charge the DRAMs in an IBM PC so that refresh works properly.

BOARD - Select Board Personality Module.

7.1.98 OPTION_REFRESH_CHARGE Option

The **OPTION_REFRESH_CHARGE** option enables or disables code in the BIOS to support charging of DRAM cells alternately with 0's and 1's to make them operational after DRAM refresh has started.

This was only necessary on early IBM PC systems, but may be enabled if refresh appears to be accessing the DRAM chips but from a software standpoint, appears not to work properly.

This option cannot be used alone. It must be used in conjunction with a refresh method, described in earlier sections.

Values:

1 - Enable charging of DRAMs.
0 - Disable charging of DRAMs.

Related Parameters:

OPTION_SUPPORT_REFRESH - Enable DRAM refresh support.

7.1.99 OPTION_DMA_8237 Option

The **OPTION_DMA_8237** option enables or disables code in the BIOS to support the routing of DMA requests to external 8237A DMA controllers.

In PC and PC/XT designs, only one 8237A is used, so **OPTION_SUPPORT_8237** must be enabled when using this option.

In PC/AT systems and beyond, two 8237A's are normally used together. If you have a target that is similar to a PC/AT or is a 386 system or beyond, you should enable both **OPTION_SUPPORT_8237** and **OPTION_SUPPORT_8237_2**.

If you have an Intel 386-EX design, note that the DMA controllers on the 386-EX CPU can closely imitate the 8237A and page register file, but not exactly. In particular, there are only 2 DMA channels on the 386-EX. Therefore, if you are using floppy disk I/O in a 386-EX design and desire DMA-based floppy I/O, you need to select **OPTION_DMA_CPU**, not **OPTION_DMA_8237**.

Values:

1 - Route DMA through 8237A.
0 - Don't route DMA through 8237A.

Related Parameters:

OPTION_SUPPORT_8237 - Enable primary 8237A support.
OPTION_SUPPORT_8237_2 - Enable secondary 8237A support.
OPTION_DMA_CPU - Support DMA via CPU integrated DMA controller.
OPTION_DMA_BOARD - Route DMA requests through Board Personality Module.

7.1.100 OPTION_DMA_CPU Option

The **OPTION_DMA_CPU** option enables or disables code in the BIOS to support the routing of DMA requests to an on-board DMA controller in the CPU.

A CPU Personality Module must be selected that supports CPU DMA operations in order for this option to be supported. Set **CPUCLASS** to the CPU type that is to be used so that the correct CPU Personality Module is enabled.

Floppy I/O is the only core BIOS subsystem that requires DMA for its operation (except for refresh in older systems). If you have CPU DMA support, this will not necessary interoperate with the floppy driver. Consult the section on **OPTION_SUPPORT_FLOPPY** for further details.

Values:

1 - Route DMA through CPU.
0 - Don't route DMA through CPU.

Related Parameters:

None.

7.1.101 OPTION_DMA_BOARD Option

The **OPTION_DMA_BOARD** option enables or disables code in the BIOS to route DMA requests through the Board Personality Module, allowing the OEM to intercept DMA calls and handle their dispatching to the CPU or chipset modules in a special way, or to manage the DMA process in an entirely proprietary way.

A Board Personality Module must be selected that supports DMA operations in order for this option to be supported. Set **BOARD** to the Board Personality Module to be used. The OEM should review the board module code (or the default code in `SYSTEM\BOARD.ASM`) in order to determine what needs to be coded in the Board Personality Module.

Values:

1 - Route DMA through board module.
0 - Don't route DMA through board module.

Related Parameters:

BOARD - Specify Board Personality Module.

7.1.102 OPTION_INT_8259 Option

The **OPTION_INT_8259** option enables or disables code in the BIOS to support the routing of interrupt management requests to one or more external 8259 interrupt controllers.

In PC and PC/XT designs, only one 8259 is used, so **OPTION_SUPPORT_8259** must be enabled when using this option.

In PC/AT systems and beyond, two 8259's are normally used together. If you have a target that is similar to a PC/AT or is a 386 system or beyond, you should enable both **OPTION_SUPPORT_8259** and **OPTION_SUPPORT_8259_2**.

Values:

1 - Route interrupts through external 8259.
0 - Don't route interrupts through external 8259.

Related Parameters:

OPTION_SUPPORT_8259 - Enable primary 8259 support.
OPTION_SUPPORT_8259_2 - Enable secondary 8259 support.

7.1.103 OPTION_INT_CPU Option

The **OPTION_INT_CPU** option enables or disables code in the BIOS to support the routing of interrupt management requests to an on-board CPU interrupt controller.

The CPU Personality Module must support the on-board interrupt controller functions for this option to be valid. Be sure to set the **CPUCLASS** parameter for the type of CPU you are using so that the CPU Personality Module is enabled.

Values:

1 - Route interrupts through CPU.
0 - Don't route interrupts through CPU.

Related Parameters:

CPUCLASS - Select CPU Personality Module.

7.1.104 OPTION_INT_BOARD Option

The **OPTION_INT_BOARD** option enables or disables code in the BIOS to support the routing of interrupt management requests through the Board Personality Module.

The Board Personality Module must contain code in its interrupt support entrypoints that handles interrupt requests from the core BIOS. The **BOARD** parameter must be set to the name of the Board Personality Module containing this support.

Values:

1 - Route interrupts through Board Personality Module.
0 - Don't route interrupts through Board personality Module.

Related Parameters:

BOARD - Select Board Personality Module.

7.1.105 OPTION_TIMER_8254 Option

The **OPTION_TIMER_8254** option enables or disables code in the BIOS to support timekeeping in the system with an 8253/8254 programmable interval timer.

The **OPTION_SUPPORT_8254** configuration option must be enabled for this option to be valid.

Values:

1 - Route timer management through 8254.
0 - Don't route timer management through 8254.

Related Parameters:

OPTION_SUPPORT_8254 - Enable 8254 support.

7.1.106 OPTION_TIMER_CPU Option

The **OPTION_TIMER_CPU** option enables or disables code in the BIOS to support timekeeping in the system with an on-board CPU programmable timer.

The CPU Personality Module must support the management of on-board CPU timers for this option to be valid. To select the correct CPU Personality Module, **CPUCLASS** must be properly specified.

Values:

1 - Route timer management through CPU.
0 - Don't route timer management through CPU.

Related Parameters:

CPUCLASS - Select CPU Personality Module.

7.1.107 OPTION_TIMER_BOARD Option

The **OPTION_TIMER_BOARD** option enables or disables code in the BIOS to support timekeeping in the system with code in the Board Personality Module.

The Board Personality Module must contain code in its timer support entrypoints that handles timer requests from the core BIOS. The **BOARD** parameter must be set to the name of the Board Personality Module containing this support.

Values:

1 - Route timer management through Board Personality Module.
0 - Don't route timer management through Board Personality Module.

Related Parameters:

BOARD - Select Board Personality Module.

7.1.108 OPTION_SOUND_8254_8255 Option

The **OPTION_SOUND_8254_8255** option enables or disables code in the BIOS that routes sound requests to 8254 and 8255 device drivers. This option must be enabled for these controllers to be programmed for sound support

If these controllers are not to be supported, but a high-integration CPU is to be used for sound support, then **OPTION_SOUND_CPU** or **OPTION_SOUND_BOARD** should be enabled instead to provide sound through those personality modules.

Values:

1 - Route sound requests through 8254 or 8255 drivers.
0 - Don't route sound requests through 8254 or 8255 drivers.

Related Parameters:

OPTION_SUPPORT_8254 - Enable 8254 support.

OPTION_SUPPORT_8255 - Enable 8255 support.

OPTION_SOUND_CPU - Use high-integration CPU for sound generation.

OPTION_SOUND_BOARD - Route sound requests through board module.

7.1.109 OPTION_SOUND_CPU Option

The **OPTION_SOUND_CPU** option enables or disables code in the BIOS to support sound generation with an on-board CPU programmable timer by routing requests through the CPU Personality Module.

The CPU Personality Module must support the management of on-board CPU timers for this option to be valid. To select the correct CPU Personality Module, **CPUCLASS** must be properly specified.

Values:

1 - Route sound generation through CPU.
0 - Don't route sound generation through CPU.

Related Parameters:

CPUCLASS - Select CPU Personality Module.

OPTION_SOUND_8254_8255 - Use 8254 or 8255 devices for sound generation.

7.1.110 OPTION_SOUND_BOARD Option

The **OPTION_SOUND_BOARD** option enables or disables code in the BIOS to support sound generation by routing requests through the Board Personality Module.

The Board Personality Module must contain code in its sound support entrypoints that handles sound requests from the core BIOS. The **BOARD** parameter must be set to the name of the Board Personality Module containing this support.

Values:

1 - Route sound generation through the Board Personality Module.
0 - Don't route sound generation through the Board Personality Module.

Related Parameters:

BOARD - Select Board Personality Module.

OPTION_SOUND_8254_8255 - Use 8254 or 8255 devices for sound generation.

OPTION_SOUND_CPU - Route sound requests to CPU Personality Module.

7.1.111 OPTION_WATCHDOG_CHIPSET Option

The **OPTION_WATCHDOG_CHIPSET** option enables or disables code in the BIOS to route watchdog timer requests to the Chipset Personality Module.

In order for this option to work, **OPTION_SUPPORT_CHIPSET** must be enabled, the **CHIPSET** type must be selected, and the Chipset Personality Module must be programmed to be capable of handling watchdog timer requests.

Values:

1 - Route watchdog timer requests through chipset module.
0 - Don't route watchdog timer requests through chipset module.

Related Parameters:

OPTION_SUPPORT_WATCHDOG - Enable watchdog timer support.
OPTION_SUPPORT_CHIPSET - Enable chipset support.
CHIPSET - Select Chipset Personality Module.
OPTION_WATCHDOG_CPU - Route requests through CPU module.
OPTION_WATCHDOG_BOARD - Route requests through board module.

7.1.112 OPTION_WATCHDOG_CPU Option

The **OPTION_WATCHDOG_CPU** option enables or disables code in the BIOS to route watchdog timer requests to the CPU Personality Module.

In order for this option to work, the **CPUCLASS** must be selected, and the CPU Personality Module must be programmed to be capable of handling watchdog timer requests.

Values:

1 - Route watchdog timer requests through CPU Personality Module.
0 - Don't route watchdog timer requests through CPU Personality Module.

Related Parameters:

OPTION_SUPPORT_WATCHDOG - Enable watchdog timer support.
CPUCLASS - Select CPU Personality Module.
OPTION_WATCHDOG_CPU - Route requests through CPU module.
OPTION_WATCHDOG_BOARD - Route requests through board module.

7.1.113 OPTION_WATCHDOG_BOARD Option

The **OPTION_WATCHDOG_BOARD** option enables or disables code in the BIOS to route watchdog timer requests to the Board Personality Module.

In order for this option to work, the **BOARD** must be selected, and the Board Personality Module must be programmed to be capable of handling watchdog timer requests.

Values:

1 - Route watchdog timer requests through Board Personality Module.
0 - Don't route watchdog timer requests through Board Personality Module.

Related Parameters:

OPTION_SUPPORT_WATCHDOG - Enable watchdog timer support.
BOARD - Select Board Personality Module.
OPTION_WATCHDOG_CPU - Route requests through CPU module.
OPTION_WATCHDOG_CHIPSET - Route requests through chipset module.

7.1.114 OPTION_WATCHDOG_TIMER_KICK Option

The **OPTION_WATCHDOG_TIMER_KICK** option enables or disables code in the BIOS to automatically "kick the dog" on each timer tick handled by the BIOS INT 8 ISR. This provides a way for the watchdog timer to function as an interrupt latency watchdog when the option is enabled, and to function as a traditional application-oriented watchdog timer when the option is disabled.

Values:

1 - Automatically kick the dog on every timer tick (appx. 55ms intervals).
0 - Don't automatically kick the dog on every timer tick.

Related Parameters:

OPTION_SUPPORT_WATCHDOG - Enable watchdog timer support.

7.1.115 OPTION_CACHE_CPU Option

The **OPTION_CACHE_CPU** option enables or disables code in the BIOS to route cache control requests to the CPU Personality Module. Only 80486 CPUs and above support L1 caches, except for certain 386 chips manufactured by IBM.

In order for this option to work, the **CPUCLASS** must be selected, and the CPU Personality Module must be programmed to be capable of handling cache control requests. CPU Personality Modules, including NOCPU, shipped by General Software, support cache control requests.

This option controls L1 caches, but can be used in conjunction with L2 cache controls.

The **OPTION_SUPPORT_CACHE** parameter does *not* affect the L1 cache logic and is not necessary for enabling this option.

Values:

1 - Route L1 cache control requests through CPU module.
0 - Don’t L1 route cache control requests through CPU module.

Related Parameters:

CPUCLASS - Select CPU Personality Module.
OPTION_SUPPORT_CACHE - Enable L2 cache support.

7.1.116 OPTION_CACHE_CHIPSET Option

The **OPTION_CACHE_CHIPSET** option enables or disables code in the BIOS to route L2 cache control requests to the Chipset Personality Module.

In order for this option to work, both **OPTION_SUPPORT_CACHE** and **OPTION_SUPPORT_CHIPSET** must be enabled, the **CHIPSET** type must be selected, and the Chipset Personality Module must be programmed to be capable of handling L2 cache control requests.

Values:

1 - Route L2 cache control requests through chipset.
0 - Don’t route L2 cache control requests through chipset.

Related Parameters:

OPTION_SUPPORT_CHIPSET - Enable chipset support.
CHIPSET - Select Chipset Personality Module.
OPTION_SUPPORT_CACHE - Enable L2 cache support.

7.1.117 OPTION_CACHE_BOARD Option

The **OPTION_CACHE_BOARD** option enables or disables code in the BIOS to route L2 cache control requests to the Board Personality Module.

In order for this option to work, **OPTION_SUPPORT_CACHE** must be enabled, the **BOARD** module must be selected, and the Board Personality Module must be programmed to be capable of handling L2 cache control requests.

Values:

1 - Route L2 cache control requests through board module.
0 - Don't route L2 cache control requests through board module.

Related Parameters:

BOARD - Select Board Personality Module.
OPTION_SUPPORT_CACHE - Enable L2 cache support.

7.1.118 OPTION_SPEED_CPU Option

The **OPTION_SPEED_CPU** option enables or disables code in the BIOS to route CPU speed control requests to the CPU Personality Module.

In order for this option to work, the **CPUCLASS** must be selected, and the CPU Personality Module must be programmed to be capable of handling CPU speed control requests.

Values:

1 - Route speed control requests through CPU.
0 - Don't route speed control requests through CPU.

Related Parameters:

CPUCLASS - Select CPU Personality Module.

7.1.119 OPTION_SPEED_CHIPSET Option

The **OPTION_SPEED_CHIPSET** option enables or disables code in the BIOS to route CPU speed control requests to the Chipset Personality Module.

In order for this option to work, **OPTION_SUPPORT_CHIPSET** must be enabled, the **CHIPSET** type must be selected, and the Chipset Personality Module must be programmed to be capable of handling CPU speed control requests.

Values:

1 - Route speed control requests through chipset.

0 - Don't route speed control requests through chipset.

Related Parameters:

OPTION_SUPPORT_CHIPSET - Enable chipset support.
CHIPSET - Select Chipset Personality Module.

7.1.120 OPTION_SPEED_BOARD Option

The **OPTION_SPEED_BOARD** option enables or disables code in the BIOS to route CPU speed control requests to the Board Personality Module.

In order for this option to work, the **BOARD** module must be selected, and the Board Personality Module must be programmed to be capable of handling CPU speed control requests.

Values:

1 - Route speed control requests through board module.
0 - Don't route speed control requests through board module.

Related Parameters:

BOARD - Select Board Personality Module.

7.1.121 OPTION_A20_8042 Option

The **OPTION_A20_8042** option enables or disables code in the BIOS to route A20 gating requests to the 8042 keyboard controller.

In order for this option to work, **OPTION_SUPPORT_8042** must be enabled, and the 8042 keyboard BIOS must be capable of processing A20 gate control requests requests for this option to be valid.

This is the traditional mechanism used in the original IBM PC/AT Personal Computer to gate the A20 line when running in protected mode. For information about how A20 is used in the system, consult the section on **OPTION_SUPPORT_PROTECT_MODE**.

If more than one A20 gate mechanism exists in the target, then several options may need to be enabled, depending on whether they are wire-OR'd or wire-AND'd together.

Values:

1 - Route A20 gate requests through 8042 keyboard controller.
0 - Don't route A20 gate requests through 8042 keyboard controller.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.
OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.

OPTION_A20_PORT92 - Enable port 92h A20 gating support.
OPTION_A20_CPU - Enable A20 gating through CPU module.
OPTION_A20_CHIPSET - Enable A20 gating through chipset module.
OPTION_A20_BOARD - Enable A20 gating through board module.

7.1.122 OPTION_A20_CHIPSET Option

The **OPTION_A20_CHIPSET** option enables or disables code in the BIOS to route A20 gating requests to the Chipset Personality Module.

In order for this option to work, **OPTION_SUPPORT_CHIPSET** must be enabled, the **CHIPSET** type must be selected, and the Chipset Personality Module must be programmed to be capable of handling A20 gate control requests.

This is the modern mechanism used to gate the A20 line in high-density motherboard designs when running in protected mode. For information about how A20 is used in the system, consult the section on **OPTION_SUPPORT_PROTECT_MODE**.

If more than one A20 gate mechanism exists in the target, then several options may need to be enabled, depending on whether they are wire-OR'd or wire-AND'd together.

Do not choose this method if the chipset's fast A20 gate is just an implementation of port 92h. If this is the case, use **OPTION_A20_PORT92** instead.

Values:

1 - Route A20 gate requests through chipset.
0 - Don't route A20 gate requests through chipset.

Related Parameters:

OPTION_SUPPORT_CHIPSET - Enable chipset support.
CHIPSET - Select Chipset Personality Module.
OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.
OPTION_A20_8042 - Enable keyboard controller A20 gating support.
OPTION_A20_PORT92 - Enable port 92h A20 gating support.
OPTION_A20_CPU - Enable A20 gating through CPU module.
OPTION_A20_BOARD - Enable A20 gating through board module.

7.1.123 OPTION_A20_CPU Option

The **OPTION_A20_CPU** option enables or disables code in the BIOS to route A20 gating requests to the CPU Personality Module.

In order for this option to work, the **CPUCLASS** parameter must be set to the proper CPU Personality Module identifier, and the CPU Personality Module must be programmed to be capable of handling A20 gate control requests.

This is a very new mechanism used to gate the A20 line in very high-integration CPUs when running in protected mode. For information about how A20 is used in the system, consult the section on **OPTION_SUPPORT_PROTECT_MODE**.

If more than one A20 gate mechanism exists in the target, then several options may need to be enabled, depending on whether they are wire-OR'd or wire-AND'd together.

Do not choose this method if the CPU's fast A20 gate is just an implementation of port 92h. If this is the case, use **OPTION_A20_PORT92** instead. For example, the Intel 80C386-EX CPU contains a port 92h A20 gate; the port is implemented in the CPU but emulates a standard port 92h.

Values:

1 - Route A20 gate requests through CPU.
0 - Don't route A20 gate requests through CPU.

Related Parameters:

CPUCLASS - Select CPU Personality Module.
OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.
OPTION_A20_8042 - Enable keyboard controller A20 gating support.
OPTION_A20_PORT92 - Enable port 92h A20 gating support.
OPTION_A20_CHIPSET - Enable A20 gating through chipset module.
OPTION_A20_BOARD - Enable A20 gating through board module.

7.1.124 OPTION_A20_BOARD Option

The **OPTION_A20_BOARD** option enables or disables code in the BIOS to route A20 gating requests to the Board Personality Module.

In order for this option to work, the **BOARD** parameter must be set to the proper Board Personality Module identifier, and the Board Personality Module must be programmed to be capable of handling A20 gate control requests.

Routing A20 requests through the Board Personality Module makes it possible for OEMs to customize handling of A20 gating requests for certain designs. For information about how A20 is used in the system, consult the section on **OPTION_SUPPORT_PROTECT_MODE**.

If more than one A20 gate mechanism exists in the target, then several options may need to be enabled, depending on whether they are wire-OR'd or wire-AND'd together.

Do not choose this method if the board's fast A20 gate is just an implementation of port 92h. If this is the case, use **OPTION_A20_PORT92** instead.

Values:

1 - Route A20 gate requests through CPU.
0 - Don't route A20 gate requests through CPU.

Related Parameters:

CPUCLASS - Select CPU Personality Module.
OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.
OPTION_A20_8042 - Enable keyboard controller A20 gating support.
OPTION_A20_PORT92 - Enable port 92h A20 gating support.
OPTION_A20_CHIPSET - Enable A20 gating through chipset module.
OPTION_A20_CPU - Enable A20 gating through CPU module.

7.1.125 OPTION_A20_PORT92 Option

The **OPTION_A20_PORT92** option enables or disables code in the BIOS to route A20 gating requests to code that manipulates the PS/2-compatible I/O port 92h. The hardware must support port 92h in order for this option to be valid.

Most new chipsets on the market support port 92h; they may also support a fast gate A20 option that is separate from port 92h. If port 92h is provided, use it first, and then experiment with the other method later to see if performance improvements are possible.

This is the modern mechanism used to gate the A20 line in high-integration CPUs or PS/2-compatible motherboard designs when running in protected mode. For information about how A20 is used in the system, consult the section on **OPTION_SUPPORT_PROTECT_MODE**.

If more than one A20 gate mechanism exists in the target, then several options may need to be enabled, depending on whether they are wire-OR'd or wire-AND'd together.

Values:

1 - Route A20 gate requests through port 92h.
0 - Don't route A20 gate requests through port 92h.

Related Parameters:

OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.
OPTION_A20_8042 - Enable keyboard controller A20 gating support.
OPTION_A20_CPU - Enable A20 gating through CPU module.
OPTION_A20_CHIPSET - Enable A20 gating through chipset module.
OPTION_A20_BOARD - Enable A20 gating through board module.

7.1.126 OPTION_A20_FAILMEM Option

The **OPTION_A20_FAILMEM** option enables or disables code in the BIOS to cause a critical error to occur during POST if the A20 gate test fails.

If no extended memory is available, then the A20 test can fail; therefore, on targets with no extended memory, this option should be disabled.

On targets with extended memory, the A20 test should be enabled.

For information about how A20 is used in the system, consult the section on **OPTION_SUPPORT_PROTECT_MODE**.

Values:

1 - Cause A20 test failures to generate critical error during POST.
0 - Don't cause A20 test failures to generate critical error during POST.

Related Parameters:

OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.

7.1.127 OPTION_REBOOT_JUMP Option

The **OPTION_REBOOT_JUMP** option enables or disables code in the BIOS to route reboot requests to code that simply jumps to location F000:FFF0 in real mode.

Do not use this with processors that support protected mode, or any special bootstrap code at the top of the extended memory address space may not be executed (for example, a CyberQuest Flash loader).

Values:

1 - Route reboot requests through jump to F000:FFF0.
0 - Don't route reboot requests through jump to F000:FFF0.

Related Parameters:

None.

7.1.128 OPTION_REBOOT_PORT92 Option

The **OPTION_REBOOT_PORT92** option enables or disables code in the BIOS to route reboot requests to code that raises a bit in the PS/2-compatible I/O port 92h.

The hardware must support port 92h in order for this option to be valid. Most newer chipsets on the market today support this I/O port; check your technical documentation for details.

Rebooting may not function properly unless the A20 line is being properly gated as well; see **OPTION_SUPPORT_PROTECT_MODE** for details.

Values:

1 - Route reboot requests through I/O port 92h.
0 - Don't route reboot requests through I/O port 92h.

Related Parameters:

None.

7.1.129 OPTION_REBOOT_8042 Option

The **OPTION_REBOOT_8042** option enables or disables code in the BIOS to route reboot requests to the 8042 keyboard controller.

In order for this option to work, you must enable **OPTION_SUPPORT_8042**, and then the 8042 keyboard BIOS must be capable of responding to a reboot request.

The 8042 method used to reboot the target works in more circumstances than port 92h, but is much slower. If you have a choice between the 8042 and port 92h, use port 92h instead.

Rebooting may not function properly unless the A20 line is being properly gated as well; see **OPTION_SUPPORT_PROTECT_MODE** for details.

Values:

1 - Route reboot requests through the 8042.
0 - Don't route reboot requests through the 8042.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.

7.1.130 OPTION_REBOOT_CHIPSET Option

The **OPTION_REBOOT_CHIPSET** option enables or disables code in the BIOS to route reboot requests to the Chipset Personality Module.

In order for this option to work, **OPTION_SUPPORT_CHIPSET** must be enabled, the **CHIPSET** type must be selected, and the Chipset Personality Module must be programmed to be capable of handling reboot control requests.

Do not choose this method if the chipset's reboot mechanism is just an implementation of port 92h. If this is the case, use **OPTION_REBOOT_PORT92** instead.

Values:

1 - Route reboot requests through chipset.
0 - Don't route reboot requests through chipset.

Related Parameters:

OPTION_SUPPORT_CHIPSET - Enable chipset support.
CHIPSET - Select CPU Personality Module.

7.1.131 OPTION_REBOOT_BOARD Option

The **OPTION_REBOOT_BOARD** option enables or disables code in the BIOS to route reboot requests to the Board Personality Module.

In order for this option to work, the **BOARD** name must be selected, and the Board Personality Module must be programmed to be capable of handling reboot control requests.

Do not choose this method if the board's reboot mechanism is just an implementation of port 92h. If this is the case, use **OPTION_REBOOT_PORT92** instead.

Values:

1 - Route reboot requests through board module.
0 - Don't route reboot requests through board module.

Related Parameters:

BOARD - Select Board Personality Module.

7.1.132 OPTION_TOREAL_PORT92 Option

The **OPTION_TOREAL_PORT92** option enables or disables code in the BIOS to route mode switch requests to the PS/2-compatible I/O port 92h.

This I/O port must be supported by the hardware in order for this option to be valid.

Most newer chipsets and high-integration CPUs support port 92h. Note that port 92h by itself does not do mode switching; instead, it allows the BIOS to reset the processor, just as the IBM PC/AT Personal Computer did with the 8042 keyboard controller. The port 92h method is faster than the 8042 method. The fastest method is the CPU mode switch (available only on 80386 and above CPUs).

Consult the section on **OPTION_SUPPORT_PROTECT_MODE** for a detailed discussion of related issues.

Values:

1 - Route mode switch requests through I/O port 92h.
0 - Don't route mode switch requests through I/O port 92h.

Related Parameters:

OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.

7.1.133 OPTION_TOREAL_8042 Option

The **OPTION_TOREAL_8042** option enables or disables code in the BIOS to route mode switch requests to the 8042 keyboard controller.

OPTION_SUPPORT_8042 must be enabled, and the reset CPU function must be supported by the 8042 in order for this option to be valid.

Traditionally, the IBM PC/AT Personal Computer switched its 80286 processor into real from protected mode by instructing the 8042 keyboard controller to toggle the CPU's reset line. This is no longer necessary with the 80386 and above processors, which can switch into and out of protected mode with simple CPU instructions. Avoid the 8042 option if alternate methods exist.

Consult the section on **OPTION_SUPPORT_PROTECT_MODE** for a detailed discussion of related issues.

Values:

1 - Route mode switch requests through the 8042.
0 - Don't route mode switch requests through the 8042.

Related Parameters:

OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.
OPTION_SUPPORT_8042 - Enable 8042 support.

7.1.134 OPTION_TOREAL_CPU Option

The **OPTION_TOREAL_CPU** option enables or disables code in the BIOS to route mode switch requests to code that uses 80386 or better instructions to switch modes directly.

This is the fastest way to switch modes, but this technique only runs on 80386 and above processors. Use this technique above the others whenever possible.

Consult the section on **OPTION_SUPPORT_PROTECT_MODE** for a detailed discussion of related issues.

Values:

1 - Route mode switch requests through the mode switch instructions on 80386s and above.
0 - Don't route mode switch requests through the mode switch instructions on 80386s and above.

Related Parameters:

OPTION_SUPPORT_PROTECT_MODE - Enable protected mode support.

7.1.135 OPTION_POWERMAN_CPU Option

The **OPTION_POWERMAN_CPU** option enables or disables code in the BIOS to route power management requests to the CPU Personality Module.

The **OPTION_SUPPORT_POWERMAN** option must be enabled to support APM, and the CPU Personality Module must support power management functionality in order for this option to be valid.

Values:

1 - Route power management requests to CPU.
0 - Disable power management requests to CPU.

Related Parameters:

OPTION_SUPPORT_POWERMAN - Enable Advanced Power Management services.

CPUCLASS - Select the CPU Personality Module.

7.1.136 OPTION_POWERMAN_CHIPSET Option

The **OPTION_POWERMAN_CHIPSET** option enables or disables code in the BIOS to route power management requests to the Chipset Personality Module.

To use this option, you must enable **OPTION_SUPPORT_CHIPSET** and set the **CHIPSET** parameter to select the correct Chipset Personality Module to be used. The Chipset Personality Module must support power management functionality in order for this option to be valid.

Values:

1 - Route power management requests to the chipset.
0 - Disable power management requests to the chipset.

Related Parameters:

OPTION_SUPPORT_POWERMAN - Enable advanced power management services.

OPTION_SUPPORT_CHIPSET - Enable chipset support.

CHIPSET - Select Chipset Personality Module.

7.1.137 OPTION_POWERMAN_BOARD Option

The **OPTION_POWERMAN_BOARD** option enables or disables code in the BIOS to route power management requests to the Board Personality Module.

To use this option, you must set the **BOARD** parameter to select the correct Board Personality Module to be used. The Board Personality Module must support power management functionality in order for this option to be valid.

Values:

1 - Route power management requests to the board module.

0 - Disable power management requests to the board module.

Related Parameters:

OPTION_SUPPORT_POWERMAN - Enable advanced power management services.

BOARD - Select Board Personality Module.

7.1.138 OPTION_SERIAL_8250 Option

The **OPTION_SERIAL_8250** option enables or disables code in the BIOS to provide serial I/O services over external 8250-compatible UARTs. This mechanism is compatible with PC, PC/XT, PC/AT and most compatible designs.

This option requires **OPTION_SUPPORT_8250** to be enabled. For more information about supporting standard PC UARTs and configuring related options, see the section on **OPTION_SUPPORT_8250**.

The 8250 code can support higher-end UARTs, such as 16450 and 16550 parts. The FIFOs can be enabled on the 16550 by setting **OPTION_SERIAL_FIFO**.

If **OPTION_SERIAL_WAIT_DSR** is enabled, then the INT 14h code will wait for DSR to be raised before receiving data. If **OPTION_SERIAL_WAIT_DSRCTS** is enabled, then the INT 14h code will wait for DSR and also CTS before sending data.

This option can be used in conjunction with the **OPTION_SERIAL_CPU** configuration option; external and on-board serial ports can be used in a system, and are automatically assigned different COM port numbers by the BIOS.

Values:

1 - Enable serial I/O over 8250/16450/16550 UARTs.
0 - Disable serial I/O over 8250/16450/16550 UARTs.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O services.
OPTION_SUPPORT_8250 - Enable 8250 support.
OPTION_SERIAL_WAIT_DSR - Enable DSR support.
OPTION_SERIAL_WAIT_DSRCTS - Enable DSR & CTS support.
OPTION_SERIAL_FIFO - Enable FIFO support.

7.1.139 OPTION_SERIAL_CPU Option

The **OPTION_SERIAL_CPU** option enables or disables code in the BIOS to route serial I/O requests through the CPU Personality Module.

The CPU Personality Module must be selected with the **CPUCLASS** parameter, and it must be capable of performing serial I/O for this option to be valid.

Many high-integration CPUs provide 8250-compatible UARTs. When this is the case, the CPU Personality Module needs **OPTION_SUPPORT_8250** to be enabled for the UART support, and then this option must be disabled. If in doubt, consult your CPU Personality Module documentation or review the source code to determine if the CPU has non-standard UARTs that cannot be supported by the 8250 code.

This option can be used in conjunction with the **OPTION_SERIAL_8250** configuration option; external and on-board serial ports can be used in a system, and are automatically assigned different COM port numbers by the BIOS.

Values:

1 - Enable serial I/O through CPU serial ports.
0 - Disable serial I/O through CPU serial ports.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O services.
CPUCLASS - Select the CPU Personality Module.

7.1.140 OPTION_SERIAL_WAIT_DSR Option

The **OPTION_SERIAL_WAIT_DSR** option enables or disables code in the BIOS to wait on receive requests for DSR to become active before actually attempting to receive a character.

When transmitting, this option has no effect. Instead, **OPTION_SERIAL_WAIT_DSRCTS** provides a way to wait for both DSR and CTS before proceeding to transmit.

This option does not affect UARTs supported by CPU Personality Modules. These modules are free to implement serial I/O in whatever manner is appropriate.

Values:

1 - Enable wait for Data Set Ready.
0 - Disable wait for Data Set Ready.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O services.

OPTION_SUPPORT_8250 - Enable 8250 support.

OPTION_SERIAL_8250 - Support COM ports over 8250 devices.

OPTION_SERIAL_WAIT_DSRCTS - Enable wait for DSR and CTS on transmits.

7.1.141 OPTION_SERIAL_WAIT_DSRCTS Option

The **OPTION_SERIAL_WAIT_DSRCTS** option enables or disables code in the BIOS to wait on transmit requests for DSR and CTS to become active before actually attempting to send a character.

When receiving, this option has no effect. Instead, **OPTION_SERIAL_WAIT_DSR** provides a way to wait for DSR before proceeding to receiving.

This option does not affect UARTs supported by CPU Personality Modules. These modules are free to implement serial I/O in whatever manner is appropriate.

Values:

1 - Enable wait for Data Set Ready and Clear To Send.
0 - Disable wait for Data Set Ready and Clear To Send.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O services.

OPTION_SUPPORT_8250 - Enable 8250 support.

OPTION_SERIAL_8250 - Support COM ports over 8250 devices.

OPTION_SERIAL_WAIT_DSR - Enable wait for DSR on receives.

7.1.142 OPTION_SERIAL_FIFO Option

The **OPTION_SERIAL_FIFO** option enables or disables code in the BIOS to enable the FIFO on UARTs that support operational FIFOs, such as the 16550.

Some UARTs cannot support FIFOs, such as 16450's. There is a bug in these parts that causes the receive FIFO to not notify the host that a character is available, even though it is in the FIFO. This results in loss of data, and some times, a seemingly dead COM port.

This option does not affect UARTs supported by CPU Personality Modules. These modules are free to implement serial I/O in whatever manner is appropriate.

Please note that not all applications are prepared to support FIFOs, and may in fact not operate correctly because they do not receive an interrupt for every character that is received, or an interrupt for every character that is transmitted. If you are having trouble getting communications software to work with this option enabled, the problem may actually reside in the application.

Values:

1 - Enable FIFO support for 16550 UARTs.
0 - Disable FIFO support for 16550 UARTs.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O services.

OPTION_SUPPORT_8250 - Enable 8250 support.

OPTION_SERIAL_8250 - Support COM ports over 8250 devices.

7.1.143 OPTION_SERIAL_HALT Option

The **OPTION_SERIAL_HALT** option enables or disables code in the BIOS to execute a HLT instruction whenever an INT 14h read with wait request is issued and there is no character waiting in the UART's receiver buffer.

This feature can be enabled to lower power consumption significantly on systems that use redirected console I/O and don't have a keyboard controller. Please note that this technique may or may not be compatible with the power management model in your system; consult the processor or chipset technical reference manual for details about how the CPU interprets a HLT instruction.

This option is only valid for the 8250 core BIOS driver.

Values:

1 - Enable HLT in spin-wait for input.
0 - Disable HLT in spin-wait for input.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O services.

OPTION_SUPPORT_8250 - Enable 8250 support.

OPTION_SERIAL_8250 - Support COM ports over 8250 devices.

7.1.144 OPTION_SERIAL_9600_BAUD Option

The **OPTION_SERIAL_9600_BAUD** option enables or disables a special modifier to the serial I/O INT 14h service handler that filters "set mode" functions. When the baud rate is set for anything other than 9600 baud, the BIOS automatically changes the request to 9600 baud.

This feature is used to support operating systems such as MS-DOS, that immediately reprogram all COM ports to other baud rates (in the case of MS-DOS, 2400,e,7,1). The problem with this is that it reprograms the serial port used by the remote disk and by the remote console I/O, both of which operate faster than 2400 baud.

When baud rates are changed using direct hardware access to UARTs by applications, this option does not prevent the changes. For examples, `INTERSVR.EXE` and `SERDRIVE.SYS` are examples of software that reprogram a UART to the maximum possible baud rate.

Values:

1 - Disallow changing of baud rates by MS-DOS.
0 - Allow changing of baud rates by MS-DOS.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable INT 14h interface.

7.1.145 OPTION_PARALLEL_EXTERNAL Option

The **OPTION_PARALLEL_EXTERNAL** option enables or disables code in the BIOS to support parallel I/O requests with PC-compatible parallel port hardware.

OPTION_SUPPORT_PARALLEL must be enabled in order for INT 17h parallel I/O requests to be processed.

This option can be used in conjunction with the **OPTION_PARALLEL_CPU** configuration option; external and on-board parallel ports can be used in a system, and are automatically assigned different LPT port numbers by the BIOS.

Values:

1 - Enable parallel I/O through PC-compatible parallel ports.
0 - Disable external parallel ports.

Related Parameters:

OPTION_SUPPORT_PARALLEL - Enable INT 17h parallel I/O services.

7.1.146 OPTION_PARALLEL_CPU Option

The **OPTION_PARALLEL_EXTERNAL** option enables or disables code in the BIOS to route parallel port I/O requests to the CPU Personality Module.

You must set the **CPUCLASS** parameter to choose the correct CPU Personality Module to be used, and the CPU Personality Module must support parallel I/O in order for this option to be valid.

This option can be used in conjunction with the **OPTION_PARALLEL_EXTERNAL** configuration option; external and on-board parallel ports can be used in a system, and are automatically assigned different LPT port numbers by the BIOS.

Values:

1 - Enable parallel I/O through CPU.
0 - Disable parallel I/O through CPU.

Related Parameters:

OPTION_SUPPORT_PARALLEL - Enable parallel I/O services.

CPUCLASS - Select CPU Personality Module.

7.1.147 OPTION_KEYBOARD_PCAT Option

The **OPTION_KEYBOARD_PCAT** option enables or disables code in the BIOS to drive a PC, PC/XT, or PC/AT-style keyboard. All of these keyboard types are supported with this one option.

If you have a very custom keyboard that cannot just plug into a PC, PC/XT, or PC/AT computer, then you may enable **OPTION_KEYBOARD_CUSTOMER** instead, and edit SYSTEM\CUSTKBD.ASM to provide an equivalent driver for your own hardware.

OPTION_SUPPORT_8042 or **OPTION_SUPPORT_8255** must be selected to provide basic controller support, and **OPTION_SUPPORT_KEYBOARD** must be enabled to provide basic keyboard controller support.

Values:

1 - Enable PC, XT, and AT keyboard support.
0 - Disable PC, XT, and AT keyboard support.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 AT keyboard controller support.

OPTION_SUPPORT_8255 - Enable 8255 XT keyboard controller support.

OPTION_SUPPORT_KEYBOARD - Enable basic keyboard controller support.

OPTION_KEYBOARD_MATRIX - Enable special key translation on matrix keyboards.

7.1.148 OPTION_KEYBOARD_CUSTOMER Option

The **OPTION_KEYBOARD_CUSTOMER** option enables or disables code in the BIOS to drive an OEM-defined keyboard controller.

If you have a very custom keyboard that cannot just plug into a PC, PC/XT, or PC/AT computer, then you may enable **OPTION_KEYBOARD_CUSTOMER**, and edit `SYSTEM\CUSTKBD.ASM` to provide an equivalent driver for your own hardware.

The OEM-defined keyboard driver does not require any other options unless the OEM specifically makes references to them in the code.

Values:

1 - Enable OEM-defined custom keyboard support.
0 - Disable OEM-defined custom keyboard support.

Related Parameters:

OPTION_SUPPORT_KEYBOARD - Enable basic keyboard controller support.

7.1.149 OPTION_KEYBOARD_MATRIX Option

The **OPTION_KEYBOARD_MATRIX** option enables or disables code in the PCAT keyboard driver to handle the special keys, PrtScrn, SysReq, Pause, and Break in a special manner compatible with most matrix keyboards.

This option is not a driver selection. It should be used in conjunction with the **OPTION_KEYBOARD_PCAT** option. Usually, matrix keyboards are driven with the **OPTION_SUPPORT_8255** low-level keyboard interface, rather than **OPTION_SUPPORT_8042**.

Values:

1 - Enable special key translation on matrix keyboards.
0 - Disable special key translation on matrix keyboards.

Related Parameters:

OPTION_SUPPORT_KEYBOARD - Enable basic keyboard controller support.
OPTION_KEYBOARD_PCAT - Enable PC/AT keyboard driver.

7.1.150 OPTION_KEYBOARD_PCXT Option

The **OPTION_KEYBOARD_PCXT** option enables or disables code in the BIOS that supports an XT keyboard interface (with raw scan codes through port 60h, not via an 8042-style AT keyboard controller).

Values:

1 - Enable XT keyboard support.
0 - Disable XT keyboard support.

Related Parameters:

OPTION_SUPPORT_KEYBOARD - Enable keyboard support in general.
OPTION_KEYBOARD_PCAT - Enable AT keyboard support instead.
OPTION_KEYBOARD_MATRIX - Enable matrix keyboard support instead.
OPTION_KEYBOARD_CHIPSET - Enable chipset module keyboard support.

7.1.151 OPTION_KEYBOARD_CHIPSET Option

The **OPTION_KEYBOARD_CHIPSET** option enables or disables code in the BIOS that supports a keyboard handled by a custom driver in the chipset module. Processors like the AMD

SC400 have special keyboard support that cannot be supported directly by the standard core drivers.

Values:

1 - Enable keyboard support in chipset module.
0 - Disable keyboard support in chipset module.

Related Parameters:

OPTION_SUPPORT_KEYBOARD - Enable keyboard support in general.
OPTION_KEYBOARD_PCXT - Enable XT keyboard support instead.
OPTION_KEYBOARD_PCAT - Enable AT keyboard support instead.
OPTION_KEYBOARD_MATRIX - Enable matrix keyboard support instead.

7.1.152 OPTION_8042_TESTP22P23 Option

The **OPTION_8042_TESTP22P23** option enables or disables code in the BIOS to test the port 2, pins 2 and 3 on the 8042 during POST. This is an esoteric function that should only be enabled if the corresponding firmware in the 8042 is supported.

OPTION_SUPPORT_8042 must be selected to provide basic 8042 support.

Values:

1 - Enable test of P22, P23 on 8042 during POST.
0 - Disable test of P22, P23 on 8042 during POST.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.

7.1.153 OPTION_8042_READPWRSTAT Option

The **OPTION_8042_READPWRSTAT** option enables or disables code in the BIOS to read the power-on status of the 8042 during POST.

Not all 8042 keyboard controllers support this function. Disable this option to start with, and after you have a working BIOS, you may want to enable it to provide additional diagnostics during POST.

OPTION_SUPPORT_8042 must be selected to provide basic 8042 support.

Values:

1 - Enable read of power-on status of 8042 during POST.
0 - Disable read of power-on status of 8042 during POST.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.

7.1.154 OPTION_8042_CHECKBAT Option

The **OPTION_8042_CHECKBAT** option enables or disables code in the BIOS to check the results of the controller's Basic Assurance Test (BAT) during POST to see if it is correct.

Some 8042 keyboard controllers may not be able to respond with a valid BAT at this early point in POST, and instead will respond with a "resend" command. Disable this option to start with, and after you have a working BIOS, you may want to enable it to provide additional diagnostics during POST.

OPTION_SUPPORT_8042 must be selected to provide basic 8042 support.

Values:

1 - Enable verification of BAT from 8042 during POST.
0 - Disable verification of BAT from 8042 during POST.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.

7.1.155 OPTION_8042_PS2 Option

The **OPTION_8042_PS2** option enables or disables code in the BIOS to insert delays between checking status bits and reading or writing data to the controller.

This is a necessary procedure for PS/2-compatible 8042 keyboard controllers. We suggest you enable this option to start with, and after you have a working BIOS, you may want to disable it to slightly (probably imperceptably) improve performance.

OPTION_SUPPORT_8042 must be selected to provide basic 8042 support.

Values:

1 - Enable PS/2 delays during 8042 read/write functions.
0 - Disable PS/2 delays during 8042 read/write functions.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.

7.1.156 OPTION_8042_WAIT_BEFORE_BAT Option

The **OPTION_8042_WAIT_BEFORE_BAT** option enables or disables code in the BIOS to issue a lengthy delay before reading the BAT code from the 8042 keyboard controller.

This is a necessary procedure for some controllers because they take a long time to perform their internal diagnostics. We suggest you disable this option to start with, and only if you are unable to get the keyboard working, you may wish to enable it to add a delay.

OPTION_SUPPORT_8042 must be selected to provide basic 8042 support.

Values:

1 - Enable delay before reading BAT during POST.
0 - Disable delay before reading BAT during POST.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.
OPTION_8042_CHECKBAT - Check BAT result.

7.1.157 OPTION_VIDEO_6845 Option

The **OPTION_VIDEO_6845** option enables or disables code in the BIOS to drive a PC-compatible, 6845 CRT controller.

This controller type is compatible with desktop monochrome cards, color cards, Hercules cards, and VGA/SVGA cards that emulate 6845's.

OPTION_SUPPORT_VIDEO must be enabled in order for INT 10h requests to be accepted from the application and routed to the 6845 driver.

If you have redirected console I/O with **CONFIG_CON_REDIR_STD**, **CONFIG_CON_REDIR_DEBUG**, or **CONFIG_CON_REDIR_SETUP**, then when redirection occurs, the 6845 driver does not receive control, unless **OPTION_VIDEO_DUPLICATE** is enabled. The latter option causes all output that is redirected to also appear on the standard video display.

If you are using a standard video card that requires this option, you should also enable **OPTION_VIDEO_VIDEOMEM**, since these cards all contain video RAM that can be tested and automatically detected during POST.

The 6845 driver requires that you specify the segment addresses of video RAM for different modes. **CONFIG_VIDEO_SEG_GRAPHIC** controls the graphics mode screen address, **CONFIG_VIDEO_SEG_MONO** controls the monochrome mode screen address, and **CONFIG_VIDEO_SEG_COLOR** controls the color screen address.

Values:

1 - Enable 6845 CRT controller support.
0 - Disable 6845 CRT controller support.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video controller support.

OPTION_SUPPORT_VIDEO_BOARDS - Enable ROM scan for additional video BIOS extensions to support EGA, VGA, and SVGA.

OPTION_VIDEO_VIDEOMEM - Enable autodetection of video RAM during POST.

CONFIG_VIDEO_SEG_GRAPHIC - Selects graphic mode video RAM segment.

CONFIG_VIDEO_SEG_MONO - Selects monochrome mode video RAM segment.

CONFIG_VIDEO_SEG_COLOR - Selects color video RAM segment.

OPTION_SUPPORT_CON_REDIRECTOR - Enable console I/O redirection.

CONFIG_CON_REDIR_STD - Standard console I/O redirection assignment.

CONFIG_CON_REDIR_DEBUG - Debugger console I/O redirection assignment.

CONFIG_CON_REDIR_SETUP - Setup screen system console I/O redirection assignment.

7.1.158 OPTION_VIDEO_HD61830 Option

The **OPTION_VIDEO_HD61830** option enables or disables code in the BIOS to drive an Hitachi HD-61830 LCD controller.

This controller type offers the same basic functionality as the 6845, but its implementation is totally different.

OPTION_SUPPORT_VIDEO must be enabled in order for INT 10h requests to be accepted from the user and routed to the driver.

If you have redirected console I/O with **CONFIG_CON_REDIR_STD**, **CONFIG_CON_REDIR_DEBUG**, or **CONFIG_CON_REDIR_SETUP**, then when redirection occurs, the driver does not receive control, unless **OPTION_VIDEO_DUPLICATE** is enabled. The latter option causes all output that is redirected to also appear on the standard video display.

There are no video cards that are compatible with the HD61830. Therefore, do not enable **OPTION_SUPPORT_VIDEO_BOARDS**.

Also, depending on the implementation of the hardware, the controller's RAM may not be available to the CPU. Therefore, do not enable **OPTION_VIDEO_VIDEOMEM**.

The HD61830 driver requires that you specify the segment addresses of video RAM for different modes. **CONFIG_VIDEO_SEG_GRAPHIC** controls the graphics mode screen address, **CONFIG_VIDEO_SEG_MONO** controls the monochrome mode screen address, and **CONFIG_VIDEO_SEG_COLOR** controls the color screen address.

This driver was donated by a German customer. The style of the code and its comments is not the same as the other code in the BIOS. This code is provided if it can be of help to you, but no support is available.

Values:

1 - Enable HD61830 LCD controller support.
0 - Disable HD61830 LCD controller support.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video controller support.

OPTION_VIDEO_VIDEOMEM - Enable autodetection of video RAM during POST.

CONFIG_VIDEO_SEG_GRAPHIC - Selects graphic mode video RAM segment.

CONFIG_VIDEO_SEG_MONO - Selects monochrome mode video RAM segment.

CONFIG_VIDEO_SEG_COLOR - Selects color video RAM segment.

OPTION_SUPPORT_CON_REDIRECTOR - Enable console I/O redirection.

CONFIG_CON_REDIR_STD - Standard console I/O redirection assignment.

CONFIG_CON_REDIR_DEBUG - Debugger console I/O redirection assignment.

CONFIG_CON_REDIR_SETUP - Setup screen system console I/O redirection assignment.

7.1.159 OPTION_VIDEO_HDMLCD Option

The **OPTION_VIDEO_HDMLCD** option enables or disables code in the BIOS to drive another family of LCD controllers.

This controller type offers the same basic functionality as the 6845, but its implementation is totally different.

OPTION_SUPPORT_VIDEO must be enabled in order for INT 10h requests to be accepted from the user and routed to the driver.

If you have redirected console I/O with **CONFIG_CON_REDIR_STD**, **CONFIG_CON_REDIR_DEBUG**, or **CONFIG_CON_REDIR_SETUP**, then when redirection occurs, the driver does not receive control, unless **OPTION_VIDEO_DUPLICATE** is enabled. The latter option causes all output that is redirected to also appear on the standard video display.

There are no video cards that are compatible with the HDMLCD module. Therefore, do not enable **OPTION_SUPPORT_VIDEO_BOARDS**.

Also, depending on the implementation of the hardware, the controller's RAM may not be available to the CPU. Therefore, do not enable **OPTION_VIDEO_VIDEOMEM**.

The HDMLCD driver requires that you specify the segment addresses of video RAM for different modes. **CONFIG_VIDEO_SEG_GRAPHIC** controls the graphics mode screen address, **CONFIG_VIDEO_SEG_MONO** controls the monochrome mode screen address, and **CONFIG_VIDEO_SEG_COLOR** controls the color screen address.

This driver was donated by a customer. The style of the code and its comments is not the same as the other code in the BIOS. This code is provided if it can be of help to you, but no support is available.

Values:

1 - Enable HDMLCD LCD controller support.
0 - Disable HDMLCD LCD controller support.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video controller support.

OPTION_VIDEO_VIDEOMEM - Enable autodetection of video RAM during POST.

CONFIG_VIDEO_SEG_GRAPHIC - Selects graphic mode video RAM segment.

CONFIG_VIDEO_SEG_MONO - Selects monochrome mode video RAM segment.

CONFIG_VIDEO_SEG_COLOR - Selects color video RAM segment.

OPTION_SUPPORT_CON_REDIRECTOR - Enable console I/O redirection.

CONFIG_CON_REDIR_STD - Standard console I/O redirection assignment.

CONFIG_CON_REDIR_DEBUG - Debugger console I/O redirection assignment.

CONFIG_CON_REDIR_SETUP - Setup screen system console I/O redirection assignment.

7.1.160 OPTION_VIDEO_AMDELAN Option

The **OPTION_VIDEO_AMDELAN** option enables or disables code in the BIOS to drive the AMD SC300 and SC400 family of integrated LCD controllers.

This controller type offers the same basic functionality as the 6845, but its implementation is different, and is handled in the Chipset Personality Module associated with the Elan CPU being used.

OPTION_SUPPORT_VIDEO must be enabled in order for INT 10h requests to be accepted from the user and routed to the driver.

If you have redirected console I/O with **CONFIG_CON_REDIR_STD**, **CONFIG_CON_REDIR_DEBUG**, or **CONFIG_CON_REDIR_SETUP**, then when redirection occurs, the driver does not receive control, unless **OPTION_VIDEO_DUPLICATE** is enabled. The latter option causes all output that is redirected to also appear on the standard video display.

The AMD Elan driver requires that you specify the segment addresses of video RAM for different modes. **CONFIG_VIDEO_SEG_GRAPHIC** controls the graphics mode screen address, **CONFIG_VIDEO_SEG_MONO** controls the monochrome mode screen address, and **CONFIG_VIDEO_SEG_COLOR** controls the color screen address.

Values:

1 - Enable AMD Elan LCD controller support.
0 - Disable AMD Elan LCD controller support.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video controller support.

OPTION_VIDEO_VIDEOMEM - Enable autodetection of video RAM during POST.

CONFIG_VIDEO_SEG_GRAPHIC - Selects graphic mode video RAM segment.

CONFIG_VIDEO_SEG_MONO - Selects monochrome mode video RAM segment.

CONFIG_VIDEO_SEG_COLOR - Selects color video RAM segment.

OPTION_SUPPORT_CON_REDIRECTOR - Enable console I/O redirection.

CONFIG_CON_REDIR_STD - Standard console I/O redirection assignment.

CONFIG_CON_REDIR_DEBUG - Debugger console I/O redirection assignment.

CONFIG_CON_REDIR_SETUP - Setup screen system console I/O redirection assignment.

7.1.161 OPTION_VIDEO_CUSTOMER Option

The **OPTION_VIDEO_CUSTOMER** option enables or disables code in the BIOS to drive an OEM-defined, custom video controller.

If you have a very custom video controller, then you may enable **OPTION_VIDEO_CUSTOMER**, and edit `SYSTEM\CUSTVID.ASM` to provide an equivalent driver for your own hardware.

The OEM-defined video driver does not require any other options unless the OEM specifically makes references to them in the code.

OPTION_SUPPORT_VIDEO must be enabled in order for INT 10h requests to be accepted from the user and routed to the driver.

If you have redirected console I/O with **CONFIG_CON_REDIR_STD**, **CONFIG_CON_REDIR_DEBUG**, or **CONFIG_CON_REDIR_SETUP**, then when redirection occurs, the driver does not receive control, unless **OPTION_VIDEO_DUPLICATE** is enabled. The latter option causes all output that is redirected to also appear on the video display.

Values:

1 - Enable OEM-defined video controller support.
0 - Disable OEM-defined video controller support.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video controller support.

OPTION_SUPPORT_CON_REDIRECTOR - Enable console I/O redirection.

CONFIG_CON_REDIR_STD - Standard console I/O redirection assignment.

CONFIG_CON_REDIR_DEBUG - Debugger console I/O redirection assignment.

CONFIG_CON_REDIR_SETUP - Setup screen system console I/O redirection assignment.

7.1.162 OPTION_VIDEO_DUPLICATE Option

The **OPTION_VIDEO_DUPLICATE** option enables or disables code in the BIOS to echo any output that the BIOS redirects over a serial port to also be displayed on the main video screen.

OPTION_SUPPORT_VIDEO must be enabled in order for INT 10h requests to be accepted from the user and routed to the video driver.

If you have redirected console I/O with **CONFIG_CON_REDIR_STD**, **CONFIG_CON_REDIR_DEBUG**, or **CONFIG_CON_REIR_SETUP**, then when redirection occurs, the driver does not receive control, unless **OPTION_VIDEO_DUPLICATE** is enabled. The latter option causes all output that is redirected to also appear on the video display.

Note that if you are using a VGA BIOS in your system, it may have hooked INT 10h, preventing the core BIOS from being able to route output to the redirection device. If this option appears to not correctly duplicate output in your system, pull the VGA BIOS to see if it corrects the problem.

Values:

1 - Enable dual video routing support.
0 - Disable dual video routing support.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video controller support.

OPTION_SUPPORT_CON_REDIRECTOR - Enable console I/O redirection.

CONFIG_CON_REDIR_STD - Standard console I/O redirection assignment.

CONFIG_CON_REDIR_DEBUG - Debugger console I/O redirection assignment.

CONFIG_CON_REDIR_SETUP - Setup screen system console I/O redirection assignment.

7.1.163 OPTION_VIDEO_VIDEOMEM Option

The **OPTION_VIDEO_VIDEOMEM** option enables or disables code in the BIOS to test video RAM during POST (and also in the Standard Diagnostics). The POST video RAM test is used to autodetect the type of video controller (color or monochrome) present in the system.

OPTION_SUPPORT_VIDEO must be enabled in order for INT 10h requests to be accepted from the user and routed to the driver.

OPTION_VIDEO_6845 must be enabled in order for the video RAM test to support 6845 video memory.

The segment addresses of video memory must be specified with three parameters in the project file. **CONFIG_VIDEO_SEG_GRAPHIC** selects the video RAM address for graphics modes. **CONFIG_VIDEO_SEG_MONO** selects the address for monochrome mode, and **CONFIG_VIDEO_SEG_COLOR** selects the color address.

Values:

1 - Enable video RAM testing support.
0 - Disable video RAM testing support.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable INT 10h video services.

OPTION_VIDEO_6845 - Enable 6845 video controller support.

CONFIG_VIDEO_SEG_GRAPHIC - Selects graphic mode video RAM segment.

CONFIG_VIDEO_SEG_MONO - Selects monochrome mode video RAM segment.

CONFIG_VIDEO_SEG_COLOR - Selects color video RAM segment.

7.1.164 OPTION_VIDEO_STDFONT Option

The **OPTION_VIDEO_STDFONT** option enables or disables the standard font tables for 6845-compatible CRT controllers operating in graphics modes. Some processors may have their own tables or may need to draw characters in different ways. In these cases, the standard fonts may need to be disabled.

Values:

1 - Enable standard video fonts for 6845 driver.
0 – Disable standard video fonts for 6845 driver.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video support in general.
OPTION_VIDEO_6845 - Enable standard PC/XT/AT (6845) video driver.

7.1.165 OPTION_VIDEO_MODE_REDIR Option

The **OPTION_VIDEO_MODE_REDIR** option enables or disables code in the BIOS's Console Redirection feature that causes mode set requests (INT 10h function 00h) to be translated to ASCII 0ch codes (form feed), allowing the screen to be cleared on a serial terminal.

Values:

1 - Enable redirection of set mode commands during console redirection.
0 – Disable redirection of set mode commands during console redirection.

Related Parameters:

OPTION_SUPPORT_CON_REDIRECTOR - Enable console redirection.
OPTION_SUPPORT_VIDEO – Enable video support.

7.1.166 OPTION_CRITICAL_BOARD Option

The **OPTION_CRITICAL_BOARD** option enables or disables code in the BIOS to pass control to the Board Personality Module's critical error handler when a critical error occurs during POST.

Examples of critical errors are RAM parity errors, or failures of interrupt controllers, DRAM refresh, etc.

This option requires that **BOARD** be set to the appropriate Board Personality Module identifier.

The default handler in the default Board Personality Module enters Manufacturing Mode (if enabled). See the section on **OPTION_SUPPORT_MFGMODE** for more information.

Values:

1 - Route critical error handling to board module.
0 - Don't route critical error handling to board module.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.
BOARD - Select the Board Personality Module.

7.1.167 OPTION_CRITICAL_BEEP Option

The **OPTION_CRITICAL_BEEP** option enables or disables code in the BIOS to beep the speaker when a critical error occurs during POST.

Examples of critical errors are RAM parity errors, or failures of interrupt controllers, DRAM refresh, etc.

This option requires that **OPTION_SUPPORT_SOUND** be enabled, and that **OPTION_SUPPORT_PORT_B** be enabled to have access to the speaker device.

The beep codes are defined in `INC\POSTERR.INC`.

Values:

1 - Enable speaker beep codes.
0 - Disable speaker beep codes.

Related Parameters:

OPTION_SUPPORT_SOUND - Enable sound support.
OPTION_SUPPORT_PORT_B - Enable access to speaker.

7.1.168 OPTION_CRITICAL_FLOPPY_LIGHT Option

The **OPTION_CRITICAL_FLOPPY_LIGHT** option enables or disables code in the BIOS to blink the drive light on the floppy when no speaker is available to beep the speaker when a critical error occurs during POST.

Examples of critical errors are RAM parity errors, or failures of interrupt controllers, DRAM refresh, etc.

This option does *not* require that **OPTION_SUPPORT_FLOPPY** be enabled. Instead, it only requires that a floppy disk controller (FDC) be available to respond to its I/O ports.

The beep codes are defined in `INC\POSTERR.INC`.

Values:

1 - Enable floppy light blinking codes.
0 - Disable floppy light blinking codes.

Related Parameters:

None.

7.1.169 OPTION_CRITICAL_MFGMODE Option

The **OPTION_CRITICAL_MFGMODE** option enables or disables code in the BIOS to enter Manufacturing Mode when a critical error occurs during POST.

Examples of critical errors are RAM parity errors, or failures of interrupt controllers, DRAM refresh, etc.

This option causes a best-effort attempt to make Manufacturing Mode work. Since a critical error has occurred prior to Manufacturing Mode being entered, it is possible that DRAM is not functional, or that other key system hardware components are not working.

Values:

1 - Enable invocation of Manufacturing Mode on critical errors.
0 - Disable invocation of Manufacturing Mode on critical errors.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

7.1.170 OPTION_CMOS_MOUSE Option

The **OPTION_CMOS_MOUSE** option specifies the factory default setting for the CMOS parameter that enables or disables the PS/2 mouse.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable PS/2 mouse.
0 - Disable PS/2 mouse.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_PS2MOUSE - Enable PS/2 mouse support.

7.1.171 OPTION_CMOS_TEST1MB Option

The **OPTION_CMOS_TEST1MB** option specifies the factory default setting for the CMOS parameter that enables or disables the testing of memory above 1MB during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable extended memory test during POST.
0 - Disable extended memory test during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_PROTECT_MODE - Enable extended memory test during POST.

7.1.172 OPTION_CMOS_TESTCLICK Option

The **OPTION_CMOS_TESTCLICK** option specifies the factory default setting for the CMOS parameter that enables or disables speaker clicks to indicate progress during POST memory tests.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable speaker clicks during POST memory tests.
0 - Disable speaker clicks during POST memory tests.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SOUND - Enable sound support.

7.1.173 OPTION_CMOS_PARITY Option

The **OPTION_CMOS_PARITY** option specifies the factory default setting for the CMOS parameter that enables or disables memory parity checking during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable memory parity checking.
0 - Disable memory parity checking.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_PARITY - Enable parity checking support.

7.1.174 OPTION_CMOS_DELETE Option

The **OPTION_CMOS_DELETE** option specifies the factory default setting for the CMOS parameter that enables or disables the “Press <DEL> to enter SETUP” message during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable “Press <DEL> ...” message during POST.
0 - Disable “Press <DEL> ...” message during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_POSTMSGS - Enable messages during POST.

7.1.175 OPTION_CMOS_HEXLOWER Option

The **OPTION_CMOS_HEXLOWER** option specifies the factory default setting for the CMOS parameter that enables or disables the display of hexadecimal numbers in lower-case alphabetics.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable lower-case hex displays.
0 - Disable lower-case hex displays (always upper-case).

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.1.176 OPTION_CMOS_F1ERROR Option

The **OPTION_CMOS_F1ERROR** option specifies the factory default setting for the CMOS parameter that enables or disables the “Press F1 to continue” message during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable “Press F1 to continue” message during POST.
0 - Disable “Press F1 to continue” message during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_POSTMSGS - Enable messages during POST.

7.1.177 OPTION_CMOS_NUMLOCK Option

The **OPTION_CMOS_NUMLOCK** option specifies the factory default setting for the CMOS parameter that controls the initial state of the NumLock key during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable NumLock key during POST.
0 - Disable NumLock key during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_KEYBOARD - Enable keyboard support.

7.1.178 OPTION_CMOS_TYPEMATIC Option

The **OPTION_CMOS_TYPEMATIC** option specifies the factory default setting for the CMOS parameter that enables or disables typematic programming during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable typematic programming during POST.
0 - Disable typematic programming during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_KEYBOARD - Enable keyboard support.
CONFIG_CMOS_TYPEMATIC_DELAY - Typematic delay for programming.
CONFIG_CMOS_TYPEMATIC_RATE - Typematic rate for programming.

7.1.179 OPTION_CMOS_WEITEK Option

The **OPTION_CMOS_WEITEK** option specifies the factory default setting for the CMOS parameter that enables or disables support for Weitek coprocessors during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable Weitek coprocessor programming during POST.
0 - Disable Weitek coprocessor programming during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.1.180 OPTION_CMOS_FLOPPYSEEK Option

The **OPTION_CMOS_FLOPPYSEEK** option specifies the factory default setting for the CMOS parameter that enables or disables the initial head seek of configured floppy drives during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable floppy disk drive head seek during POST.
0 - Disable floppy disk drive head seek during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_FLOPPY - Enable floppy support.

7.1.181 OPTION_CMOS_EXTCACHE Option

The **OPTION_CMOS_EXTCACHE** option specifies the factory default setting for the CMOS parameter that enables or disables the external (L2) cache during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable L2 cache during POST.
0 - Disable L2 cache during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_CACHE - Enable L2 cache support.

7.1.182 OPTION_CMOS_INTCACHE Option

The **OPTION_CMOS_INTCACHE** option specifies the factory default setting for the CMOS parameter that enables or disables the internal (L1) cache during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable L1 cache during POST.
0 - Disable L1 cache during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_CACHE_CPU - Enable L1 cache support.

7.1.183 OPTION_CMOS_FASTA20 Option

The **OPTION_CMOS_FASTA20** option specifies the factory default setting for the CMOS parameter that enables or disables the fast A20 gate during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable fast A20 gate during POST.
0 - Disable fast A20 gate during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_A20_CPU - Enable CPU A20 support.
OPTION_A20_CHIPSET - Enable Chipset A20 support.
OPTION_A20_BOARD - Enable Board A20 support.
OPTION_A20_PORT92 - Enable Port 92h A20 support.

7.1.184 OPTION_CMOS_HDSEEK Option

The **OPTION_CMOS_HDSEEK** option specifies the factory default setting for the CMOS parameter that enables or disables the seek of the configured hard disk heads during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable hard disk seek during POST.
0 - Disable hard disk seek during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_IDE - Enable IDE support.

7.1.185 OPTION_CMOS_CONFIGBOX Option

The **OPTION_CMOS_CONFIGBOX** option specifies the factory default setting for the CMOS parameter that enables or disables the display of the configuration box during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable configuration box display during POST.
0 - Disable configuration box display during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_CONFIGBOX - Enable Configuration Box support.

7.1.186 OPTION_CMOS_EXHMEMTEST Option

The **OPTION_CMOS_EXHMEMTEST** option specifies the factory default setting for the CMOS parameter that enables or disables the invocation of exhaustive memory tests during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable exhaustive memory tests during POST.
0 - Disable exhaustive memory tests during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_EXHMEMTEST - Enable exhaustive memory test support.

7.1.187 OPTION_CMOS_PASSWORD Option

The **OPTION_CMOS_PASSWORD** option specifies the factory default setting for the CMOS parameter that enables or disables password checking during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable password checking during POST.
0 - Disable password checking during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_PASSWORD - Enable password support.
OPTION_SETUP_PASSWORD - Enable password Setup screen.

7.1.188 OPTION_CMOS_KEYBOARD Option

The **OPTION_CMOS_KEYBOARD** option specifies the factory default setting for the CMOS parameter that enables or disables keyboard tests during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable keyboard testing during POST.
0 - Disable keyboard testing during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_KEYBOARD - Enable keyboard support.

7.1.189 OPTION_CMOS_SHADOW_ENABLE Option

The **OPTION_CMOS_SHADOW_ENABLE** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing during POST.

0 - Disable shadowing during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_C000 - Enable shadowing at segment C000h.
OPTION_CMOS_SHADOW_C400 - Enable shadowing at segment C400h.
OPTION_CMOS_SHADOW_C800 - Enable shadowing at segment C800h.
OPTION_CMOS_SHADOW_CC00 - Enable shadowing at segment CC00h.
OPTION_CMOS_SHADOW_D000 - Enable shadowing at segment D000h.
OPTION_CMOS_SHADOW_D400 - Enable shadowing at segment D400h.
OPTION_CMOS_SHADOW_D800 - Enable shadowing at segment D800h.
OPTION_CMOS_SHADOW_DC00 - Enable shadowing at segment DC00h.
OPTION_CMOS_SHADOW_E000 - Enable shadowing at segment E000h.
OPTION_CMOS_SHADOW_E400 - Enable shadowing at segment E400h.
OPTION_CMOS_SHADOW_E800 - Enable shadowing at segment E800h.
OPTION_CMOS_SHADOW_EC00 - Enable shadowing at segment EC00h.
OPTION_CMOS_SHADOW_F000 - Enable shadowing at segment F000h.

7.1.190 OPTION_CMOS_SHADOW_C000 Option

The **OPTION_CMOS_SHADOW_C000** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at C000h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.191 OPTION_CMOS_SHADOW_C400 Option

The **OPTION_CMOS_SHADOW_C400** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at C400h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.192 OPTION_CMOS_SHADOW_C800 Option

The **OPTION_CMOS_SHADOW_C800** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at C800h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.193 OPTION_CMOS_SHADOW_CC00 Option

The **OPTION_CMOS_SHADOW_CC00** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at CC00h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.194 OPTION_CMOS_SHADOW_D000 Option

The **OPTION_CMOS_SHADOW_D000** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at D000h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.195 OPTION_CMOS_SHADOW_D400 Option

The **OPTION_CMOS_SHADOW_D400** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at D400h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.196 OPTION_CMOS_SHADOW_D800 Option

The **OPTION_CMOS_SHADOW_D800** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at D800h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.197 OPTION_CMOS_SHADOW_DC00 Option

The **OPTION_CMOS_SHADOW_DC00** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at DC00h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.198 OPTION_CMOS_SHADOW_E000 Option

The **OPTION_CMOS_SHADOW_E000** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at E000h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.199 OPTION_CMOS_SHADOW_E400 Option

The **OPTION_CMOS_SHADOW_E400** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at E400h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.200 OPTION_CMOS_SHADOW_E800 Option

The **OPTION_CMOS_SHADOW_E800** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at E800h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.201 OPTION_CMOS_SHADOW_EC00 Option

The **OPTION_CMOS_SHADOW_EC00** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 16K segment at EC00h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of specified segment during POST.
0 - Disable shadowing of specified segment during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.202 OPTION_CMOS_SHADOW_F000 Option

The **OPTION_CMOS_SHADOW_F000** option specifies the factory default setting for the CMOS parameter that enables or disables shadowing at the 64K segment at F000h during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable shadowing of full 64KB segment at F000h during POST.
0 - Disable shadowing of full 64KB segment at F000h during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_SHADOW - Enable shadowing support.
OPTION_CMOS_SHADOW_ENABLE - Enable shadowing during POST.

7.1.203 OPTION_CMOS_SPEED Option

The **OPTION_CMOS_SPEED** option specifies the factory default setting for the CMOS parameter that enables or disables high CPU speed during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable high CPU speed during POST.
0 - Disable high CPU speed during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SPEED_CPU - Enable CPU module speed support.
OPTION_SPEED_CHIPSET - Enable chipset module speed support.
OPTION_SPEED_BOARD - Enable board module speed support.

7.1.204 OPTION_CMOS_REFRESH Option

The **OPTION_CMOS_ REFRESH** option specifies the factory default setting for the CMOS parameter that enables or disables automatic DRAM refresh during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable automatic DRAM refresh during POST.
0 - Disable automatic DRAM refresh during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_REFRESH_8237 - Enable traditional DMA-based refresh support.
OPTION_REFRESH_CPU - Enable CPU module refresh support.
OPTION_REFRESH_CHIPSET - Enable chipset module refresh support.
OPTION_REFRESH_BOARD - Enable board module refresh support.
OPTION_REFRESH_CHARGE - Enable charging of DRAMs after refresh starts.

7.1.205 OPTION_CMOS_POWER Option

The **OPTION_CMOS_ POWER** option specifies the factory default setting for the CMOS parameter that enables or disables power management during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable power management during POST.
0 - Disable power management during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_POWERMAN - Enable power management support.

7.1.206 OPTION_CMOS_ATA Option

The **OPTION_CMOS_ ATA** option specifies the factory default setting for the CMOS parameter that enables or disables ATA card support during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable ATA card during POST.
0 - Disable ATA card during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.

7.1.207 OPTION_CMOS_RFD Option

The **OPTION_CMOS_ RFD** option specifies the factory default setting for the CMOS parameter that enables or disables the Resident Flash Disk (RFD) during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS must be enabled for this option to work.

Values:

1 - Enable RFD during POST.
0 - Disable RFD during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_RFD_DISK - Enable RFD support.

7.1.208 OPTION_CMOS_LOAD_WINCE Option

The **OPTION_CMOS_ LOAD_WINCE** option specifies the factory default setting for the CMOS parameter that enables or disables the loading of Windows CE during POST.

This option does not enable or disable the assembly of code in the BIOS; instead, it specifies the default use of the assembled code; therefore, the actual feature must be enabled with other parameters for this option to be useful.

OPTION_SUPPORT_CMOS and **OPTION_SUPPORT_WINCE** must be enabled for this option to work.

Values:

1 - Enable RFD during POST.
0 - Disable RFD during POST.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_SUPPORT_RFD_DISK - Enable RFD support.

7.1.209 OPTION_HARDERR_A20 Option

The **OPTION_HARDERR_A20** option specifies whether POST should consider failure during the A20 test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.210 OPTION_HARDERR_DISSHADOW Option

The **OPTION_HARDERR_DISSHADOW** option specifies whether POST should consider failure during disabling shadowing as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.211 OPTION_HARDERR_KBDCTRL Option

The **OPTION_HARDERR_KBDCTRL** option specifies whether POST should consider failure during the keyboard controller test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.212 OPTION_HARDERR_CMOS Option

The **OPTION_HARDERR_CMOS** option specifies whether POST should consider failure during the CMOS RAM test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.213 OPTION_HARDERR_PCI Option

The **OPTION_HARDERR_PCI** option specifies whether POST should consider failure during the initialization of the PCI bus as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.

0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.214 OPTION_HARDERR_TIMER Option

The **OPTION_HARDERR_TIMER** option specifies whether POST should consider failure during the timer test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.215 OPTION_HARDERR_REFRESH Option

The **OPTION_HARDERR_TIMER** option specifies whether POST should consider failure during the DRAM refresh test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.

OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.216 OPTION_HARDERR_MEMCFG Option

The **OPTION_HARDERR_MEMCFG** option specifies whether POST should consider failure during the memory geometry detection as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.217 OPTION_HARDERR_BASEMEM Option

The **OPTION_HARDERR_BASEMEM** option specifies whether POST should consider failure during the base (64K or less) memory test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.218 OPTION_HARDERR_DMA Option

The **OPTION_HARDERR_DMA** option specifies whether POST should consider failure during the DMA controller test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.219 OPTION_HARDERR_INT Option

The **OPTION_HARDERR_INT** option specifies whether POST should consider failure during the interrupt controller test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.220 OPTION_HARDERR_FIRMWARE Option

The **OPTION_HARDERR_FIRMWARE** option specifies whether POST should consider failure during the downloading of OEM-proprietary firmware as a critical error that stops POST.

If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.221 OPTION_HARDERR_KBD Option

The **OPTION_HARDERR_KBD** option specifies whether POST should consider failure during the keyboard test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.222 OPTION_HARDERR_VIDEO Option

The **OPTION_HARDERR_VIDEO** option specifies whether POST should consider failure during the video test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.223 OPTION_HARDERR_PSWD Option

The **OPTION_HARDERR_PSWD** option specifies whether POST should consider failure during the password checking as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.224 OPTION_HARDERR_LOWMEM Option

The **OPTION_HARDERR_LOWMEM** option specifies whether POST should consider failure during the low (<1MB) memory test as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.

0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.225 OPTION_HARDERR_PROTMODE Option

The **OPTION_HARDERR_PROTMODE** option specifies whether POST should consider failure during the protected mode tests as a critical error that stops POST. If enabled, failure results in an error. If disabled, failure causes POST to continue, although system operation may not be normal.

The BIOS can be configured to handle critical errors in several ways, including entering Manufacturing Mode, beeping the speaker, blinking the floppy drive light, or calling OEM-proprietary code in the Board Personality Module.

Values:

1 - Enable this critical error during POST.
0 - Disable this critical error during POST.

Related Parameters:

OPTION_CRITICAL_BEEP - Beep speaker on critical errors.
OPTION_CRITICAL_FLOPPY_LIGHT - Blink floppy light on critical errors.
OPTION_CRITICAL_BOARD - Call board module on critical errors.
OPTION_CRITICAL_MFGMODE - Enter Manufacturing Mode on critical errors.

7.1.226 OPTION_SOFTERR_SETUP Option

The **OPTION_SOFTERR_SETUP** option enables or disables code in the BIOS that enables routing of soft errors to the SETUP screen system.

Examples of soft errors are corrupt CMOS, low battery indications, keyboard errors, and similar things that are usually correctable by the user in SETUP. All errors that are not critical errors (such as RAM parity errors, etc.) are soft errors.

OPTION_SUPPORT_SETUP must be enabled for this option to work.

Values:

1 - Enable soft error routing to SETUP screen.
0 - Disable soft error routing to SETUP screen.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable SETUP screen system.

7.1.227 OPTION_SOFTERR_LPT Option

The **OPTION_SOFTERR_LPT** option enables or disables code in the BIOS to generate a soft error if missing LPT ports are encountered.

Values:

1 - Enable strict LPT port checking.
0 - Disable strict LPT port checking.

Related Parameters:

OPTION_SUPPORT_PARALLEL - Enable parallel port support.

7.1.228 OPTION_SOFTERR_MEMMIS Option

The **OPTION_SOFTERR_MEMMIS** option enables or disables code in the BIOS to generate a soft error if a CMOS memory size mismatch is detected.

Values:

1 - Enable CMOS memory size mismatch soft error.
0 - Disable CMOS memory size mismatch soft error.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.1.229 OPTION_SOFTERR_BOARD Option

The **OPTION_SOFTERR_BOARD** option enables or disables code in the BIOS to generate a soft error if the board module detects one.

The board module should unconditionally report soft errors to the core by setting the **POST_ERR_BOARD** bit in the **PostErrFlag2** Extended BIOS Data Area field. The core uses this option to determine whether this soft error is reported to the user.

Values:

1 - Enable board-level soft errors.
0 - Disable board-level soft errors.

Related Parameters:

None.

7.1.230 OPTION_SOFTERR_CHIPSET Option

The **OPTION_SOFTERR_CHIPSET** option enables or disables code in the BIOS to generate a soft error if the chipset module detects one.

The chipset module should unconditionally report soft errors to the core by setting the **POST_ERR_CHIPSET** bit in the **PostErrFlag2** Extended BIOS Data Area field. The core uses this option to determine whether this soft error is reported to the user.

Values:

1 - Enable chipset-level soft errors.
0 - Disable chipset-level soft errors.

Related Parameters:

None.

7.1.231 OPTION_SOFTERR_CPU Option

The **OPTION_SOFTERR_CPU** option enables or disables code in the BIOS to generate a soft error if the CPU module detects one.

The CPU module should unconditionally report soft errors to the core by setting the **POST_ERR_CPU** bit in the **PostErrFlag2** Extended BIOS Data Area field. The core uses this option to determine whether this soft error is reported to the user.

Values:

1 - Enable CPU-level soft errors.
0 - Disable CPU-level soft errors.

Related Parameters:

None.

7.1.232 OPTION_QUERY_ENTERSETUP Option

The **OPTION_QUERY_ENTERSETUP** option enables or disables code in the BIOS to ask the user if he wants to enter the Setup system.

Values:

1 - Enable query.
0 - Disable query; automatically perform action.

Related Parameters:

OPTION_SUPPORT_SETUP - Enable Setup screen support.

7.1.233 OPTION_QUERY_FORMATRFD Option

The **OPTION_QUERY_FORMATRFD** option enables or disables code in the BIOS to ask the user if he wants to reformat the RFD during POST, if POST has determined that the RFD is unformatted.

Values:

1 - Enable query.
0 - Disable query; do not format RFD.

Related Parameters:

OPTION_SUPPORT_RFD_DISK - Enable RFD support.

7.1.234 OPTION_QUERY_VERIFYRFD Option

The **OPTION_QUERY_VERIFYRFD** option enables or disables code in the BIOS to ask the user if he wants to check the integrity of the RFD and fix any discovered problems during POST. Normally this integrity check is always done during POST, if the RFD is to be used after the operating system has booted. The only reason to bypass this (and hence make this option useful) is for a test lab environment, where the RFD should not always be autoinitialized by the system until the Flash I/O has been debugged.

Values:

1 - Enable query.
0 - Disable query; automatically verify RFD integrity & fix problems.

Related Parameters:

OPTION_SUPPORT_RFD_DISK - Enable RFD support.

7.1.235 OPTION_QUERY_FORMATRAM Option

The **OPTION_QUERY_FORMATRAM** option enables or disables code in the BIOS to ask the user if he wants to format the RAM disk during POST. If this parameter is enabled, and the user responds affirmatively, the RAM disk will be initialized. If the parameter is enabled and the user responds negatively, or if the parameter is disabled, then the RAM disk will not be explicitly formatted.

This does not affect the logic that determines if a valid RAM disk image is detected. During RAM disk initialization, separately from the query discussed above, the RAM disk server checks the BIOS Parameter Block in the RAM disk's boot record to determine if the disk is valid. If not, the RAM disk is automatically formatted in any case.

Values:

1 - Enable query.
0 - Disable query; do not format unless RAM disk is uninitialized.

Related Parameters:

OPTION_SUPPORT_RAM_DISK - Enable RFD support.

7.1.236 OPTION_QUERY_DEBUG Option

The **OPTION_QUERY_DEBUG** option enables or disables code in the BIOS to ask the user if he wants to enter the debugger before POST completes and boots the operating system.

Values:

1 - Enable query.
0 - Disable query; do not enter the debugger.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable debugger support.

7.1.237 OPTION_MFGMODE_TIMEOUT Option

The **OPTION_MFGMODE_TIMEOUT** option enables or disables code in the BIOS that causes POST to time-out the Manufacturing Mode if the test mode bit drops once Manufaturing Mode is entered.

This feature allows consumer electronic devices that are being managed by Manufacturing Mode to resume normal operations when they are removed from the field-support test hardware. Typically, this is an RS-232 signal such as DTR that goes low when the Manufacturing Mode cable is disconnected from the target.

The actual method by which the hardware is tested is OEM-specific, and handled with a call to the Board Personality Module (BPM). See Chapter 20 for details.

OPTION_SUPPORT_MFGMODE must be enabled for Manufacturing Mode to be properly supported. See the notes associated with this option for futher information about Manufacturing Mode.

Values:

1 - Enable Manufacturing Mode timeout handling.
0 - Disable Manufacturing Mode timeout handling.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode.
OPTION_MFGMODE_FIFO - Enable target UART’s FIFO during Manufacturing Mode.

7.1.238 OPTION_MFGMODE_FIFO Option

The **OPTION_MFGMODE_FIFO** option enables or disables code in the BIOS that enables the FIFO of the UART used during Manufacturing Mode.

This feature can significantly improve performance on targets that have working FIFOs in their UARTs, because interrupts are not delivered on a per-byte basis, but instead when the UART's high water mark is reached.

OPTION_SUPPORT_MFGMODE must be enabled for Manufacturing Mode to be properly supported. See the notes associated with this option for futher information about Manufacturing Mode.

Values:

1 - Enable Manufacturing Mode FIFO support.
0 - Disable Manufacturing Mode FIFO support.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode.
OPTION_MFGMODE_TIMEOUT - Enable Manufacturing Mode timeout handling.

7.1.239 OPTION_MEMTEST_LOW_POST Option

The **OPTION_MEMTEST_LOW_POST** option enables or disables code in the BIOS that causes POST to use a more extensive, exhaustive memory test on low memory below 1MB.

OPTION_SUPPORT_EXHMEMTEST must be enabled for this option to be valid so that the exhaustive memory test software is enabled.

Values:

1 - Enable exhaustive testing of low memory during POST.
0 - Disable exhaustive testing of low memory during POST.

Related Parameters:

OPTION_SUPPORT_EXHMEMTEST - Enable exhaustive memory test code.

7.1.240 OPTION_MEMTEST_HIGH_POST Option

The **OPTION_MEMTEST_HIGH_POST** option enables or disables code in the BIOS that causes POST to use a more extensive, exhaustive memory test on high memory above 1MB (extended memory).

OPTION_SUPPORT_EXHMEMTEST must be enabled for this option to be valid so that the exhaustive memory test software is enabled.

Additionally, support for extended memory requires that **OPTION_SUPPORT_PROTECT_MODE** be enabled. There are other considerations to make with respect to protected mode; see the details under the **OPTION_SUPPORT_PROTECT_MODE** option for more information.

Values:

1 - Enable exhaustive testing of high memory during POST.
0 - Disable exhaustive testing of high memory during POST.

Related Parameters:

OPTION_SUPPORT_EXHMEMTEST - Enable exhaustive memory test code.
OPTION_SUPPORT_PROTECT_MODE - Enable protected mode and extended memory support.

7.1.241 OPTION_MEMTEST_WAIT Option

The **OPTION_MEMTEST_WAIT** option enables or disables code in the BIOS that causes POST to pause between tested memory blocks so that the user has a chance to hit the <DEL> key or <ESC> key during the memory test.

This option is used both with exhaustive memory tests and with the standard memory tests.

The **CONFIG_WAIT_COUNT** configuration parameter is used to configure the length of the delay. This is units of CPU loops, so it is best to start with the default value and adjust it as necessary for the performance of your CPU.

Values:

1 - Enable pauses between tested memory blocks during POST.
0 - Disable pauses between tested memory blocks during POST.

Related Parameters:

CONFIG_WAIT_COUNT - Specifies the amount of time to wait between tested blocks.

7.1.242 OPTION_MEMTEST_CLEAR Option

The **OPTION_MEMTEST_CLEAR** option enables or disables code in the BIOS that causes POST to store a 00h pattern into each byte of low memory, to prevent MS-DOS from failing.

The problem is that some MS-DOS utility programs erroneously use values from memory locations that are uninitialized in areas that they do not own. These utility programs fail when the values read are not zeroes.

If you are running MS-DOS on the target, you should enable this option. Otherwise, it should be disabled to save code space and valuable time during POST.

Values:

1 - Enable zero-fill of low memory during POST.
0 - Disable zero-fill of low memory during POST.

Related Parameters:

None.

7.1.243 OPTION_MEMTEST_CLICK Option

The **OPTION_MEMTEST_CLICK** option enables or disables code in the BIOS that causes POST to click the speaker after testing each block during POST.

This option is used both with exhaustive memory tests and with the standard memory tests.

Values:

1 - Enable clicks between tested memory blocks during POST.
0 - Disable clicks between tested memory blocks during POST.

Related Parameters:

None.

7.1.244 OPTION_MEMTEST_QUICK Option

The **OPTION_MEMTEST_QUICK** option enables or disables code in the BIOS that optimizes POST's memory test for time by only testing the first word of each 1KB unit of memory.

Values:

1 - Enable memory test optimization.
0 – Disable memory test optimization.

Related Parameters:

OPTION_MEMTEST_CLEAR – Clear memory blocks to 0 during test.
OPTION_MEMTEST_CLICK – Click speaker during test.
OPTION_MEMTEST_HIGH_POST – Test high memory exhaustively.
OPTION_MEMTEST_LOW_POST – Test low memory exhaustively.
OPTION_MEMTEST_WAIT – Pause between each tested block during test.

7.1.245 OPTION_RFD_TESTFREE Option

The **OPTION_RFD_TESTFREE** option enables or disables code in the BIOS that the Resident Flash Disk (RFD) uses to verify that Flash blocks marked "free" or "deleted" do in fact contain nothing but bytes with the value ffh. If errors are found, the offending free areas are marked bad.

This option requires that at least one RFD in the system be defined with the **FILE_SYSTEM** macro.

Values:

1 - Enable check of free block contents during POST.
0 - Disable check of free block contents during POST.

Related Parameters:

FILE_SYSTEM - Define file system.

7.1.246 OPTION_RFD_FAT_SNOOP Option

The **OPTION_RFD_FAT_SNOOP** option enables or disables code in the BIOS that the Resident Flash Disk (RFD) uses to optimize writes to Flash. The result is a substantial improvement in sustained INT 13h write performance up to a factor of 10.

Without FAT snooping, an RFD must maintain the contents of previously-written clusters, even when the directory entries for the files written to those clusters have long since been deleted. Consider that an UNDELETE utility may be able to restore the contents of a previously-deleted file by finding these "deleted" clusters and chaining them together again. While UNDELETE is useful in the desktop PC environment, it is not so important in embedded designs, where performance is paramount. The FAT snooping performance optimization causes the RFD to detect writes to the FATs on the disk which free-up previously-written clusters. When this condition is detected, the freed clusters' sectors are marked dead, so that they may be reclaimed for reuse on the next write.

This performance enhancement is supported for both soft and hard-formatted RFDs in versions 4.3 and beyond, whereas prior versions only supported FAT snooping on soft-formatted RFDs. See the **FILE_SYSTEM** macro for details.

This option requires that at least one RFD in the system be defined with the **FILE_SYSTEM** macro.

Values:

1 - Enable FAT snooping for RFDs.
0 - Disable FAT snooping for RFDs.

Related Parameters:

FILE_SYSTEM - Define file system.

7.1.247 OPTION_DEBUG_HOTKEY Option

The **OPTION_DEBUG_HOTKEY** option enables or disables code in the BIOS to intercept the Control (Ctl) and Left Shift (Shf) key chord in the keyboard BIOS as a command to enter ("break into") the debugger. This allows the user to break into the debugger at any time interrupts are enabled.

This option requires PC, PC/XT, or PC/AT keyboard support by enabling **OPTION_KEYBOARD_PCAT**, since it intercepts the IRQ1 handler at interrupt vector 09h and reads shift flags in the BIOS data area.

OEM-defined keyboard modules may choose to review the **Int09Isr** code in SYSTEM\KEYBOARD.ASM and derive their own way of entering the debugger, if necessary.

Values:

1 - Enable debugger hotkey support.
0 - Disable debugger hotkey support.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

OPTION_KEYBOARD_PCAT - Use PC/XT or PC/AT keyboard.

7.1.248 OPTION_DEBUG_FLASH Option

The **OPTION_DEBUG_FLASH** option enables or disables code in the BIOS that provides the Erase Flash (EFL), Read Flash (RFL), Write Flash (WFL), Update Flash (UFL) and Set Flash (SFL) commands in the integrated BIOS debugger.

This allows the OEM to test Flash drivers and hardware with the debugger.

This option requires the **OPTION_SUPPORT_MCL** option to be enabled, and a valid media table to be defined with the **MEDIA_REGION** macro.

Values:

1 - Enable Flash commands in debugger.
0 - Disable Flash commands in debugger.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

OPTION_SUPPORT_MCL - Enable Flash programming support.

MEDIA_REGION - Macro used to define media table for media types.

7.1.249 OPTION_DEBUG_WATCHINT Option

The **OPTION_DEBUG_WATCHINT** option enables or disables code in the BIOS that supports the WATCH command in the integrated BIOS debugger.

The WATCH command allows the OEM to instruct the debugger to display the contents of the general registers on entry and on exit to a BIOS service interrupt routine.

The **OPTION_DEBUG_WATCHINT** option causes additional code to be compiled at the beginning and end of every service routine supporting the BIOS APIs; this code calls the debugger to notify it so that a trace can be displayed.

Enabling this option does degrade performance because every service routine performs extra work in anticipation of providing the debugging information. This option should be disabled in a production system.

Values:

1 - Enable Watch command in debugger.
0 - Disable Watch command in debugger.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

7.1.250 OPTION_DEBUG_NMI Option

The **OPTION_DEBUG_NMI** option enables or disables code in the BIOS that causes NMI interrupts to enter the debugger.

NMIs can be generated in an ISA system by manipulating the I/O check line on the bus. Common "break-out switches" do exactly this, effectively providing a hardware way to break into the debugger, even when a software method such as using a keystroke combination on the keyboard is unable to.

Because the NMI interrupt is nonmaskable, this allows the debugger to be used to debug real system hangs that leave interrupts disabled.

Values:

1 - Enable NMI debugger support.
0 - Disable NMI debugger support.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

7.1.251 OPTION_DEBUG_PCMCIA Option

The **OPTION_DEBUG_PCMCIA** option enables or disables code in the BIOS that provides PCMCIA debugging commands. In particular, the debugger command, CIS, is supported.

Values:

1 - Enable PCMCIA debugger support.
0 - Disable PCMCIA debugger support.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

7.1.252 OPTION_DEBUG_ASSEMBLY Option

The **OPTION_DEBUG_ASSEMBLY** option enables or disables code in the BIOS that provides a disassembler that can translate raw bytes into 80386 mnemonics.

Values:

1 - Enable debugger disassembler.
0 - Disable debugger disassembler.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

7.1.253 OPTION_DEBUG_EDOSROM Option

The **OPTION_DEBUG_EDOSROM** option enables or disables code in the BIOS that provides a facility to selectively, at run time, enable or disable the execution of XPRINTF macros for debugging inside the Embedded DOS-ROM kernel. Normally, this facility is only used at General Software.

Values:

1 - Enable debugger disassembler.
0 - Disable debugger disassembler.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

7.1.254 OPTION_DEBUG_CHIPSET Option

The **OPTION_DEBUG_CHIPSET** option enables or disables code in the BIOS that allows the debugger to support the CSR (chipset read) and CSW (chipset write) debugger commands.

If this option is enabled, the commands are enabled in the debugger, and calls are made to **CsReadReg** and **CsWriteReg** routines in the chipset module, respectively. The chipset module must implement these routines in order for the debugger commands to work properly; by default, these routines do not write or read a chipset register.

Values:

1 - Enable debugger chipset read/write commands.
0 - Disable debugger chipset read/write commands.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable integrated BIOS debugger.

7.1.255 OPTION_FLOPPY_SEEK Option

The **OPTION_FLOPPY_SEEK** option enables or disables code in the BIOS that can seek the floppy disk drive heads during POST. This POST action is used to determine whether the floppy disk drives are functional or not.

This option only supports the seeking feature; it must be enabled in CMOS via SETUP screen for the seek to actually occur. In order to permanently enable this feature, the **OPTION_CMOS_FLOPPYSEEK** option must be set.

Failure to seek properly causes a soft error to occur.

Values:

1 - Enable POST floppy seek feature.
0 - Disable POST floppy seek feature.

Related Parameters:

FILE_SYSTEM - Define file system for floppy support.

OPTION_CMOS_FLOPPYSEEK - Factory default value for floppy option.

7.1.256 OPTION_FLOPPY_DMA Option

The **OPTION_FLOPPY_DMA** option enables or disables code in the BIOS that supports floppy I/O with DMA transfers. If this option is enabled, then DMA transfers are supported. This is the normal method of transferring data between RAM and the floppy disk controller.

Some systems do not have PC-compatible 8237A DMA controllers. These systems therefore cannot use DMA-based floppy disk I/O. Instead, they use a polled approach that requires a fairly high CPU performance to sustain a 500KB/second transfer rate with error checking on each byte. If you are supporting a target that must have floppy I/O without DMA support, then this option should be disabled. Otherwise, in all other circumstances, this option should be enabled if you are supporting floppy disk I/O.

The **OPTION_FLOPPY_FAST_POLL** option is used when **OPTION_FLOPPY_DMA** is disabled. Fast polling uses an in-line instruction sequence that avoids a few pushes, pops, calls, and returns. In some cases, this can make the difference between supporting and not supporting a polled approach on slower targets.

OPTION_FLOPPY_POLL_ERRORS should really be set when **OPTION_FLOPPY_DMA** is disabled. Error polling is used to check the status port before reading data from the floppy

disk controller. This allows the floppy disk code to determine if an error is occurring during a data transfer. Without this support, significant timeouts can occur, and in some cases, lockups can occur.

The **OPTION_FLOPPY_82077** option should be set whenever an 82077A or 82078 floppy disk controller is used, to take advantage of the built-in FIFO. This does not solve polled I/O throughput problems, but it can smooth-out situations where a slight delay would ordinarily have caused a single byte to be missed.

If you are using polling without error detection, then **OPTION_FLOPPY_144_ONLY** can help eliminate superfluous errors. The floppy disk state machine tries to recognize different media in the drive, because the user can insert different media. The floppy disk controller must be programmed appropriately, so the test is necessary in order to support both 720KB and 1.44MB floppy disks in a 3.5" drive. If you are confident that your target will only support 1.44MB floppy disks in your application, then you can enable **OPTION_FLOPPY_144_ONLY** and avoid this guessing game played by the floppy disk driver.

Polling the floppy disk controller is such a time-critical operation that interrupts must be disabled during the transfers. Thus, no keyboard activity is recognized during floppy I/O, and the system's time of day is not maintained accurately.

Values:

1 - Enable DMA-based floppy I/O.
0 - Disable DMA-based floppy I/O (poll instead).

Related Parameters:

FILE_SYSTEM - Define file system for floppy support.

OPTION_FLOPPY_82077 - Enable floppy FIFO support.

OPTION_FLOPPY_FAST_POLL - Enable in-line polling code.

OPTION_FLOPPY_POLL_ERRORS - Enable error detection for non-DMA operation.

OPTION_FLOPPY_144_ONLY - Only support 1.44MB floppies in 3.5" drives.

7.1.257 OPTION_FLOPPY_82077 Option

The **OPTION_FLOPPY_82077** option enables or disables code in the BIOS that supports the FIFO in Intel 82077A or 82078 floppy disk controllers.

The **OPTION_FLOPPY_82077** option should be set whenever an 82077A or 82078 floppy disk controller is used, to take advantage of the built-in FIFO. This does not solve polled I/O throughput problems, but it can smooth-out situations where a slight delay would ordinarily have caused a single byte to be missed.

Values:

1 - Enable FIFO in floppy disk controller.
0 - Disable FIFO in floppy disk controller.

Related Parameters:

FILE_SYSTEM - Define file system for floppy support.

7.1.258 OPTION_FLOPPY_WATCHIO Option

The **OPTION_FLOPPY_WATCHIO** option enables or disables code in the BIOS that displays the general registers on entry and on exit to the floppy disk service routine. This allows debugging of floppy disk I/O on targets that are having trouble supporting it.

Values:

1 - Enable debugging code in floppy disk service routine.
0 - Disable debugging code in floppy disk service routine.

Related Parameters:

FILE_SYSTEM - Define file system for floppy support.

7.1.259 OPTION_FLOPPY_FAST_POLL Option

The **OPTION_FLOPPY_FAST_POLL** option is used when **OPTION_FLOPPY_DMA** is disabled. Fast polling uses an in-line instruction sequence that avoids a few pushes, pops, calls, and returns.

If disabled, then some code space is saved, at the expense of slower execution time.

Polled floppy disk I/O is an extremely time-critical operation. In some cases, this can make the difference between supporting and not supporting a polled approach on slower targets.

Values:

1 - Enable fast polling code in floppy disk service routine.
0 - Disable fast polling code in floppy disk service routine.

Related Parameters:

FILE_SYSTEM - Define file system for floppy support.
OPTION_FLOPPY_DMA - Enable DMA-based floppy I/O.

7.1.260 OPTION_FLOPPY_POLL_ERRORS Option

The **OPTION_FLOPPY_POLL_ERRORS** option causes code to be generated in the BIOS that checks for error conditions when polling the floppy disk controller in a polled I/O mode.

This option is only valid when **OPTION_FLOPPY_DMA** is disabled. Error polling is used to check the status port before reading data from the floppy disk controller. This allows the floppy disk code to determine if an error is occurring during a data transfer. Without this support, significant timeouts can occur, and in some cases, lockups can occur.

Polled floppy disk I/O is an extremely time-critical operation. In some cases, this can make the difference between supporting and not supporting a polled approach on slower targets.

Values:

1 - Enable error detection in floppy disk service routine.
0 - Disable error detection in floppy disk service routine.

Related Parameters:

FILE_SYSTEM - Define file system for floppy support.
OPTION_FLOPPY_DMA - Enable DMA-based floppy I/O.

7.1.261 OPTION_FLOPPY_144_ONLY Option

The **OPTION_FLOPPY_144_ONLY** option causes code to be generated in the BIOS that disables the floppy disk state machine's testing for 720KB or 1.44MB floppy disks inserted in a 3.5" disk drive. Instead of doing this checking, it assumes that only 1.44MB floppy disks will be inserted.

This option is only necessary when **OPTION_FLOPPY_DMA** is disabled. The purpose of enabling this function is to reduce the chance that errors are encountered. Error checking is extremely time-critical in polled systems, so is only really necessary when DMA support is disabled.

Values:

1 - Enable 1.44-only floppy disk I/O.
0 - Disable 1.44-only floppy disk I/O.

Related Parameters:

FILE_SYSTEM - Define file system for floppy support.
OPTION_FLOPPY_DMA - Enable DMA-based floppy I/O.

7.1.262 OPTION_IDE_RESET Option

The **OPTION_IDE_RESET** option enables or disables code in the BIOS to reset the hard disk controller during POST. The reset function takes time, and may be removed in most targets.

In targets using the IDE code to operate PCMCIA PC Cards with ATA interfaces, this option should be disabled, as it causes a significant timeout during POST.

Values:

1 - Enable reset of hard disk controller in POST.
0 - Disable reset of hard disk controller in POST.

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

7.1.263 OPTION_IDE_SEEK Option

The **OPTION_IDE_SEEK** option enables or disables code in the BIOS to seek the IDE disk drive heads during POST. This POST action is used to determine whether the hard drives are functional or not.

This option only supports the seek feature; it must be enabled in SETUP for the seek to actually occur. In order to permanently enable this feature, the **OPTION_CMOS_HDSEEK** parameter must be set.

Failure to seek properly causes a soft error to occur.

Values:

1 - Enable POST hard drive seek feature.
0 - Disable POST hard drive seek feature.

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

OPTION_CMOS_HDSEEK - Factory default value for hard drive seek option.

7.1.264 OPTION_IDE_DISABLE_INTS Option

The **OPTION_IDE_DISABLE_INTS** option enables or disables code in the BIOS to disable interrupts around the REP INSW or REP OUTSW instructions that perform the actual data transfer.

Most desktop PC BIOS implementations do disable interrupts during I/O; however, this increases interrupt latency, which degrades real-time systems' performance.

Not all IDE controllers can operate without disabling of interrupts during data transfers in all situations; start by enabling this feature, and disabling it later to improve performance if needed.

CAUTION: Allowing interrupts around the data transfer causes some CPUs to create longer bus cycles for the data transfer, which can lead to erroneous transfers. We recommend that you begin with this option enabled, and disable the option only when required, and when proven safe for a given target and drive combination.

Values:

1 - Disable interrupts during hard disk I/O.
0 - Enable interrupts during hard disk I/O.

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

7.1.265 OPTION_IDE_SLOWDOWN Option

The **OPTION_IDE_SLOWDOWN** option enables or disables code in the BIOS to perform replacements for the standard REP INSW or REP OUTSW instructions that perform the actual data transfer. The replacements perform programmed loops that issue INSW and OUTSW instructions, one at a time.

This feature should be disabled unless a hard disk is found to not work with the IDE BIOS. There are no known hard drives that fail with the IDE code, so this should never be necessary.

Values:

1 - Enable slowdown code in IDE data transfers.
0 - Disable slowdown code in IDE data transfers.

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

7.1.266 OPTION_IDE_POLLED Option

The **OPTION_IDE_POLLED** option selects the type of I/O completion to be used by the IDE driver. If this option is enabled, then the IDE controller's status is polled until status bits indicate that a pending operation has been completed. If this option is disabled, then interrupts are used to complete the transfer.

Values:

1 - Enable polling to detect I/O completion (disables interrupts).
0 - Disable polling to detect I/O completion (enables interrupts).

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

7.1.267 OPTION_IDE_AUTODETECT Option

The **OPTION_IDE_AUTODETECT** option enables or disables code in the BIOS to automatically detect the geometry of attached IDE drives during POST. When enabled, this

option eliminates the need for the user to specify the number of heads, cylinders, and sectors per track for drives that support the Extended IDE Protocols.

Not all drives support this feature, or if they do, support it correctly. Older drives, usually under 120MB in size, may have troubles with this protocol. Newer drives above this size are all supporting the Extended IDE Specification.

This feature must be enabled for the LBA or CHS translation mechanisms to be supported, since those methods offer additional translation on top of IDE autodetection.

Values:

1 - Enable autodetect code in SETUP and POST.
0 - Disable autodetect code in SETUP and POST.

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

7.1.268 OPTION_IDE_LBA Option

The **OPTION_IDE_LBA** option enables or disables code in the BIOS to translate the physical geometry of a drive as determined by the geometry autodetect code into logical geometry that can accommodate support for drives larger than 528MB.

Large drives normally support more than 1024 cylinders, but the INT 13h BIOS interface used by DOS and other applications to perform disk I/O does not allow cylinder numbers beyond 1023 to be specified. To solve this problem, the LBA method packs the bits differently according to an industry-standard formula, so that the heads and tracks fields are used to accommodate the extra space provided by the drive. LBA is one method, and CHS is another method, that can be used to address this additional drive space. EMBEDDED BIOS provides both methods so that it can be used to interoperate with all drives formatted in other systems.

Not all drives support this feature, or if they do, support it correctly. Older drives, usually under 120MB in size, may have troubles with this protocol. Newer drives above this size are all supporting the Extended IDE Specification.

This feature requires **OPTION_IDE_AUTODETECT** to be enabled in order to be useful.

Values:

1 - Enable LBA code in SETUP and POST.
0 - Disable LBA code in SETUP and POST.

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

OPTION_IDE_CHS - Enable Phoenix-compatible CHS translation support.

7.1.269 OPTION_IDE_CHS Option

The **OPTION_IDE_CHS** option enables or disables code in the BIOS to translate the physical geometry of a drive as determined by the geometry autodetect code into logical geometry that can accommodate support for drives larger than 528MB.

Large drives normally support more than 1024 cylinders, but the INT 13h BIOS interface used by DOS and other applications to perform disk I/O does not allow cylinder numbers beyond 1023 to be specified. To solve this problem, the CHS method packs the bits differently according to an industry-standard formula, so that the heads and tracks fields are used to accommodate the extra space provided by the drive. CHS is one method compatible with some Phoenix BIOSes, and LBA is another method, that can be used to address this additional drive space. EMBEDDED BIOS provides both methods so that it can be used to interoperate with all drives formatted in other systems.

Not all drives support this feature, or if they do, support it correctly. Older drives, usually under 120MB in size, may have troubles with this protocol. Newer drives above this size are all supporting the Extended IDE Specification.

This feature requires **OPTION_IDE_AUTODETECT** to be enabled in order to be useful.

Values:

1 - Enable CHS code in SETUP and POST.
0 - Disable CHS code in SETUP and POST.

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

OPTION_IDE_LBA - Enable LBA translation support.

7.1.270 OPTION_IDE_LBACMD Option

The **OPTION_IDE_LBACMD** option enables or disables code in the IDE disk driver to use a packetized interface (ATAPI) to communicate with IDE drives. If this option is disabled, then the original PC/AT MFM register file is used for communicating commands, parameters, and data to the target drives.

Values:

1 - Enable LBA packet protocol in IDE driver.
0 - Disable LBA packet protocol in IDE driver.

Related Parameters:

FILE_SYSTEM - Define file system for IDE support.

OPTION_IDE_LBA - Enable LBA autodetect support.

7.1.271 OPTION_IDE_BYTE_IO Option

The **OPTION_IDE_BYTE_IO** option enables or disables code in the IDE driver in the BIOS to only perform 8-bit I/O to the IDE/ATA drive instead of 16-bit I/O. This is necessary for support of ATA cards on platforms that must operate in 8-bit mode.

Values:

1 - Enable 8-bit IDE/ATA I/O.
0 – Disable 8-bit IDE/ATA I/O.

Related Parameters:

FILE_SYSTEM – Declare IDE file system.
OPTION_IDE_AUTODETECT – Autodetect IDE drives.
OPTION_IDE_CHS – Autodetect IDE drives with Phoenix CHS geometry.
OPTION_IDE_DISABLE_INTS – Disable interrupts during data transfers.
OPTION_IDE_LBA – Autodetect IDE drives using LBA geometry.
OPTION_IDE_POLLED – Disable use of interrupts for IDE I/O.
OPTION_IDE_RESET – Issue reset command to drives during POST.
OPTION_IDE_SEEK – Issue seek command to drives during POST.
OPTION_IDE_SLOWDOWN – Use slower I/O loops instead of string I/O.
OPTION_IDE_QUICK_DETECT – Skip IDE detection if drive not ready.

7.1.272 OPTION_IDE_QUICK_DETECT Option

The **OPTION_IDE_QUICK_DETECT** option enables or disables code in the IDE driver in the BIOS to only perform drive autodetection if the drive is ready. In this mode, when a drive reports “not ready” status during POST, it will be skipped (disabled), allowing a faster boot time.

Values:

1 - Enable quick detection of IDE drives.
0 – Disable quick detection of IDE drives.

Related Parameters:

FILE_SYSTEM – Declare IDE file system.
OPTION_IDE_AUTODETECT – Autodetect IDE drives.
OPTION_IDE_CHS – Autodetect IDE drives with Phoenix CHS geometry.
OPTION_IDE_DISABLE_INTS – Disable interrupts during data transfers.
OPTION_IDE_LBA – Autodetect IDE drives using LBA geometry.
OPTION_IDE_POLLED – Disable use of interrupts for IDE I/O.
OPTION_IDE_RESET – Issue reset command to drives during POST.
OPTION_IDE_SEEK – Issue seek command to drives during POST.
OPTION_IDE_SLOWDOWN – Use slower I/O loops instead of string I/O.
OPTION_IDE_BYTE_IO – Perform only 8-bit I/O to drives.

7.1.273 OPTION_BOOT_BEEP Option

The **OPTION_BOOT_BEEP** option enables or disables code in the BIOS to beep the speaker when POST has completed and it is ready to boot the operating system.

This option requires the **OPTION_SUPPORT_SOUND** and the **OPTION_SUPPORT_PORT_B** options to be enabled in order to make noise.

Values:

1 - Enable beep upon POST completion.
0 - Disable beep upon POST completion.

Related Parameters:

OPTION_SUPPORT_SOUND - Enable speaker support.
OPTION_SUPPORT_PORT_B - Enable speaker control hardware.

7.1.274 OPTION_ BOOT_QUICK Option

The **OPTION_BOOT_QUICK** option enables or disables code in the BIOS to disable all messages, pauses, and prompts issued for the user's sake when the system boots, causing the system to boot much faster than a desktop PC system.

Destructive memory tests are also disabled with this option; memory is tested on 1KB boundaries with a nondestructive algorithm.

Values:

1 - Enable quick boot POST.
0 - Disable quick boot POST.

Related Parameters:

None.

7.1.275 OPTION_ BOOT_PRESERVE_WARM Option

The **OPTION_BOOT_PRESERVE_WARM** option enables or disables code in the BIOS to change the warm boot indicator (1234h) in segment 40h after a warm boot to a done status.

This option is normally disabled. It may be enabled to cause a warm boot to not reset its status in the BIOS data area for some applications.

Values:

1 - Don't change indicator from WARM_BOOT to WARM_DONE.
0 - Change indicator from WARM_BOOT to WARM_DONE.

Related Parameters:

None.

7.1.276 OPTION_ BOOT_WARM_DELAY Option

The **OPTION_BOOT_WARM_DELAY** option enables or disables code in the BIOS to delay for approximately one second on a warm boot so that the user has a chance to press the <DEL> or ^C keys so that SETUP can be entered. On some systems, a warm boot is processed so quickly that this delay is necessary for the user to get a chance to enter the keystroke.

Values:

1 - Enable the delay on warm boots.
0 - Disable the delay on warm boots.

Related Parameters:

None.

7.1.277 OPTION_ CON_REDIR_WAIT Option

The **OPTION_CON_REDIR_WAIT** option enables or disables code in the BIOS to wait for the UART's Transmit Buffer Empty (TBE) status bit in the Line Status Register (LSR) to go high before sending the next character via INT 14h.

INT 14h has a built-in test for the case where the characters cannot be transmitted, say, due to a cable being disconnected. However, the INT 14h service times out, which may lead to dropped characters if a cable is disconnected for an extended length of time. Enabling this option causes the console redirection code to keep trying the service until the character is sent.

Values:

1 - Enable the wait for TBE on redirected console output.
0 - Disable the wait for TBE on redirected console output.

Related Parameters:

OPTION_SUPPORT_CON_REDIRECTOR - Enable console redirection.
OPTION_CON_REDIR_DISABLE - Disable redirection if timeout occurs.

7.1.278 OPTION_ CON_REDIR_DISABLE Option

The **OPTION_CON_REDIR_DISABLE** option enables or disables code in the BIOS stop redirecting console I/O over a serial port if output directed to the port times out. When the console redirection is reset, output is delivered to the main video controller and input is read from the keyboard driver.

Values:

1 - Enable reset of console redirection if a timeout occurs.

0 - Disable reset of console redirection if a timeout occurs.

Related Parameters:

OPTION_SUPPORT_CON_REDIRECTOR - Enable console redirection.
OPTION_CON_REDIR_WAIT - Wait for characters to be output.

7.1.279 OPTION_ CON_REDIR_CANCEL Option

The **OPTION_CON_REDIR_CANCEL** option enables or disables code in the BIOS that cancels console redirection if a key is pressed on the main keyboard.

This option is most useful in the lab where the default keyboard and screen are redirected to a serial terminal program running on a host, but when the keyboard is used on the target after some period of debugging, console I/O continues to run on the target's main keyboard and screen.

Values:

1 - Enable autoredirect cancel feature.
0 – Disable autoredirect cancel feature.

Related Parameters:

OPTION_SUPPORT_CON_REDIRECTOR – Enable console redirection feature.

OPTION_CON_REDIR_WAIT – Wait for TBE before outputting characters.

OPTION_CON_REDIR_DISABLE - Disable redirection if timeout expires.

OPTION_CON_REDIR_AUTO – Cancel redirection if video controller detected.

7.1.280 OPTION_ CON_REDIR_AUTO Option

The **OPTION_CON_REDIR_AUTO** option enables or disables code in the BIOS that cancels console redirection if a video BIOS (add-on VGA card or option ROM) is detected in the system.

When this option is enabled, it is most typical to enable **OPTION_SUPPORT_CON_REDIRECTOR**, and then set **CONFIG_CON_REDIR_STD**, **CONFIG_CON_REDIR_DEBUG**, and **CONFIG_CON_REDIR_SETUP** to the COM port number that would be used if a video card were not detected in the system. Then, redirection will be used as specified in the project file unless a video BIOS is detected in the system.

Values:

1 - Enable video card detection feature.
0 – Disable video card detection feature.

Related Parameters:

OPTION_SUPPORT_CON_REDIRECTOR – Enable console redirection feature.

OPTION_CON_REDIR_WAIT – Wait for TBE before outputting characters.

OPTION_CON_REDIR_DISABLE - Disable redirection if timeout expires.

OPTION_CON_REDIR_CANCEL – Cancel redirection if main keyboard used.

7.1.281 OPTION_RTC_CMOS Option

The **OPTION_RTC_CMOS** option enables or disables code in the BIOS to use the Dallas Semiconductor Real-Time Clock chip that contains the CMOS RAM for maintaining the system's date and time. This is the standard mechanism used in most ISA systems today.

OPTION_SUPPORT_CMOS must be enabled in order for this option to be used.

The configuration parameter, **CONFIG_DEFAULT_RTC**, defines the initialization value to be loaded into the Dallas part to define its operating mode. If a change is required, then this parameter must be edited.

Values:

1 - Enable standard PC/AT-compatible real time clock.
0 - Disable standard PC/AT-compatible real time clock.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS hardware driver.
CONFIG_DEFAULT_RTC - Default programming of RTC.

7.1.282 OPTION_RTC_72421 Option

The **OPTION_RTC_72421** option enables or disables code in the BIOS to use the 72421 Real-Time Clock chip for maintaining the system's date and time.

This chip is used by the VersaLogic VL-186 family of industrial computer systems.

Values:

1 - Enable 72421 support.
0 - Disable 72421 support.

Related Parameters:

OPTION_SUPPORT_72421 - Enable 72421 hardware driver.

7.2 Parameters Found in CONFIG.INC

This section explains the purpose of the parameters defined in the INC\CONFIG.INC configuration file. You should make sure that all of the parameters in this section are set properly for your target hardware configuration, or the target will not function properly.

Do not edit the INC\CONFIG.INC file directly! Instead, if you need to make changes to the settings of these parameters, copy the lines from INC\CONFIG.INC to your project file, and change the values in the *project file*.

Note that most parameters in INC\CONFIG.INC are tied closely to options selected in INC\OPTIONS.INC. Please carefully compare the two sets of configuration parameters to be sure they accurately describe your hardware architecture.

7.2.1 BIOS_DATE Parameter

The **BIOS_DATE** parameter is edited by the adaptation engineer to provide a build date timestamp on the BIOS. This date is assembled into the binary image of the BIOS.

The date is automatically configured by the BIOS if you do not specify a date; the date that is used is the date determined by the assembler at the time the BIOS is built. If you override the automatic date with this parameter, then any date can be specified. This may be necessary for OEM version control.

Values:

’MM/DD/YY’ - a string containing the year, month, and date.

Related Parameters:

None.

7.2.2 BIOS_NAME Constant

The **BIOS_NAME** constant is displayed by the BIOS during POST to identify the BIOS software as the property of General Software, Inc.

This constant must not be edited by the adaptation engineer or any adaptations that are modified in this way will be deemed unlicensed by General Software. Do not attempt to translate this string into a foreign language.

Values:

’Copyright (C) 1990-2000 General Software, Inc.’

Related Parameters:

BIOS_RESERVED - All rights reserved message, required for distribution outside of the United States of America.

7.2.3 BIOS_RESERVED Constant

The **BIOS_NAME** constant is displayed by the BIOS during POST. It must not be edited by the adaptation engineer or any adaptations that are modified in this way will be deemed unlicensed by General Software. Do not attempt to translate this string into a foreign language.

Values:

’, All Rights reserved.’

Related Parameters:

BIOS_NAME - EMBEDDED BIOS copyright message.

7.2.4 CPU_TYPE Parameter

The **CPU_TYPE** parameter is used during assembly of the BIOS to determine the minimum CPU level that will be executing the BIOS at run-time. This allows the BIOS to conditionally use more advanced techniques that use 286, 386, 486, Pentium, or Pentium II-specific features to save space and time.

Values:

CPU_86 - CPU core is 8088 or 8086-compatible.
CPU_186 - CPU core is 80186 or 80188-compatible.
CPU_286 - CPU core is 80286 or 80288-compatible.
CPU_386 - CPU core is 80386 -compatible.
CPU_486 - CPU core is 80486-compatible.
CPU_586 - CPU core is Pentium-compatible.
CPU_686 - CPU core is Pentium II, Pentium III, Pentium Pro, or K6-compatible.

7.2.5 CPU_MHZ Parameter

The **CPU_MHZ** parameter is used by some CPU Personality Modules to program baud rates for internal UARTs that have their clocks tied to the CPU clock rate.

Intended for use with 80C186-EC and similar systems, this parameter provides a way to customize basically common code with a minimum of modification to the core BIOS.

Values:

16 - 16 Mhz CPU clock.
20 - 20 Mhz CPU clock.
25 - 25 Mhz CPU clock.
33 - 33 Mhz CPU clock.
50 - 50 Mhz CPU clock.
66 - 66 Mhz CPU clock.
133 - 133 Mhz CPU clock.
166 - 166 Mhz CPU clock.
200 - 200 Mhz CPU clock.
233 - 233 Mhz CPU clock.

400 - 400 Mhz CPU clock.
450 - 450 Mhz CPU clock.
500 - 500 Mhz CPU clock.
1000 - 1 Ghz CPU clock.
n - other CPU clocks.

Related Parameters:

None.

7.2.6 CONFIG_BOARD_VERSION Parameter

The **CONFIG_BOARD_VERSION** parameter is an unarchitected parameter that may be used by the OEM's Board Personality Module (BPM) to indicate which revision of the hardware is being supported by the BIOS.

This allows an OEM to code a BPM which supports several similar hardware platforms, each of which may be slightly different and require subtle changes in chipset initialization, for example. With conditional assembly using this parameter, the BPM can determine which values to use for chipset or other initialization.

Values:

n - Any value, to be passed to Board Personality Module, architected by OEM.

Related Parameters:

None.

7.2.7 CONFIG_POWER_ON_DELAY Parameter

The **CONFIG_POWER_ON_DELAY** parameter specifies a delay executed during POST that compensates for the period of time on power-on when external peripheral components are not yet ready to operate.

Usually, the components on a target initialize at different times, even though these times are very close together. For example, while the CPU may start running if it is a low-power device, the 8042 keyboard controller may still be in an indeterminant state. The delay that this parameter introduces causes the CPU to wait for a specified period of time to give the peripherals a chance to initialize themselves.

The delay is actually specified as a number without specific units such as seconds. Because no timing is available during this early stage (peripherals are assumed to not work yet), the delay is specified in "CPU loops."

If you find that your target sometimes boots, and sometimes does not, then it may be a power-on delay problem. If you are able to reset the target without dropping power, and the problem persists, then it is not a power-on delay problem.

However, if you find that substituting a more heavy-duty power supply for a lighter-duty one causes the target to start working reliably, then you need to increase this parameter, or get a bigger power supply.

If the problem is not related to the power supply, then it can be related to 8042 initialization. See the discussion on **OPTION_SUPPORT_8042** for further details.

Values:

n - A number of iterations (0=none, 1 is the minimum delay, 65,535 is the maximum).

Related Parameters:

OPTION_SUPPORT_POWERON_DELAY - Enable power-on delay feature.

OPTION_SUPPORT_8042 - Discussion about 8042 initialization requirements of 8042 keyboard controllers. If this initialization is incorrect, the target may appear to have a power-on delay problem, when in fact the problem is not the power supply.

7.2.8 CONFIG_CPU_DATA_BYTES Parameter

The **CONFIG_CPU_DATA_BYTES** parameter specifies the number of bytes to reserve in the Extended BIOS Data Area in a field called CpuData for the CPU Personality Module's exclusive use (i.e., to maintain its internal state, as might be needed for shadowing, cache control, etc.)

Values:

n - Number of bytes to reserve in EBDA.

Related Parameters:

CONFIG_CS_DATA_BYTES - Space for Chipset Personality Module.
CONFIG_BOARD_DATA_BYTES - Space for Board Personality Module.

7.2.9 CONFIG_CS_DATA_BYTES Parameter

The **CONFIG_CS_DATA_BYTES** parameter specifies the number of bytes to reserve in the Extended BIOS Data Area in a field called CsData for the Chipset Personality Module's exclusive use (i.e., to maintain its internal state, as might be needed for shadowing, cache control, etc.)

Values:

n - Number of bytes to reserve in EBDA.

Related Parameters:

CONFIG_CPU_DATA_BYTES - Space for CPU Personality Module.
CONFIG_BOARD_DATA_BYTES - Space for Board Personality Module.

7.2.10 CONFIG_BOARD_DATA_BYTES Parameter

The **CONFIG_BOARD_DATA_BYTES** parameter specifies the number of bytes to reserve in the Extended BIOS Data Area in a field called BoardData for the Board Personality Module's exclusive use (i.e., to maintain its internal state, as might be needed for shadowing, cache control, etc.)

Values:

n - Number of bytes to reserve in EBDA.

Related Parameters:

CONFIG_CPU_DATA_BYTES - Space for CPU Personality Module.
CONFIG_CS_DATA_BYTES - Space for Chipset Personality Module.

7.2.11 CONFIG_MAX_CMOS_LOCATIONS Parameter

The **CONFIG_MAX_CMOS_LOCATIONS** parameter specifies the number of CMOS locations available to the BIOS.

Normally, on a standard IBM PC/AT machine, there are 50 cells in the CMOS RAM. Many chipsets extend this limitation when they implement the CMOS RAM feature. This parameter tells the BIOS to what extent CMOS is implemented.

Consult your chipset documentation for complete details, if the CMOS RAM is implemented by the chipset.

Values:

n - A number of cells (50 was standard, 80h is now more common).

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

CONFIG_START_BOARD_CMOS - 1st CMOS cell available to Board Personality Module for its Custom Setup Screen's information.

CONFIG_START_CMOS_CACHE - 1st CMOS cell not used by Real-Time clock's date and time information.

7.2.12 CONFIG_START_BOARD_CMOS Parameter

The **CONFIG_START_BOARD_CMOS** parameter specifies the first CMOS location that can be used by the Board Personality Module to store its own proprietary configuration data.

This provides an architected means by which chipset modules can be compatible with different CMOS RAM sizes.

Consult your chipset documentation for complete details about how many cells you will need to store configuration data, and how many CMOS locations are implemented by the actual hardware.

The use of CMOS by the board module is unarchitected, except for the definition of the first cell's index. Thus, the OEM can use these fields in any way necessary.

Values:

n - A cell number (normally, use the symbol, **CMOS_END_STD**, as a value).

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

CONFIG_MAX_CMOS_LOCATIONS - Total number of CMOS cells supported by CMOS RAM part.

CONFIG_START_CMOS_CACHE - 1st CMOS cell not used by Real-Time clock's date and time.

7.2.13 CONFIG_START_CMOS_CACHE Parameter

The **CONFIG_START_CMOS_CACHE** parameter specifies the first CMOS location that does not contain Real-Time Clock information, such as the date and time.

Technically, this parameter specifies the first cell's index that can be cached by EMBEDDED BIOS into a RAM buffer in the Extended BIOS Data Area for purposes of manipulating a local copy of CMOS without disrupting the stored copy during SETUP. Since the Real-Time clock information is constantly updated, it cannot be cached.

Consult your chipset hardware documentation for complete details about how many cells are supported by your chipset. This cell number almost always starts at 10h, since the cells before cell 10h are usually used by the Real-Time Clock and are not saved by SETUP in the same way that other cells are.

Values:

n - A cell number (10h is common).

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

CONFIG_START_BOARD_CMOS - 1st CMOS cell available to Board Personality Module.

CONFIG_MAX_CMOS_LOCATIONS - Total number of CMOS cells supported by CMOS RAM part.

7.2.14 CONFIG_CMOS_INDEX Parameter

The **CONFIG_CMOS_INDEX** parameter specifies the I/O port assigned to the index register on the CMOS RAM.

In most systems, the I/O port address is 70h. However, in some systems based on processors such as the NEC V51, the I/O port changes to other values (156h for the V51).

Consult your hardware documentation to be sure you have the correct I/O port established for the index register.

If you change the index register, you will most likely need to also change the data register, by adjusting **CONFIG_CMOS_DATA**.

Values:

n - An I/O port number from 000h to fffh. For ISA systems, this is almost always 70h.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

CONFIG_CMOS_DATA - Data register I/O address.

7.2.15 CONFIG_CMOS_DATA Parameter

The **CONFIG_CMOS_DATA** parameter specifies the I/O port assigned to the data register on the CMOS RAM.

In most systems, the I/O port address is 71h. However, in some systems based on processors such as the NEC V51, the I/O port changes to other values (157h for the V51).

Consult your hardware documentation to be sure you have the correct I/O port established for the data register.

If you change the data register, you will most likely need to also change the index register, by adjusting **CONFIG_CMOS_INDEX**.

Values:

n - An I/O port number from 000h to fffh. For ISA systems, this is almost always 71h.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

CONFIG_CMOS_INDEX - Index register I/O address.

7.2.16 CONFIG_DEFAULT_RTC Parameter

The **CONFIG_DEFAULT_RTC** parameter specifies the base rate at which the Real Time Clock is configured to operate.

For ISA systems, this value should be set at 26h and not modified. If you are using a different part other than the standard Dallas one, or if you find that the Real-Time clock is not keeping accurate time, then you may need to adjust this value. See the documentation for the Real-Time Clock you are using to determine how to change this parameter.

Values:

26h - Base rate.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_RTC_CMOS - Enable Dallas RTC support.

7.2.17 CONFIG_CMOS_BOOT_0 Parameter

The **CONFIG_CMOS_BOOT_0** parameter specifies the factory default value for the CMOS first boot action to be performed by POST. Boot actions include booting an operating system or Windows CE from drives A: through K:, booting Windows CE out of ROM, booting DOS from ROM, entering Manufacturing Mode, and entering the debugger.

There are six boot actions, and POST executes them one at a time, until no more actions are possible, at which time it displays a short menu that allows the user to reboot the system, or enter the Setup system. Any combination of actions may be specified by the user, making the system flexible enough to attempt booting a desktop operating system such as Windows NT before booting the backup boot operating system, Embedded DOS-ROM, or entering Manufacturing Mode if no operating system has been programmed into the Flash yet.

Values:

BOOT_NONE - No action for this boot step.
BOOT_DRIVEA - Attempt to boot from logical drive A:.
BOOT_DRIVEB - Attempt to boot from logical drive B:.
BOOT_DRIVEC - Attempt to boot from logical drive C:.
BOOT_DRIVED - Attempt to boot from logical drive D:.
BOOT_DRIVEE - Attempt to boot from logical drive E:.
BOOT_DRIVEF - Attempt to boot from logical drive F:.
BOOT_DRIVEG - Attempt to boot from logical drive G:.
BOOT_DRIVEH - Attempt to boot from logical drive H:.
BOOT_DRIVEI - Attempt to boot from logical drive I:.
BOOT_DRIVEJ - Attempt to boot from logical drive J:.
BOOT_DRIVEK - Attempt to boot from logical drive K:.
BOOT_EDOSROM - Attempt to boot DOS out of ROM.

BOOT_WINCE - Attempt to boot Windows CE out of ROM.
BOOT_MFGMODE - Enter Manufacturing Mode.
BOOT_DEBUGGER - Enter debugger.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.18 CONFIG_CMOS_BOOT_1 Parameter

The **CONFIG_CMOS_BOOT_1** parameter specifies the factory default value for the CMOS second boot action to be performed by POST. Boot actions include booting an operating system or Windows CE from drives A: through K:, booting Windows CE out of ROM, booting DOS from ROM, entering Manufacturing Mode, and entering the debugger.

There are six boot actions, and POST executes them one at a time, until no more actions are possible, at which time it displays a short menu that allows the user to reboot the system, or enter the Setup system. Any combination of actions may be specified by the user, making the system flexible enough to attempt booting a desktop operating system such as Windows NT before booting the backup boot operating system, Embedded DOS-ROM, or entering Manufacturing Mode if no operating system has been programmed into the Flash yet.

Values:

BOOT_NONE - No action for this boot step.
BOOT_DRIVEA - Attempt to boot from drive A:.
BOOT_DRIVEB - Attempt to boot from logical drive B:.
BOOT_DRIVEC - Attempt to boot from logical drive C:.
BOOT_DRIVED - Attempt to boot from logical drive D:.
BOOT_DRIVEE - Attempt to boot from logical drive E:.
BOOT_DRIVEF - Attempt to boot from logical drive F:.
BOOT_DRIVEG - Attempt to boot from logical drive G:.
BOOT_DRIVEH - Attempt to boot from logical drive H:.
BOOT_DRIVEI - Attempt to boot from logical drive I:.
BOOT_DRIVEJ - Attempt to boot from logical drive J:.
BOOT_DRIVEK - Attempt to boot from logical drive K:.
BOOT_EDOSROM - Attempt to boot DOS out of ROM.
BOOT_WINCE - Attempt to boot Windows CE out of ROM.
BOOT_MFGMODE - Enter Manufacturing Mode.
BOOT_DEBUGGER - Enter debugger.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.19 CONFIG_CMOS_BOOT_2 Parameter

The **CONFIG_CMOS_BOOT_2** parameter specifies the factory default value for the CMOS third boot action to be performed by POST. Boot actions include booting an operating system or Windows CE from drives A: through K:, booting Windows CE out of ROM, booting DOS from ROM, entering Manufacturing Mode, and entering the debugger.

There are six boot actions, and POST executes them one at a time, until no more actions are possible, at which time it displays a short menu that allows the user to reboot the system, or enter the Setup system. Any combination of actions may be specified by the user, making the system flexible enough to attempt booting a desktop operating system such as Windows NT before booting the backup boot operating system, Embedded DOS-ROM, or entering Manufacturing Mode if no operating system has been programmed into the Flash yet.

Values:

BOOT_NONE - No action for this boot step.
BOOT_DRIVEA - Attempt to boot from drive A:.
BOOT_DRIVEB - Attempt to boot from logical drive B:.
BOOT_DRIVEC - Attempt to boot from logical drive C:.
BOOT_DRIVED - Attempt to boot from logical drive D:.
BOOT_DRIVEE - Attempt to boot from logical drive E:.
BOOT_DRIVEF - Attempt to boot from logical drive F:.
BOOT_DRIVEG - Attempt to boot from logical drive G:.
BOOT_DRIVEH - Attempt to boot from logical drive H:.
BOOT_DRIVEI - Attempt to boot from logical drive I:.
BOOT_DRIVEJ - Attempt to boot from logical drive J:.
BOOT_DRIVEK - Attempt to boot from logical drive K:.
BOOT_WINCE - Attempt to boot Windows CE out of ROM.
BOOT_EDOSROM - Attempt to boot DOS out of ROM.
BOOT_MFGMODE - Enter Manufacturing Mode.
BOOT_DEBUGGER - Enter debugger.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.20 CONFIG_CMOS_BOOT_3 Parameter

The **CONFIG_CMOS_BOOT_3** parameter specifies the factory default value for the CMOS fourth boot action to be performed by POST. Boot actions include booting an operating system or Windows CE from drives A: through K:, booting Windows CE out of ROM, booting DOS from ROM, entering Manufacturing Mode, and entering the debugger.

There are six boot actions, and POST executes them one at a time, until no more actions are possible, at which time it displays a short menu that allows the user to reboot the system, or enter the Setup system. Any combination of actions may be specified by the user, making the system flexible enough to attempt booting a desktop operating system such as Windows NT before booting the backup boot operating system, Embedded DOS-ROM, or entering Manufacturing Mode if no operating system has been programmed into the Flash yet.

Values:

BOOT_NONE - No action for this boot step.
BOOT_DRIVEA - Attempt to boot from drive A:.
BOOT_DRIVEB - Attempt to boot from logical drive B:.
BOOT_DRIVEC - Attempt to boot from logical drive C:.
BOOT_DRIVED - Attempt to boot from logical drive D:.

BOOT_DRIVEE - Attempt to boot from logical drive E:.
BOOT_DRIVEF - Attempt to boot from logical drive F:.
BOOT_DRIVEG - Attempt to boot from logical drive G:.
BOOT_DRIVEH - Attempt to boot from logical drive H:.
BOOT_DRIVEI - Attempt to boot from logical drive I:.
BOOT_DRIVEJ - Attempt to boot from logical drive J:.
BOOT_DRIVEK - Attempt to boot from logical drive K:.
BOOT_WINCE - Attempt to boot Windows CE out of ROM.
BOOT_EDOSROM - Attempt to boot DOS out of ROM.
BOOT_MFGMODE - Enter Manufacturing Mode.
BOOT_DEBUGGER - Enter debugger.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.21 CONFIG_CMOS_BOOT_4 Parameter

The **CONFIG_CMOS_BOOT_4** parameter specifies the factory default value for the CMOS fifth boot action to be performed by POST. Boot actions include booting an operating system or Windows CE from drives A: through K:, booting Windows CE out of ROM, booting DOS from ROM, entering Manufacturing Mode, and entering the debugger.

There are six boot actions, and POST executes them one at a time, until no more actions are possible, at which time it displays a short menu that allows the user to reboot the system, or enter the Setup system. Any combination of actions may be specified by the user, making the system flexible enough to attempt booting a desktop operating system such as Windows NT before booting the backup boot operating system, Embedded DOS-ROM, or entering Manufacturing Mode if no operating system has been programmed into the Flash yet.

Values:

BOOT_NONE - No action for this boot step.
BOOT_DRIVEA - Attempt to boot from drive A:.
BOOT_DRIVEB - Attempt to boot from logical drive B:.
BOOT_DRIVEC - Attempt to boot from logical drive C:.
BOOT_DRIVED - Attempt to boot from logical drive D:.
BOOT_DRIVEE - Attempt to boot from logical drive E:.
BOOT_DRIVEF - Attempt to boot from logical drive F:.
BOOT_DRIVEG - Attempt to boot from logical drive G:.
BOOT_DRIVEH - Attempt to boot from logical drive H:.
BOOT_DRIVEI - Attempt to boot from logical drive I:.
BOOT_DRIVEJ - Attempt to boot from logical drive J:.
BOOT_DRIVEK - Attempt to boot from logical drive K:.
BOOT_WINCE - Attempt to boot Windows CE out of ROM.
BOOT_EDOSROM - Attempt to boot DOS out of ROM.
BOOT_MFGMODE - Enter Manufacturing Mode.
BOOT_DEBUGGER - Enter debugger.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.22 CONFIG_CMOS_BOOT_5 Parameter

The **CONFIG_CMOS_BOOT_5** parameter specifies the factory default value for the CMOS sixth boot action to be performed by POST. Boot actions include booting an operating system or Windows CE from drives A: through K:, booting Windows CE out of ROM, booting DOS from ROM, entering Manufacturing Mode, and entering the debugger.

There are six boot actions, and POST executes them one at a time, until no more actions are possible, at which time it displays a short menu that allows the user to reboot the system, or enter the Setup system. Any combination of actions may be specified by the user, making the system flexible enough to attempt booting a desktop operating system such as Windows NT before booting the backup boot operating system, Embedded DOS-ROM, or entering Manufacturing Mode if no operating system has been programmed into the Flash yet.

<u>Values:</u>

BOOT_NONE - No action for this boot step.
BOOT_DRIVEA - Attempt to boot from drive A:.
BOOT_DRIVEB - Attempt to boot from logical drive B:.
BOOT_DRIVEC - Attempt to boot from logical drive C:.
BOOT_DRIVED - Attempt to boot from logical drive D:.
BOOT_DRIVEE - Attempt to boot from logical drive E:.
BOOT_DRIVEF - Attempt to boot from logical drive F:.
BOOT_DRIVEG - Attempt to boot from logical drive G:.
BOOT_DRIVEH - Attempt to boot from logical drive H:.
BOOT_DRIVEI - Attempt to boot from logical drive I:.
BOOT_DRIVEJ - Attempt to boot from logical drive J:.
BOOT_DRIVEK - Attempt to boot from logical drive K:.
BOOT_WINCE - Attempt to boot Windows CE out of ROM.
BOOT_EDOSROM - Attempt to boot DOS out of ROM.
BOOT_MFGMODE - Enter Manufacturing Mode.
BOOT_DEBUGGER - Enter debugger.

<u>Related Parameters:</u>

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.23 CONFIG_CMOS_FLOPPY_0 Parameter

The **CONFIG_CMOS_FLOPPY_0** parameter specifies the factory-default device assignment for the first physical floppy drive. Note that this parameter has nothing to do with drive emulators or drive letter assignments. This parameter tells the BIOS which type of physical floppy drive will be found as the second one on the floppy drive cable.

<u>*If you are using a floppy disk in your system, you need to adjust this parameter to properly indicate what the factory-default floppy disk configuration will be.*</u>

If your system has no CMOS configuration, then <u>*this is the information the BIOS uses to determine what floppy drive type is used*</u> for the first physical floppy drive.

If no floppy drive is assigned to the drive letter, then the value is 0 or **DRIVE_NONE**. The other values are specified below. Note that these values define *drive types*, not floppy disk types. For example, it is possible to insert either a 720KB or a 1.44MB floppy in a 3.5" 1.44MB drive. It is the drive type that is specified here; the discovery of a particular disk type when a disk is inserted into the drive is the job of the floppy disk driver.

Values:

DRIVE_NONE - No device assigned to drive.
DRIVE_360 - 5.25", 360KB floppy drive.
DRIVE_12 - 5.25", 1.2MB floppy drive.
DRIVE_720 - 3.5", 720KB floppy drive.
DRIVE_144 - 3.5", 1.44MB floppy drive.

Related Parameters:

FILE_SYSTEM - Enable floppy disk support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_FLOPPY_1 - Floppy 1 device assignment.
CONFIG_CMOS_FLOPPY_2 - Floppy 2 device assignment.
CONFIG_CMOS_FLOPPY_3 - Floppy 3 device assignment.

7.2.24 CONFIG_CMOS_FLOPPY_1 Parameter

The **CONFIG_CMOS_FLOPPY_1** parameter specifies the factory-default device assignment for the second physical floppy drive. Note that this parameter has nothing to do with drive emulators or drive letter assignments. This parameter tells the BIOS which type of physical floppy drive will be found as the second one on the floppy drive cable.

If you are using a floppy disk in your system, you need to adjust this parameter to properly indicate what the factory-default floppy disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine what floppy drive type is used* for the second physical floppy drive.

If no floppy drive is assigned to the drive letter, then the value is 0 or **DRIVE_NONE**. The other values are specified below. Note that these values define *drive types*, not floppy disk types. For example, it is possible to insert either a 720KB or a 1.44MB floppy in a 3.5" 1.44MB drive. It is the drive type that is specified here; the discovery of a particular disk type when a disk is inserted into the drive is the job of the floppy disk driver.

Values:

DRIVE_NONE - No device assigned to drive.
DRIVE_360 - 5.25", 360KB floppy drive.
DRIVE_12 - 5.25", 1.2MB floppy drive.
DRIVE_720 - 3.5", 720KB floppy drive.
DRIVE_144 - 3.5", 1.44MB floppy drive.
DRIVE_288 - 3.5", 2.88MB floppy drive

Related Parameters:

FILE_SYSTEM - Enable floppy disk support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_FLOPPY_0 - Floppy 0 device assignment.
CONFIG_CMOS_FLOPPY_2 - Floppy 2 device assignment.
CONFIG_CMOS_FLOPPY_3 - Floppy 3 device assignment.

7.2.25 CONFIG_CMOS_FLOPPY_2 Parameter

The **CONFIG_CMOS_FLOPPY_2** parameter specifies the factory-default device assignment for the third physical floppy drive. Note that this parameter has nothing to do with drive emulators or drive letter assignments. This parameter tells the BIOS which type of physical floppy drive will be found as the third one on the floppy drive cable.

If you are using a floppy disk in your system, you need to adjust this parameter to properly indicate what the factory-default floppy disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine what floppy drive type is used* for the third physical floppy drive.

If no floppy drive is assigned to the drive letter, then the value is 0 or **DRIVE_NONE**. The other values are specified below. Note that these values define *drive types*, not floppy disk types. For example, it is possible to insert either a 720KB or a 1.44MB floppy in a 3.5" 1.44MB drive. It is the drive type that is specified here; the discovery of a particular disk type when a disk is inserted into the drive is the job of the floppy disk driver.

Values:

DRIVE_NONE - No device assigned to drive.
DRIVE_360 - 5.25", 360KB floppy drive.
DRIVE_12 - 5.25", 1.2MB floppy drive.
DRIVE_720 - 3.5", 720KB floppy drive.
DRIVE_144 - 3.5", 1.44MB floppy drive.
DRIVE_288 - 3.5", 2.88MB floppy drive.

Related Parameters:

FILE_SYSTEM - Enable floppy disk support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_FLOPPY_0 - Floppy 0 device assignment.
CONFIG_CMOS_FLOPPY_1 - Floppy 1 device assignment.
CONFIG_CMOS_FLOPPY_3 - Floppy 3 device assignment.

7.2.26 CONFIG_CMOS_FLOPPY_3 Parameter

The **CONFIG_CMOS_FLOPPY_0** parameter specifies the factory-default device assignment for the fourth physical floppy drive. Note that this parameter has nothing to do with drive

emulators or drive letter assignments. This parameter tells the BIOS which type of physical floppy drive will be found as the fourth one on the floppy drive cable.

If you are using a floppy disk in your system, you need to adjust this parameter to properly indicate what the factory-default floppy disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine what floppy drive type is used* for the fourth physical floppy drive.

If no floppy drive is assigned to the drive letter, then the value is 0 or **DRIVE_NONE**. The other values are specified below. Note that these values define *drive types*, not floppy disk types. For example, it is possible to insert either a 720KB or a 1.44MB floppy in a 3.5" 1.44MB drive. It is the drive type that is specified here; the discovery of a particular disk type when a disk is inserted into the drive is the job of the floppy disk driver.

Values:

DRIVE_NONE - No device assigned to drive.
DRIVE_360 - 5.25", 360KB floppy drive.
DRIVE_12 - 5.25", 1.2MB floppy drive.
DRIVE_720 - 3.5", 720KB floppy drive.
DRIVE_144 - 3.5", 1.44MB floppy drive.
DRIVE_288 - 3.5", 2.88MB floppy drive.

Related Parameters:

FILE_SYSTEM - Enable floppy disk support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_FLOPPY_0 - Floppy 0 device assignment.
CONFIG_CMOS_FLOPPY_1 - Floppy 1 device assignment.
CONFIG_CMOS_FLOPPY_2 - Floppy 2 device assignment.

7.2.27 CONFIG_CMOS_IDE_0 Parameter

The **CONFIG_CMOS_IDE_0** parameter specifies the factory-default value to be used as the first hard drive's drive type.

If you are using a hard drive in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine what hard disk type is used* for the physical hard drive.

Type **IDE_USER** is user-defined. If you use type **IDE_USER**, then the geometry will be read from **CONFIG_CMOS_IDE0_CYL**, **CONFIG_CMOS_IDE0_HEADS**, and **CONFIG_CMOS_IDE0_SPT** to be used for the number of cylinders, heads, and sectors per track, respectively.

The following example shows how to define the the first hard drive as being type IDE_USER (the other three geometry parameters are also specified here to support an ATA card with 320 cylinders, 2 heads, and 32 sectors per track):

```
CONFIG_CMOS_IDE_0         =    IDE_USER     ; the drive type.
CONFIG_CMOS_IDE0_CYL      =    320          ; cylinders.
CONFIG_CMOS_IDE0_HEADS    =    2            ; heads.
CONFIG_CMOS_IDE0_SPT      =    32           ; sectors per track.
```

Type **IDE_AUTO** is autodetect, without any geometry translation. This allows EMBEDDED BIOS to determine during POST the actual geometry (heads, tracks, and sectors per track) so that it becomes unnecessary for the user to key-in the actual geometry with type **IDE_USER**. Some older drives do not support the industry-standard IDE protocol for determining the geometry, so this may not work in some older systems. Also, if a drive has been used in a system with user-specified geometry that does not match the drive-reported geometry, type **IDE_AUTO** should not be used, because it cannot know the geometry used on the other system.

Type **IDE_LBA** is another autodetect type, and it also adds LBA (Logical Block Addressing) translation, supporting drives larger than 528MB. This has become the industry standard, and General Software recommends using type **IDE_LBA** for embedded use.

Type **IDE_PHOENIX** is another autodetect type, and adds CHS (Cylinder/Head/Sector) translation, a proprietary scheme introduced by Phoenix Technologies. Use of this type is discouraged since it is only provided for compatibility with drives already formatted CHS.

Values:

IDE_NONE (0) - Specifies drive not installed.

IDE_AUTO (1) - Specifies drive type is detected automatically during POST through extended IDE protocol. This is not supported by all IDE drives because some drives don't have this feature, and others may implement it incorrectly.

IDE_LBA (2) - Specifies drive type is detected automatically during POST through extended IDE protocol, and that LBA translation will be performed to support drives with more than 1024 cylinders. The recommended standard for all drives larger than 528MB.

IDE_PHOENIX (3) - Specifies drive type is detected automatically during POST through extended IDE protocol, and that Phoenix-compatible CHS translation will be performed to support drives with more than 1024 cylinders.

Related Parameters:

FILE_SYSTEM - Enable file systems.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_IDE_AUTODETECT - Enable IDE autodetect (type 48) support.
OPTION_IDE_LBA - Enable IDE Logical Block Addressing support.
OPTION_IDE_CHS - Enable IDE Cylinder/Head/Sector translation support.
CONFIG_CMOS_IDE_1 - Configure second IDE drive type.
CONFIG_CMOS_IDE_2 - Configure third IDE drive type.
CONFIG_CMOS_IDE_3 - Configure fourth IDE drive type.

7.2.28 CONFIG_CMOS_IDE_1 Parameter

The **CONFIG_CMOS_IDE_1** parameter specifies the factory-default value to be used as the second hard drive's drive type. See the section on **CONFIG_CMOS_IDE_0** for details.

Values:

IDE_NONE (0) - Specifies drive not installed.

IDE_AUTO (1) - Specifies drive type is detected automatically during POST through extended IDE protocol. This is not supported by all IDE drives because some drives don't have this feature, and others may implement it incorrectly.

IDE_LBA (2) - Specifies drive type is detected automatically during POST through extended IDE protocol, and that LBA translation will be performed to support drives with more than 1024 cylinders. The recommended standard for all drives larger than 528MB.

IDE_PHOENIX (3) - Specifies drive type is detected automatically during POST through extended IDE protocol, and that Phoenix-compatible CHS translation will be performed to support drives with more than 1024 cylinders.

Related Parameters:

FILE_SYSTEM - Enable file systems.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_IDE_AUTODETECT - Enable IDE autodetect (type 48) support.
OPTION_IDE_LBA - Enable IDE Logical Block Addressing support.
OPTION_IDE_CHS - Enable IDE Cylinder/Head/Sector translation support.
CONFIG_CMOS_IDE_0 - Configure first IDE drive type.
CONFIG_CMOS_IDE_2 - Configure third IDE drive type.
CONFIG_CMOS_IDE_3 - Configure fourth IDE drive type.

7.2.29 CONFIG_CMOS_IDE_2 Parameter

The **CONFIG_CMOS_IDE_2** parameter specifies the factory-default value to be used as the third hard drive's drive type. See the section on **CONFIG_CMOS_IDE_0** for details.

Values:

IDE_NONE (0) - Specifies drive not installed.

IDE_AUTO (1) - Specifies drive type is detected automatically during POST through extended IDE protocol. This is not supported by all IDE drives because some drives don't have this feature, and others may implement it incorrectly.

IDE_LBA (2) - Specifies drive type is detected automatically during POST through extended IDE protocol, and that LBA translation will be performed to support

drives with more than 1024 cylinders. The recommended standard for all drives larger than 528MB.

IDE_PHOENIX (3) - Specifies drive type is detected automatically during POST through extended IDE protocol, and that Phoenix-compatible CHS translation will be performed to support drives with more than 1024 cylinders.

Related Parameters:

FILE_SYSTEM - Enable file systems.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_IDE_AUTODETECT - Enable IDE autodetect (type 48) support.
OPTION_IDE_LBA - Enable IDE Logical Block Addressing support.
OPTION_IDE_CHS - Enable IDE Cylinder/Head/Sector translation support.
CONFIG_CMOS_IDE_0 - Configure first IDE drive type.
CONFIG_CMOS_IDE_1 - Configure second IDE drive type.
CONFIG_CMOS_IDE_3 - Configure fourth IDE drive type.

7.2.30 CONFIG_CMOS_IDE_3 Parameter

The **CONFIG_CMOS_IDE_3** parameter specifies the factory-default value to be used as the fourth hard drive's drive type. See the section on **CONFIG_CMOS_IDE_0** for details.

Values:

IDE_NONE (0) - Specifies drive not installed.

IDE_AUTO (1) - Specifies drive type is detected automatically during POST through extended IDE protocol. This is not supported by all IDE drives because some drives don't have this feature, and others may implement it incorrectly.

IDE_LBA (2) - Specifies drive type is detected automatically during POST through extended IDE protocol, and that LBA translation will be performed to support drives with more than 1024 cylinders. The recommended standard for all drives larger than 528MB.

IDE_PHOENIX (3) - Specifies drive type is detected automatically during POST through extended IDE protocol, and that Phoenix-compatible CHS translation will be performed to support drives with more than 1024 cylinders.

Related Parameters:

FILE_SYSTEM - Enable file systems.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_IDE_AUTODETECT - Enable IDE autodetect (type 48) support.
OPTION_IDE_LBA - Enable IDE Logical Block Addressing support.
OPTION_IDE_CHS - Enable IDE Cylinder/Head/Sector translation support.
CONFIG_CMOS_IDE_0 - Configure first IDE drive type.
CONFIG_CMOS_IDE_1 - Configure second IDE drive type.
CONFIG_CMOS_IDE_2 - Configure third IDE drive type.

7.2.31 CONFIG_CMOS_IDE0_CYL Parameter

The **CONFIG_CMOS_IDE0_CYL** parameter specifies the factory-default value to be used as the first drive's number of cylinders should **CONFIG_CMOS_IDE_0** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of cylinders supported* by the drive.

Values:

n - Specifies number of cylinders (1-4096).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE_0 - Configure first drive type.
CONFIG_CMOS_IDE0_HDS - Number of heads for drive.
CONFIG_CMOS_IDE0_SPT - Number of sectors per track for drive.

7.2.32 CONFIG_CMOS_IDE0_HDS Parameter

The **CONFIG_CMOS_IDE0_HDS** parameter specifies the factory-default value to be used as the first drive's number of heads should **CONFIG_CMOS_IDE_0** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of heads supported* by the drive.

Values:

n - Specifies number of heads (1-63).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE - Configure hard drives.
CONFIG_CMOS_IDE_0 - Configure hard drive type.
CONFIG_CMOS_IDE0_CYL - Number of cylinders for drive.

CONFIG_CMOS_IDE0_SPT - Number of sectors per track for drive.

7.2.33 CONFIG_CMOS_IDE0_SPT Parameter

The **CONFIG_CMOS_IDE0_SPT** parameter specifies the factory-default value to be used as the first drive's number of sectors per track should **CONFIG_CMOS_IDE_0** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of sectors per track supported* by the drive.

Values:

n - Specifies number of sector per track (1-63).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE_0 - Configure hard drive type.
CONFIG_CMOS_IDE0_CYL - Number of cylinders for drive.
CONFIG_CMOS_IDE0_HDS - Number of heads for drive.

7.2.34 CONFIG_CMOS_IDE1_CYL Parameter

The **CONFIG_CMOS_IDE1_CYL** parameter specifies the factory-default value to be used as the second drive's number of cylinders should **CONFIG_CMOS_IDE_1** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of cylinders supported* by the drive.

Values:

n - Specifies number of cylinders (1-4096).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE_1 - Configure first drive type.
CONFIG_CMOS_IDE1_HDS - Number of heads for drive.

CONFIG_CMOS_IDE1_SPT - Number of sectors per track for drive.

7.2.35 CONFIG_CMOS_IDE1_HDS Parameter

The **CONFIG_CMOS_IDE0_HDS** parameter specifies the factory-default value to be used as the second drive's number of heads should **CONFIG_CMOS_IDE_1** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of heads supported* by the drive.

Values:

n - Specifies number of heads (1-63).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE - Configure hard drives.
CONFIG_CMOS_IDE_1 - Configure hard drive type.
CONFIG_CMOS_IDE1_CYL - Number of cylinders for drive.
CONFIG_CMOS_IDE1_SPT - Number of sectors per track for drive.

7.2.36 CONFIG_CMOS_IDE1_SPT Parameter

The **CONFIG_CMOS_IDE1_SPT** parameter specifies the factory-default value to be used as the second drive's number of sectors per track should **CONFIG_CMOS_IDE_1** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of sectors per track supported* by the drive.

Values:

n - Specifies number of sector per track (1-63).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE_1 - Configure hard drive type.

CONFIG_CMOS_IDE1_CYL - Number of cylinders for drive.
CONFIG_CMOS_IDE1_HDS - Number of heads for drive.

7.2.37 CONFIG_CMOS_IDE2_CYL Parameter

The **CONFIG_CMOS_IDE2_CYL** parameter specifies the factory-default value to be used as the third drive's number of cylinders should **CONFIG_CMOS_IDE_2** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of cylinders supported* by the drive.

Values:

n - Specifies number of cylinders (1-4096).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE_2 - Configure first drive type.
CONFIG_CMOS_IDE2_HDS - Number of heads for drive.
CONFIG_CMOS_IDE2_SPT - Number of sectors per track for drive.

7.2.38 CONFIG_CMOS_IDE2_HDS Parameter

The **CONFIG_CMOS_IDE0_HDS** parameter specifies the factory-default value to be used as the third drive's number of heads should **CONFIG_CMOS_IDE_2** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of heads supported* by the drive.

Values:

n - Specifies number of heads (1-63).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE - Configure hard drives.

CONFIG_CMOS_IDE_2 - Configure hard drive type.
CONFIG_CMOS_IDE2_CYL - Number of cylinders for drive.
CONFIG_CMOS_IDE2_SPT - Number of sectors per track for drive.

7.2.39 CONFIG_CMOS_IDE2_SPT Parameter

The **CONFIG_CMOS_IDE2_SPT** parameter specifies the factory-default value to be used as the third drive's number of sectors per track should **CONFIG_CMOS_IDE_2** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of sectors per track supported* by the drive.

Values:

n - Specifies number of sector per track (1-63).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE_2 - Configure hard drive type.
CONFIG_CMOS_IDE2_CYL - Number of cylinders for drive.
CONFIG_CMOS_IDE2_HDS - Number of heads for drive.

7.2.40 CONFIG_CMOS_IDE3_CYL Parameter

The **CONFIG_CMOS_IDE3_CYL** parameter specifies the factory-default value to be used as the fourth drive's number of cylinders should **CONFIG_CMOS_IDE_3** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of cylinders supported* by the drive.

Values:

n - Specifies number of cylinders (1-4096).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

CONFIG_CMOS_IDE_3 - Configure first drive type.
CONFIG_CMOS_IDE3_HDS - Number of heads for drive.
CONFIG_CMOS_IDE3_SPT - Number of sectors per track for drive.

7.2.41 CONFIG_CMOS_IDE3_HDS Parameter

The **CONFIG_CMOS_IDE3_HEADS** parameter specifies the factory-default value to be used as the fourth drive's number of heads should **CONFIG_CMOS_IDE_3** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of heads supported* by the drive.

Values:

n - Specifies number of heads (1-63).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE - Configure hard drives.
CONFIG_CMOS_IDE_3 - Configure hard drive type.
CONFIG_CMOS_IDE3_CYL - Number of cylinders for drive.
CONFIG_CMOS_IDE3_SPT - Number of sectors per track for drive.

7.2.42 CONFIG_CMOS_IDE3_SPT Parameter

The **CONFIG_CMOS_IDE3_SPT** parameter specifies the factory-default value to be used as the fourth drive's number of sectors per track should **CONFIG_CMOS_IDE_3** contain the value **IDE_USER** (user defined type).

If you are using a user-defined hard drive type in your system, you need to adjust this parameter to properly indicate what the factory-default hard disk configuration will be.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the number of sectors per track supported* by the drive.

Values:

n - Specifies number of sector per track (1-63).

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_ATA - Enable PCMCIA ATA card support.

OPTION_SUPPORT_CMOS - Enable CMOS support.
CONFIG_CMOS_IDE_3 - Configure hard drive type.
CONFIG_CMOS_IDE3_CYL - Number of cylinders for drive.
CONFIG_CMOS_IDE3_HDS - Number of heads for drive.

7.2.43 CONFIG_CMOS_ASSIGN_A Parameter

The **CONFIG_CMOS_ASSIGN_A** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.44 CONFIG_CMOS_ASSIGN_B Parameter

The **CONFIG_CMOS_ASSIGN_B** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.45 CONFIG_CMOS_ASSIGN_C Parameter

The **CONFIG_CMOS_ASSIGN_C** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.46 CONFIG_CMOS_ASSIGN_D Parameter

The **CONFIG_CMOS_ASSIGN_D** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.47 CONFIG_CMOS_ASSIGN_E Parameter

The **CONFIG_CMOS_ASSIGN_E** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.48 CONFIG_CMOS_ASSIGN_F Parameter

The **CONFIG_CMOS_ASSIGN_F** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive

numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.49 CONFIG_CMOS_ASSIGN_G Parameter

The **CONFIG_CMOS_ASSIGN_G** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.50 CONFIG_CMOS_ASSIGN_H Parameter

The **CONFIG_CMOS_ASSIGN_H** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.51 CONFIG_CMOS_ASSIGN_I Parameter

The **CONFIG_CMOS_ASSIGN_I** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.

n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.52 CONFIG_CMOS_ASSIGN_J Parameter

The **CONFIG_CMOS_ASSIGN_J** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.53 CONFIG_CMOS_ASSIGN_K Parameter

The **CONFIG_CMOS_ASSIGN_K** parameter specifies what file system will be mapped to the drive letter. While it is actually DOS that provides drive letter assignments, it gets the drive numbering from the BIOS, and the numbered drives are mapped to physical file systems in the BIOS itself.

The value associated with this parameter is an index into the **FILE_SYSTEM** table created by the OEM in the project file. The value 0 means no assignment (that is, the drive letter will not have any device mapping).

Nonzero values are indexes into the file system table. The file system table's entries are numbered 1, 2, 3, and so on, starting with all of the "soft" entries first, then the "hard" entries, regardless of whether the soft entries appear before the hard entries in the table.

If your system has no CMOS configuration, then *this is the information the BIOS uses to determine the file system assigned to the drive*.

Values:

0 - No device is assigned to this drive letter.
n - An index into the **FILE_SYSTEM** table, from 1 to the maximum number of entries.

Related Parameters:

FILE_SYSTEM - Enable file system support.
OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.54 CONFIG_CMOS_TYPEMATIC_DELAY Parameter

The **CONFIG_CMOS_TYPEMATIC_DELAY** parameter specifies the factory default value to be used as the delay when programming the keyboard for typematic repeat of pressed keys.

The delay parameter specifies how long a key should be pressed before the keyboard begins repeating the character automatically.

This feature requires that **OPTION_SUPPORT_KEYBOARD** and **OPTION_CMOS_TYPEMATIC** both be enabled.

Values:

0 - 250 milliseconds.
1 - 500 milliseconds.
2 - 750 milliseconds.
3 - one second.

Related Parameters:

OPTION_SUPPORT_KEYBOARD - Enable keyboard support.
OPTION_CMOS_TYPEMATIC - Factory default for typematic enable.
CONFIG_CMOS_TYPEMATIC_RATE - Factory default for typematic repeat rate.

7.2.55 CONFIG_CMOS_TYPEMATIC_RATE Parameter

The **CONFIG_CMOS_TYPEMATIC_RATE** parameter specifies the factory default value to be used as the repeat rate when programming the keyboard for typematic repeat of pressed keys.

The rate parameter specifies how fast the keyboard should repeat a character once typematic action commences.

This feature requires that **OPTION_SUPPORT_KEYBOARD** and **OPTION_CMOS_TYPEMATIC** both be enabled.

Values:

0 - 30.0 characters per second.

1 - 26.7 characters per second.
2 - 24.0 characters per second.
3 - 21.8 characters per second.
4 - 20.0 characters per second.
5 - 18.5 characters per second.
6 - 17.1 characters per second.
7 - 16.0 characters per second.
8 - 15.0 characters per second.
9 - 13.3 characters per second.
10 - 12.0 characters per second.
11 - 10.9 characters per second.
12 - 10.0 characters per second.
13 - 9.2 characters per second.
14 - 8.6 characters per second.
15 - 8.0 characters per second.
16 - 7.5 characters per second.
17 - 6.7 characters per second.
18 - 6.0 characters per second.
19 - 5.5 characters per second.
20 - 5.0 characters per second.
21 - 4.6 characters per second.
22 - 4.3 characters per second.
23 - 4.0 characters per second.
24 - 3.7 characters per second.
25 - 3.3 characters per second.
26 - 3.0 characters per second.
27 - 2.7 characters per second.
28 - 2.5 characters per second.
29 - 2.3 characters per second.
30 - 2.1 characters per second.
31 - 2.0 characters per second.

Related Parameters:

OPTION_SUPPORT_KEYBOARD - Enable keyboard support.
OPTION_CMOS_TYPEMATIC - Factory default for typematic enable.
CONFIG_CMOS_TYPEMATIC_DELAY - Factory default for typematic delay.

7.2.56 CONFIG_CMOS_FLOPPY_RETRY Parameter

The **CONFIG_CMOS_FLOPPY_RETRY** parameter specifies the factory default value to be stored in CMOS representing the number of times the floppy disk driver will step through its state table looking for the correct media in a given drive when an operation is performed.

Ordinarily, this value should be at least three (3), since the state tables can involve up to three steps before a correct media type can be determined.

If an embedded system is to be fixed so that it only operates with a specific drive type, then this parameter can be set to 1, and **OPTION_FLOPPY_144_ONLY** can be enabled.

Values:

n - Number of retries before floppy disk I/O returns sector not found error.

Related Parameters:

OPTION_SUPPORT_FLOPPY - Enable floppy disk support.
OPTION_SUPPORT_CMOS - Enable CMOS support.
OPTION_FLOPPY_144_ONLY - Only support 1.44MB floppy disks.

7.2.57 CONFIG_CMOS_EQUIP Parameter

The **CONFIG_CMOS_EQUIP** parameter specifies the factory default value to be used to initialize the equipment byte in the BIOS data area, if not initialized in other ways on a system.

Values:

***xx*h** - Factory default equipment byte as saved in CMOS.

Related Parameters:

OPTION_SUPPORT_CMOS - Enable CMOS support.

7.2.58 CONFIG_BOOT_ATTEMPT Parameter

The **CONFIG_BOOT_ATTEMPT** parameter specifies the number of times that POST will attempt to boot from each of the boot drives selected in the SETUP options before timing out the operation and switching to the next boot action.

Ordinarily, more than one attempt is made to account for floppy drive spin-up on the first try. However, if the configuration parameters that govern retries in the floppy disk BIOS are set to suitably higher values, then this value can be reduced. Remember that this value controls the boot retries for *all* drives in the system, floppy and otherwise.

Values:

n - Number of retries used to boot operating system.

Related Parameters:

FILE_SYSTEM - Enable file system support.

7.2.59 CONFIG_WAIT_8042 Parameter

The **CONFIG_WAIT_8042** parameter specifies the amount of time (in iterated loops) required for an 8042 command to be accepted.

The 8042 keyboard controller is a separate microcontroller that takes a certain amount of time to respond to requests submitted to its input ports. This parameter is used as a delay factor that when increased, results in a larger delay to account for slower 8042 controllers.

Values:

n - Number of iterations through a polling loop to wait for the 8042 to receive a command.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.

7.2.60 CONFIG_WAIT_8042_INIT Parameter

The **CONFIG_WAIT_8042_INIT** parameter specifies the number of CPU loops to be executed to allow the 8042 keyboard controller to recover after reading the BAT code during POST.

Values:

n – Specifies a wait value in CPU loops.

Related Parameters:

OPTION_SUPPORT_8042 – Enable 8042 keyboard controller support.
CONFIG_WAIT_8042 – Delay factor when waiting for 8042 to become ready.
CONFIG_SETTLE_8042 – Delay factor when waiting for 8042 accept commands.

7.2.61 CONFIG_SETTLE_8042 Parameter

The **CONFIG_SETTLE_8042** parameter specifies the amount of time (in iterated loops) required for an 8042 command to cause the A20 line to be gated.

The 8042 keyboard controller is a separate microcontroller that takes a certain amount of time to perform a given function. This parameter is used as a delay factor that when increased, results in a larger delay to account for slower 8042 controllers.

Values:

n - Number of iterations through a polling loop to wait for the 8042 to gate the A20 line circuit.

Related Parameters:

OPTION_SUPPORT_8042 - Enable 8042 support.

7.2.62 CONFIG_WAIT_COUNT Parameter

The **CONFIG_WAIT_COUNT** parameter specifies the delay used during POST's memory tests between blocks. This delay allows the user a chance to view the memory test as it is being performed, and also gives the user a chance to intervene and press the <Del> key to enter the SETUP system.

This value determines how many iterations of a CPU-controlled software loop is executed. The larger the value, the more loops will be used to kill time. Because this mechanism is CPU-speed specific, the value should be fine-tuned for your target.

Values:

n - Number of iterations through a polling loop to pause between memory block checks during POST.

Related Parameters:

OPTION_MEMTEST_WAIT - Enable the delay associated with this parameter.

OPTION_MEMTEST_CLICK - Enable speaker clicks during POST memory testing.

7.2.63 CONFIG_WAIT_LPT Parameter

The **CONFIG_WAIT_LPT** parameter specifies the amount of time (in iterated loops) wasted by POST when an LPT port is initialized.

Some parallel ports take additional time to settle when initialized.

Values:

n - Number of iterations through a polling loop to delay during initialization of LPT port.

Related Parameters:

OPTION_SUPPORT_PARALLEL - Enable parallel port support.

7.2.64 CONFIG_WAIT_IDE_INIT Parameter

The **CONFIG_WAIT_IDE_INIT** parameter specifies the time in seconds that the IDE file system will wait for IDE drives configured in the Setup Screen to initialize during POST. If a drive does not become ready within this minimum time period, the initialization of that drive will fail.

This timeout is used to handle the condition where a drive is connected improperly or simply not connected, although it is configured in the Setup Screen. The timeout permits POST to continue and give the user a chance to change parameters in the Setup Screen.

Values:

n – Timeout in seconds.

Related Parameters:

FILE_SYSTEM – Define IDE file system.

CONFIG_WAIT_IDE_IO – IDE timeout for drive I/O.

7.2.65 CONFIG_WAIT_IDE_IO Parameter

The **CONFIG_WAIT_IDE_IO** parameter specifies the time in seconds that the IDE file system will wait for IDE drives to perform I/O before a timeout failure is indicated.

Values:

n – Timeout in seconds.

Related Parameters:

FILE_SYSTEM – Define IDE file system.
CONFIG_WAIT_IDE_INIT – IDE timeout for drive initialization.

7.2.66 CONFIG_WAIT_PROGRESS_COM Parameter

The **CONFIG_WAIT_PROGRESS_COM** parameter specifies a CPU-specific delay that will be incurred between successive character writes to the POSTCODE_COM UART device, so that handshaking that might not be working properly during board bring-up is not required for pacing I/O to the UART.

The default value of this parameter is zero, a special value that indicates that the standard TBE method of pacing is used.

Values:

n – Delay in CPU loops.

Related Parameters:

OPTION_SUPPORT_POSTCODES_COM – Enable progress messages debugging feature.

7.2.67 CONFIG_SERIAL_TIMEOUT Parameter

The **CONFIG_SERIAL_TIMEOUT** parameter specifies the time in seconds to initialize the BIOS Data Area timeouts for all serial ports in the system.

The serial I/O services inspect the timeout value for a serial port when performing a read or write to the port.

Values:

n - Timeout value for all system serial ports, in seconds.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.68 CONFIG_PARALLEL_TIMEOUT Parameter

The **CONFIG_PARALLEL_TIMEOUT** parameter specifies the time in seconds to initialize the BIOS Data Area timeouts for all parallel ports in the system.

The parallel I/O services inspect the timeout value for a parallel port when performing a write to the port.

Values:

n - Timeout value for all system parallel ports, in seconds.

Related Parameters:

OPTION_SUPPORT_PARALLEL - Enable parallel I/O support.

7.2.69 CONFIG_POST_PROGRESS_PORT Parameter

The **CONFIG_POST_PROGRESS_PORT** parameter specifies I/O port that the **POSTCODE** macro uses to write POST progress codes to.

Ordinarily, this value is 80h. However, some systems can benefit by writing POST codes to other ports that can be both read and written, such as UART scratch registers (i.e., 2ffh, 3ffh). The benefit of using alternate ports in these targets is that Manufacturing Mode can retrieve the last POST code value and return it to the host.

OPTION_SUPPORT_POSTCODES must be enabled for this parameter to be effective.

Values:

*xx*h - I/O port assignment for POST progress port.

Related Parameters:

OPTION_SUPPORT_POSTCODES - Enable POSTCODE status codes.

7.2.70 CONFIG_POST_PROGRESS_COM Parameter

The **CONFIG_POST_PROGRESS_COM** parameter specifies base I/O port of an 8250-compatible UART that the **POSTCODECOM** macro uses to write ASCII characters to POST progress codes to.

In production systems, this feature is not used because it interferes with serial port initialization, takes time to work, and produces unwanted output. However, it can be very useful for debugging an otherwise inoperative target.

The value chosen for this parameter need not match one of the standard COM port addresses. The feature uses hard-coded OUT instructions to access the UART's data register and does not require other BIOS functionality to work. Typical values are 3f8h for COM1, or 2f8h for COM2.

OPTION_SUPPORT_POSTCODE_COM must be enabled for this parameter to be effective.

Values:

*xxx*h - Base I/O port of UART for POST status port.

Related Parameters:

OPTION_SUPPORT_POSTCODE_COM - Enable UART-based POST status codes.
CONFIG_POST_PROGRESS_BAUD - Specify baud rate for UART-based progress codes.

7.2.71 CONFIG_POST_PROGRESS_BAUD Parameter

The **CONFIG_POST_PROGRESS_BAUD** parameter baud rate to assign when initializing the 8250-compatible UART that the **POSTCODECOM** macro uses to write ASCII characters to POST progress codes to.

OPTION_SUPPORT_POSTCODE_COM must be enabled for this parameter to be effective.

Values:

COM_BAUD_110 - 110 baud.
COM_BAUD_150 - 150 baud.
COM_BAUD_300 - 300 baud.
COM_BAUD_600 - 600 baud.
COM_BAUD_1200 - 1200 baud.
COM_BAUD_2400 - 2400 baud.
COM_BAUD_4800 - 4800 baud.
COM_BAUD_9600 - 9600 baud.
COM_BAUD_19K - 19.2K baud.
COM_BAUD_28K - 28.4K baud.
COM_BAUD_56K - 56K baud.
COM_BAUD_115K - 115K baud.

Related Parameters:

OPTION_SUPPORT_POSTCODE_COM - Enable UART-based POST status codes.
CONFIG_POST_PROGRESS_COM - UART base address for status codes.

7.2.72 CONFIG_MFG_PROGRESS_PORT Parameter

The **CONFIG_MFG_PROGRESS_PORT** parameter specifies I/O port that the Manufacturing Mode will use to copy the incoming command codes to whenever incoming requests arrive. This is typically used to drive a 2-digit 7-segment hex display on an evaluation board for debugging.

Ordinarily, this value is 80h. However, some systems can benefit by writing the codes to other ports that can be both read and written, such as UART scratch registers (i.e., 2ffh, 3ffh). The benefit of using alternate ports in these targets is that the EMBEDDED BIOS debugger can be used to read the port and determine the last message code that was processed.

OPTION_SUPPORT_MFGCODES must be enabled for this parameter to be effective.

Values:

*xx*h - I/O port assignment for Manufacturing Mode progress port.

Related Parameters:

OPTION_SUPPORT_MFGCODES - Enable Manufacturing Mode status codes.

7.2.73 CONFIG_MAX_LOW_MEMORY Parameter

The **CONFIG_MAX_LOW_MEMORY** parameter specifies the maximum number of kilobytes of low memory to be scanned by POST.

This limit causes POST to stop its memory scan before running into special regions of the memory map, such as battery-backed RAM or video regeneration memory. The typical value for this parameter in ISA systems is 640, because VGA memory starts at segment A000h, which corresponds to the 640KB address mark.

If additional memory beyond the 640KB address mark is available (either because real memory is available or because shadow memory has been made available for the purpose of augmenting the size of the <1MB area), this parameter may be increased to present the memory to DOS.

Note that the Extended BIOS Data Area takes away from the top of low memory, by an amount that depends on the particular features enabled by the OEM. This is usually on the order of 1-5KB.

Values:

n - Amount of low memory to scan during POST, in kilobytes.

Related Parameters:

CONFIG_MAX_EXT_MEMORY - Limit of extended memory scan.

7.2.74 CONFIG_TESTBASE_SIZE Parameter

The **CONFIG_TESTBASE_SIZE** parameter specifies the maximum number of kilobytes of low memory to be tested before POST initializes the BIOS Data Area and creates its initial stack.

Ordinarily, this value is 64 (expressed in kilobytes), although this value can be reduced or expanded as necessary to accommodate nonstandard memory maps**. Do not set this value less than 8K (8) or greater than 64K (64).**

Testing of the base memory during POST is exhaustive; different destructive patterns are written during this phase, and it is expensive in terms of time to complete the test.

Values:

n - Amount of low memory to test as base RAM during POST, in kilobytes.

Related Parameters:

None.

7.2.75 CONFIG_MAX_EXT_MEMORY Parameter

The **CONFIG_MAX_EXT_MEMORY** parameter specifies the maximum number of kilobytes of extended memory to be scanned by POST.

This limit causes POST to stop its memory scan before running into special regions of the memory map, such as battery-backed RAM or Flash memory.

Values:

n - Amount of extended memory to scan during POST, in kilobytes.

Related Parameters:

CONFIG_MAX_LOW_MEMORY - Limit of low memory scan.

7.2.76 CONFIG_EXTRA_SEGMENT Parameter

The **CONFIG_EXTRA_SEGMENT** parameter specifies the initial location of the 1KB region known as the Extended BIOS Data Segment, before this region is moved to the top of low memory at a certain point during POST. By default, this region is started at segment 50h.

Values:

*nnnn*h - Segment address where the Extended BIOS Data Segment is initially created.

Related Parameters:

None.

7.2.77 CONFIG_FSINIT_SEGMENT Parameter

The **CONFIG_FSINIT_SEGMENT** parameter specifies the segment address of an area of memory used during POST's file system initialization. This parameter should not be modified without understanding how the file system initialization internals work.

Values:

*nnnn*h - Segment address of scratch space used during POST's file system initialization.

Related Parameters:

FILE_SYSTEM - Enable file systems in the BIOS.

7.2.78 CONFIG_DEFAULT_EQUIP_BYTE Parameter

The **CONFIG_DEFAULT_EQUIP_BYTE** parameter specifies the initial equipment byte to be used when CMOS is not available on PC and PC/XT-compatible systems.

This value is read by the core BIOS through PORT B; therefore, **OPTION_SUPPORT_PORT_B** must be enabled for this parameter to be effective.

Values:

*nn*h - Equipment byte contents (see CMOS.INC for equivalent bit definitions).

Related Parameters:

OPTION_SUPPORT_PORT_B - Enable PC & PC/XT-compatible peripheral access register.

7.2.79 CONFIG_VIDEO_ROM_SCAN Parameter

The **CONFIG_VIDEO_ROM_SCAN** parameter specifies the segment address to be scanned for an EGA or VGA ROM BIOS extension during the initialization of the video BIOS.

Normally, this value is 0C000h, but it can be changed to other values such as 0E000h, for example.

This value is excluded from the general ROM scan, even if it lies in the middle of the scan range.

OPTION_SUPPORT_VIDEO_BOARDS must be enabled in order for this parameter to be useful.

Values:

*nnnn*h - Segment address to be scanned for a video BIOS extension.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable base video support.
OPTION_SUPPORT_VIDEO_BOARDS - Enable video ROM scan.

7.2.80 CONFIG_LOW_ROM_SCAN Parameter

The **CONFIG_LOW_ROM_SCAN** parameter specifies the first segment address in a range to be scanned for non-video ROM BIOS extensions during the POST ROM scan.

Normally, this value is 0C800h. However, it can be changed to any segment value desired by the OEM, such that the value of this parameter is lower than the value of the **CONFIG_HIGH_ROM_SCAN** value.

The ROM scan addresses of Embedded DOS-ROM, and the video ROM extensions are excluded from this general scan so that these pieces of software are not initialized twice.

The ROM scan checks for ROM scan signatures at regular intervals specified by **CONFIG_ROM_SCAN_INTERVAL**. In desktop PC systems, this interval is fixed at 2048 bytes. However, in embedded designs where ROM space is more expensive, the interval can be reduced to values such as 1024 so that more ROM extensions can be packed together.

Values:

*nnnn*h - First segment address to be scanned for user ROM BIOS extensions.

Related Parameters:

OPTION_SUPPORT_ROM_EXTENSIONS - Enable general ROM scan.

CONFIG_HIGH_ROM_SCAN - Set upper limit of ROM scan.

CONFIG_ROM_SCAN_INTERVAL - Set increment for scan between lower and upper limits.

7.2.81 CONFIG_HIGH_ROM_SCAN Parameter

The **CONFIG_HIGH_ROM_SCAN** parameter specifies the first segment address above the range to be scanned for non-video ROM BIOS extensions during the POST ROM scan.

Normally, this value is 0DE00h. However, it can be changed to any segment value desired by the OEM, such that the value of this parameter is higher than the value of the **CONFIG_LOW_ROM_SCAN** value.

The ROM scan addresses of Embedded DOS-ROM, and the video ROM extensions are excluded from this general scan so that these pieces of software are not initialized twice.

The ROM scan checks for ROM scan signatures at regular intervals specified by **CONFIG_ROM_SCAN_INTERVAL**. In desktop PC systems, this interval is fixed at 2048 bytes. However, in embedded designs where ROM space is more expensive, the interval can be reduced to values such as 1024 so that more ROM extensions can be packed together.

Values:

*nnnn*h - First segment address *outside the range* to be scanned for user ROM BIOS extensions (this address will *not* be scanned).

Related Parameters:

OPTION_SUPPORT_ROM_EXTENSIONS - Enable general ROM scan.

CONFIG_LOW_ROM_SCAN - Set start address of ROM scan.

CONFIG_ROM_SCAN_INTERVAL - Set increment for scan between lower and upper limits.

7.2.82 CONFIG_ROM_SCAN_INTERVAL Parameter

The **CONFIG_ROM_SCAN_INTERVAL** parameter specifies the increment in addresses that is used during the general ROM scan to scan the range between **CONFIG_LOW_ROM_SCAN** and **CONFIG_HIGH_ROM_SCAN**.

In desktop PC systems, this interval is fixed at 2048 bytes. However, in embedded designs where ROM space is more expensive, the interval can be reduced to values such as 1024 so that more ROM extensions can be packed together.

Values:

n - Interval between scan points.

Related Parameters:

OPTION_SUPPORT_ROM_EXTENSIONS - Enable general ROM scan.

CONFIG_LOW_ROM_SCAN - Set start address of ROM scan.

CONFIG_HIGH_ROM_SCAN - Set upper limit of ROM scan.

7.2.83 CONFIG_MINI_DOS_SCAN Parameter

The **CONFIG_MINI_DOS_SCAN** parameter specifies the segment address to be scanned for the Embedded DOS-ROM system image (as generated by the Embedded DOS-ROM build process, a file called DOS.ROM).

Normally, this value is 0E000h, but can be changed to any segment value where Embedded DOS-ROM is located. If you change this value, you must relocate the Embedded DOS-ROM system file to the address you specify. Otherwise, it will not function properly.

This segment value is excluded from the general ROM scan so that Embedded DOS-ROM is not initialized twice.

OPTION_SUPPORT_MINI_DOS must be enabled for this parameter to be useful.

Values:

*nnnn*h - Segment address to be scanned for Embedded DOS-ROM.

Related Parameters:

OPTION_SUPPORT_MINI_DOS - Enable Embedded DOS-ROM scan.

7.2.84 CONFIG_PCI_ROM_SHADOW_START Parameter

The **CONFIG_PCI_ROM_SHADOW_START** parameter specifies the starting segment address of the upper memory area in a PCI system where PCI device option ROMs may be copied into read/write shadow memory.

The PCI chipset must be capable of shadowing in this region. The core BIOS automatically allocates space starting at this segment address, enabling shadowing as necessary to copy more option ROMs.

OPTION_SUPPORT_PCI must be enabled for this parameter to be useful.

Values:

*xxxx*h - Starting address of PCI option ROM shadow area.

Related Parameters:

OPTION_SUPPORT_PCI - Enable PCI support.

7.2.85 CONFIG_VIDEO_SEG_GRAPHIC Parameter

The **CONFIG_VIDEO_SEG_GRAPHIC** parameter specifies the segment address that the video BIOS will use when testing and manipulating video RAM, if available, when the video controller is in graphics mode.

In desktop PC systems, this value is 0A000h. However, in embedded designs employing nonstandard video controllers, this value can be adjusted to make room for additional low system RAM.

OPTION_SUPPORT_VIDEO must be enabled for this parameter to be useful. **OPTION_VIDEO_VIDEOMEM** needs to be specified if the video RAM should be tested during POST.

Values:

*xxxx*h - Segment address of video RAM in *graphics* mode.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video services.
OPTION_VIDEO_VIDEOMEM - Enable testing of video RAM during POST.

7.2.86 CONFIG_VIDEO_SEG_MONO Parameter

The **CONFIG_VIDEO_SEG_MONO** parameter specifies the segment address that the video BIOS will use when testing and manipulating video RAM, if available, when the video controller is in monochrome mode.

In desktop PC systems, this value is 0B000h. However, in embedded designs employing nonstandard video controllers, this value can be adjusted to make room for additional low system RAM.

OPTION_SUPPORT_VIDEO must be enabled for this parameter to be useful. **OPTION_VIDEO_VIDEOMEM** needs to be specified if the video RAM should be tested during POST.

Values:

*xxxx*h - Segment address of video RAM in *monochrome* mode.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video services.
OPTION_VIDEO_VIDEOMEM - Enable testing of video RAM during POST.

7.2.87 CONFIG_VIDEO_SEG_COLOR Parameter

The **CONFIG_VIDEO_SEG_COLOR** parameter specifies the segment address that the video BIOS will use when testing and manipulating video RAM, if available, when the video controller is in color mode.

In desktop PC systems, this value is 0B800h. However, in embedded designs employing nonstandard video controllers, this value can be adjusted to make room for additional low system RAM.

OPTION_SUPPORT_VIDEO must be enabled for this parameter to be useful. **OPTION_VIDEO_VIDEOMEM** needs to be specified if the video RAM should be tested during POST.

Values:

*xxxx*h - Segment address of video RAM in *color* mode.

Related Parameters:

OPTION_SUPPORT_VIDEO - Enable video services.
OPTION_VIDEO_VIDEOMEM - Enable testing of video RAM during POST.

7.2.88 CONFIG_BEEP_LENGTH Parameter

The **CONFIG_BEEP_LENGTH** parameter specifies the duration for speaker beeps to last.

This parameter is specified in "loop iteration" units, which is CPU-performance-specific. Normally, this value should be set to values around 200 for a 386SX-25, but can be adjusted to suit the CPU speed of the target.

Values:

n - Length of beeps.

Related Parameters:

OPTION_SUPPORT_SOUND - Enable speaker support.
OPTION_SUPPORT_PORT_B - Enable PORT B peripheral access.
CONFIG_BEEP_CYCLE - Wavelength of beep.

7.2.89 CONFIG_BEEP_CYCLE Parameter

The **CONFIG_BEEP_CYCLE** parameter specifies the micro delay used to create a square wave, when the 8042 timer cannot be programmed to deliver an accurate tone through the speaker.

Normally, this value should start around 100 for a CPU with a performance of about a 386SX-25, but can be adjusted to suit the CPU speed of the target.

Values:

n - Inverse frequency of beeps (actually, the wavelength).

Related Parameters:

OPTION_SUPPORT_SOUND - Enable speaker support.
OPTION_SUPPORT_PORT_B - Enable PORT B peripheral access.
CONFIG_BEEP_LENGTH - Duration of beep.

7.2.90 CONFIG_BEEP_8254_TONE Parameter

The **CONFIG_BEEP_8254_TONE** parameter specifies the divisor to be used when programming the 8254's T2 timer to generate beeps for the speaker.

This is a more reliable way to specify beep frequency because it is controlled by a system that is clocked independently from the CPU. However, POST cannot use this tone production mechanism before the 8254 counter-timer has been initialized, so the other method that uses **CONFIG_BEEP_CYCLE** must also be supported.

The **CONFIG_BEEP_LENGTH** is used to determine the length of tones produced with the 8254 as well as those produced manually with **CONFIG_BEEP_CYCLE**.

Values:

n - 8254's T2 divisor value that determines frequency of beeps once 8254 hardware is initialized.

Related Parameters:

OPTION_SUPPORT_SOUND - Enable speaker support.
OPTION_SUPPORT_8254 - Enable 8254 support.
CONFIG_BEEP_LENGTH - Duration of beep.

7.2.91 CONFIG_PCMCIA_IOBASE Parameter

The **CONFIG_PCMCIA_IOBASE** parameter specifies the base I/O port of the PCMCIA controller being used when **OPTION_SUPPORT_ATA** is enabled.

This allows the PCMCIA controller to be placed anywhere in the I/O space of the target. By default, the Cirrus Logic 6710 and 6720 controllers are located at address 3e0h, but this can be changed by editing this value to locate the part anywhere.

Values:

*xxx*h - Base I/O address of PCMCIA controller.

Related Parameters:

OPTION_SUPPORT_ATA - Enable ATA PC Cards over PCMCIA controller.

7.2.92 CONFIG_RFDDISK_KBBLKSIZE Parameter

The **CONFIG_RFDDISK_KBBLKSIZE** parameter specifies the size of the minimum erasable unit (Flash block) within the Flash array to be used by the Resident Flash Disk (RFD), in kilobytes.

Typically, Flash blocks are a power of 2 in size. For example, 16KB, 32KB, 64KB, 128KB, and so on. The block size is a device parameter that is not changable by simply changing this parameter; instead, this parameter must be modified to fit the block size associated with the Flash devices you are using.

If you have a Flash array that is interleaved (i.e., two 8-bit parts ganged together to form a 16-bit data path, etc.), then make sure you take into account that 2-way part interleaving effectively doubles the block size, and 4-way part interleaving quadruples it.

The **FILE_SYSTEM** macro is used to define RFDs in the system. Consult that section for more information about how to specify the starting address and size of the RFD. For information about the RFD disk, see Chapter 12.

Values:

n - Size of the RFD's Flash array blocks in kilobytes.

Related Parameters:

FILE_SYSTEM - Enable RFD support.

7.2.93 CONFIG_FLASH_DATASEG Parameter

The **CONFIG_FLASH_DATASEG** parameter specifies a segment address that Manufacturing Mode can use as a RAM buffer for staging incoming and outgoing data over the serial link.

This allows the host to download several messages into one contiguous buffer, which can then be written to Flash with one target operation.

The staging buffer occupies 64KB of RAM. Typically, this buffer is located at segment 2000h, so that it does not interfere with the **CONFIG_FLASH_CODESEG** parameter or low memory where the interrupt vector table and BIOS data area are stored.

OPTION_SUPPORT_MCL must be enabled for this parameter to be useful.

Values:

*xxxx*h - Segment address of 64KB scratch area for Manufacturing Mode staging buffer.

Related Parameters:

OPTION_SUPPORT_MCL - Enable Flash support.
OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

7.2.94 CONFIG_FLASH_CODESEG Parameter

The **CONFIG_FLASH_CODESEG** parameter specifies a segment address that Manufacturing Mode can use as a RAM buffer for copying the BIOS to so that it can execute out of RAM when programming the Flash.

This allows the host to cause the target to reprogram the BIOS Flash itself and continue executing. The BIOS Flash routines cannot run out of the same device that is being programmed, because (1) it must be erased, and (2) the Flash enters a command/status mode instead of a read mode, so that instructions fetched out of the Flash would not be instruction bytes, but status bytes.

This code segment buffer requires 64KB of RAM. Typically, this buffer is located at segment 1000h, so that it does not interfere with the **CONFIG_FLASH_DATASEG** parameter or low memory where the interrupt vector table and BIOS data area are stored.

OPTION_SUPPORT_MCL must be enabled for this parameter to be useful.

Values:

*xxxx*h - Segment address of 64KB scratch area for Manufacturing Mode to run a copy of the BIOS from.

Related Parameters:

OPTION_SUPPORT_MCL - Enable Flash support.
OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

7.2.95 CONFIG_PAGED_MEM_SEG Parameter

The **CONFIG_PAGED_MEM_SEG** parameter specifies the segment address of a memory window below the 1MB address marker that the OEM defines to map to a specific page of some physical device, such as a RAM, ROM, EPROM, or Flash part.

This value is used by the Chipset Personality Module in some cases (for example, the SC300, SC310, SC400, and SC410 processors by AMD) to determine how to program the memory management unit on the chipset itself for Flash operations.

Not all values are valid for all chipsets. Please note that adjusting this value may imply a change in the way the chipset's memory management hardware is used; i.e., some addresses are handled with MMSA, and others with MMSB, on AMD Elan processors.

Values:

xxxxh - Specifies the real-mode segment address of the memory window.

Related Parameters:

OPTION_SUPPORT_MCL - Enable Flash support.

7.2.96 CONFIG_VPP_TIMEOUT_IN_TICKS Parameter

The **CONFIG_VPP_TIMEOUT_IN_TICKS** parameter specifies the number of 55ms timer ticks to pass after the last Flash function is requested by any portion of the BIOS (RFD, debugger, Manufacturing Mode, etc.) before Vpp is disabled.

EMBEDDED BIOS Flash Media Technology Drivers (MTDs) can take advantage of automatic Vpp regulation in the core BIOS by making calls to enable and disable Vpp at appropriate points inside the MTDs. Controlling Vpp involves OEM-proprietary methods, so a call to the OEM's Board Personality Module hides the actual mechanism. Normally, raising Vpp is followed by a delay (OEM-specific) to ensure that Vpp has had adequate time to become stable before being used. This delay is unnecessarily incurred if Vpp is commanded to go high before an operation, then commanded to go low, followed by the same sequence, many times. To improve the performance of back-to-back Flash I/O, EMBEDDED BIOS implements lazy Vpp regulation that causes MTD disable Vpp requests to start a timer (specified by this parameter). When the timer expires (at system tick time in the background), Vpp is disabled by a call to the board module.

The higher this parameter is specified, the longer Vpp will be left on after the last Flash I/O operation. Since erase commands can occur in the background and be temporarily preempted by reads in some MTDs, Vpp is left on even during read operations. Leaving Vpp on for an excessive amount of time wastes battery power in low-power applications. Leaving Vpp on for too short a time after the last I/O could lead to data loss.

The minimum time to set this value to should be the time necessary for the longest operation to proceed in the background, plus some margin for error. On some Flash devices, erase time can exceed two or three seconds.

Values:

n - Specifies the number of timer ticks to delay after the last Flash operation before Vpp is disabled in the background to save battery power.

Related Parameters:

OPTION_SUPPORT_MCL - Enable Flash programming support.

7.2.97 CONFIG_PCI_ROM_MAP Parameter

The **CONFIG_PCI_ROM_MAP** parameter specifies the top 16 bits of the 32-bit physical address to use in PCI systems for temporarily mapping device option ROM extensions during the time that they are copied into shadow memory. Note that this is only a temporary mapping during the copy process during PCI POST, and is the same for each PCI option ROM, because they are copied one at a time.

The full 32-bit physical address is formed by using the 16 bits specified by this parameter as the high 16 bits in a 32-bit address. The bottom 16 bits are always zeroes.

Commonly, the physical address is configured so as not to interfere with any boot ROM mapped to the top of the address space, yet be positioned beyond any reasonable address space that might be consumed by main memory.

Values:

xxxxh - Specifies the top 16 bits of a 32-bit physical address to map PCI ROM extensions.

Related Parameters:

OPTION_SUPPORT_PCI - Enable PCI support.

7.2.98 CONFIG_PCI_MEM_AVAIL Parameter

The **CONFIG_PCI_MEM_AVAIL** parameter specifies the top 16 bits of the first 32-bit physical address that is to be made available to PCI devices requesting memory address space during PCI POST. As each device requests its own memory address space, the 32-bit pointer is advanced by the core BIOS so that each device is able to acquire a unique range of memory addresses.

The full 32-bit physical address is formed by using the 16 bits specified by this parameter as the high 16 bits in a 32-bit address. The bottom 16 bits are always zeroes.

Commonly, the physical address is configured so as not to interfere with any boot ROM mapped to the top of the address space, yet be positioned beyond any reasonable address space that might

be consumed by main memory. Additionally, this address space must be positioned so as not to interfere with the address space defined by the **CONFIG_PCI_ROM_MAP** parameter.

Values:

xxxxh - Specifies the top 16 bits of the first 32-bit physical to be made available to PCI devices as a memory address space resource.

Related Parameters:

OPTION_SUPPORT_PCI - Enable PCI support.

7.2.99 CONFIG_PCI_IO_PORT_BASE Parameter

The **CONFIG_PCI_IO_PORT_BASE** parameter specifies the starting 16-bit I/O port address used by the PCI subsystem for its allocation of I/O address space resources to devices during system initialization.

Normally, this parameter is defaulted to FC00h, and the PCI subsystem decrements it by **CONFIG_PCI_IO_ALLOC** (normally 400h) to avoid conflicts due to ISA aliasing. This is the effective base address to be used when assigning I/O resources to PCI devices. However, it is not the first actual address to be assigned; instead, the extra offset 100h is added to this base address to skip past the first 100h hex bytes that conflict with ISA I/O ports.

According to the PCI Specification, configuration software should configure PCI devices such that no conflicts exist. This can be performed in one of two ways. In the first method the configuration software can explicitly know what ISA devices are in the system, what addresses and aliases those devices use, and then configure PCI devices so that no conflicts occur. The major problem with this method is that the automatic detection of ISA devices as well as determining what resources they consume is very difficult. The second method is to pre-allocate three-fourths of the address space to ISA devices and their aliases. PCI devices are placed in the remaining spaces. Essentially, and I/O address that is greater than 4K (to avoid platform ISA devices) and where SA<8-9>=00b (to avoid ISA aliases) is a valid address for PCI devices. This technique provides for sixty 256 byte-wide addresses where PCI devices can be mapped without conflicting with platform or ISA devices. This is the preferred method for allocating I/O address space in a PCI based system. In practice, this means that I/O addresses 0100h-01ffh, 1100h-11ffh, 1500h-15ffh, 1900h-19ffh, 2100h-21ffh, and so on, are available for PCI use in a mixed PCI/ISA system.

As a matter of practice, the PCI POST process attempts to allocate PCI I/O addresses from the top, rather than the bottom, of the I/O address range, so as to minimize any possible conflict with ISA I/O ports. Thus, the first I/O range assigned to a PCI device in the above example would actually be fd00h-fdffh, then f900h-f9ffh, and so on.

Values:

xxxxh - Specifies the 16-bit I/O base address from which I/O port ranges may be assigned to PCI devices during initialization. Note that this value does not account for the extra 100h byte offset needed to avoid ISA aliasing; the 100h is automatically added by the PCI subsystem.

Related Parameters:

OPTION_SUPPORT_PCI - Enable PCI support.
CONFIG_PCI_IO_ALLOC – Amount to subtract from **CONFIG_PCI_IO_PORT_BASE** to obtain next available base address in system.

7.2.100 CONFIG_PCI_IO_ALLOC Parameter

The **CONFIG_PCI_IO_ADDRESS** parameter specifies the number of consecutive I/O addresses to be found at the I/O address specified by **CONFIG_PCI_IO_PORT_BASE**.

This parameter does not specify how many addresses are available for allocation; rather, it defines the value to be used to decrement **CONFIG_PCI_IO_PORT_BASE** in the allocation algorithm.

For details about the PCI I/O space assignment process, consult the description of **CONFIG_PCI_IO_BASE**.

Values:

n - Specifies the value to be subtracted from **CONFIG_PCI_IO_PORT_BASE** in an interative algorithm used to generate new I/O port ranges for assignment.

Related Parameters:

OPTION_SUPPORT_PCI - Enable PCI support.
CONFIG_PCI_IO_PORT_BASE – First (and highest) I/O base address used by PCI subsystem to generate available I/O ranges for assignments to devices.

7.2.101 CONFIG_PCI_IO_TMP_TBL_SEG Parameter

The **CONFIG_PCI_IO_TMP_TBL_SEG** parameter specifies the scratch RAM segment to use for copying the I/O table to during POST's PCI enumeration.

Values:

n – 16-bit segment address of scratch area for PCI during POST.

Related Parameters:

OPTION_SUPPORT_PCI – Enable PCI support.
CONFIG_PCI_IO_BM_OFFSET – Specify offset to use within scratch segment for PCI bus map during POST.

7.2.102 CONFIG_PCI_BM_OFFSET Parameter

The **CONFIG_PCI_BM_OFFSET** parameter specifies the offset into the scratch segment to store the PCI bus map.

Values:

n – 16-bit offset within the scratch area.

Related Parameters:

OPTION_SUPPORT_PCI – Enable PCI support.
CONFIG_PCI_IO_DATASEG – Specify scratch segment address.

7.2.103 CONFIG_PCI_MMIO_AVAIL Parameter

The **CONFIG_PCI_MMIO_AVAIL** parameter specifies the high 16 bits of the physical address for non-prefetchable memory given to PCI devices.

Values:

n – top 16 bits of a 32-bit physical address.

Related Parameters:

OPTION_SUPPORT_PCI – Enable PCI support.

7.2.104 CONFIG_PCI_LATENCY Parameter

The **CONFIG_PCI_LATENCY** parameter specifies the value to be programmed into the latency field in the header of each PCI device during PCI enumeration.

Values:

n – 8-bit initial latency value to be programmed into PCI headers.

Related Parameters:

OPTION_SUPPORT_PCI – Enable PCI support.

7.2.105 CONFIG_PCI_IRQ_BITMAP Parameter

The **CONFIG_PCI_IRQ_BITMAP** parameter specifies a 16-bit bitmask containing bits that, when set, indicate that the associated IRQ may be allocated by the PCI subsystem.

Bit 0 corresponds to IRQ0, bit 1 corresponds to IRQ1, and so on, up to bit 15's association with IRQ15.

Not all the specified IRQs will necessarily be assigned to PCI devices. The PCI subsystem will choose up to four IRQs from the specified set, and map them to the INTA, INTB, INTC, and INTD PCI bus lines.

Values:

n – 16-bit bitmask of IRQs assignable to PCI devices by PCI subsystem.

Related Parameters:

OPTION_SUPPORT_PCI – Enable PCI support.

7.2.106 CONFIG_PS2_MOUSE_IRQ Parameter

The **CONFIG_PCI_PS2_MOUSE_IRQ** parameter specifies system interrupt request level used by the keyboard controller to generate mouse interrupts. Normally, this value is 12, but may be assigned to any IRQ as appropriate for the platform.

Values:

n - Specifies an IRQ level from 0 to 15.

Related Parameters:

OPTION_SUPPORT_PS2MOUSE - Enable mouse support.
CONFIG_PS2_MOUSE_LOOP - Specify device timeout for PS/2 mouse.

7.2.107 CONFIG_PS2_MOUSE_LOOP Parameter

The **CONFIG_PS2_MOUSE_LOOP** parameter specifies a timeout value, in CPU loops, to wait for the keyboard controller to return status information about the PS/2 mouse device after it has been commanded to provide status.

Values:

n - Specifies a timeout value from 1 to 65535.

Related Parameters:

OPTION_SUPPORT_PS2MOUSE - Enable mouse support.
CONFIG_PS2_MOUSE_IRQ - Specify PS/2 mouse interrupt level.

7.2.108 CONFIG_IDE_PORT_BASE Parameter

The **CONFIG_IDE_PORT_BASE** parameter specifies a base I/O address, to which are added 0f0h or 070h for the primary or secondary IDE controllers, respectively. By default, this parameter has the value 100h, so that the address 1f0h and 170h are used.

Do not change this value with out a full understanding of how the IDE and ATA file system drivers work.

Values:

*nnn*h – Base I/O port number which is used to compute the primary and secondary IDE controller addresses.

Related Parameters:

FILE_SYSTEM - Enable file system.

7.2.109 CONFIG_IDE_PORT_ALT_STATUS Parameter

The **CONFIG_IDE_PORT_ALT_STATUS** parameter specifies a value to be added to the f0h or 70h values associated with the primary and secondary controllers, respectively, to access the IDE alternate status register for each controller.

Normally, this parameter has the value 306h, so that the alternate status registers for the primary and secondary controller are 3f6h and 376h, respectively.

Do not change this value with out a full understanding of how the IDE and ATA file system drivers work.

Values:

*nnn*h – Base I/O port number which is used to compute the primary and secondary IDE controller's alternate status register addresses.

Related Parameters:

FILE_SYSTEM - Enable file system.

7.2.110 CONFIG_IDE_PORT_CTRL Parameter

The **CONFIG_IDE_PORT_CTRL** parameter specifies a value to be added to the f0h or 70h values associated with the primary and secondary controllers, respectively, to access the IDE control register for each controller.

Normally, this parameter has the value 306h, so that the control registers for the primary and secondary controller are 3f6h and 376h, respectively.

Do not change this value with out a full understanding of how the IDE and ATA file system drivers work.

Values:

*nnn*h – Base I/O port number which is used to compute the primary and secondary IDE controller's control register addresses.

Related Parameters:

FILE_SYSTEM - Enable file system.

7.2.111 LPT1_BASE Parameter

The **LPT1_BASE** parameter specifies the I/O port to be scanned for the existence of the primary parallel port.

Values:

*nnn*h - I/O port number associated with the LPT port. The value 0 indicates no port assignment.

Related Parameters:

OPTION_SUPPORT_PARALLEL - Enable parallel I/O support.

7.2.112 LPT2_BASE Parameter

The **LPT2_BASE** parameter specifies the I/O port to be scanned for the existence of the secondary parallel port.

Values:

*nnn*h - I/O port number associated with the LPT port. The value 0 indicates no port assignment.

Related Parameters:

OPTION_SUPPORT_PARALLEL - Enable parallel I/O support.

7.2.113 LPT3_BASE Parameter

The **LPT3_BASE** parameter specifies the I/O port to be scanned for the existence of the third parallel port.

Values:

*nnn*h - I/O port number associated with the LPT port. The value 0 indicates no port assignment.

Related Parameters:

OPTION_SUPPORT_PARALLEL - Enable parallel I/O support.

7.2.114 COM1_BASE Parameter

The **COM1_BASE** parameter specifies the I/O port to be scanned for the existence of the first external serial port.

Values:

*nnn*h - I/O port number associated with the COM port. The value 0 indicates no port assignment.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.115 COM2_BASE Parameter

The **COM2_BASE** parameter specifies the I/O port to be scanned for the existence of the second external serial port.

Values:

*nnn*h - I/O port number associated with the COM port. The value 0 indicates no port assignment.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.116 COM3_BASE Parameter

The **COM3_BASE** parameter specifies the I/O port to be scanned for the existence of the third external serial port.

Values:

*nnn*h - I/O port number associated with the COM port. The value 0 indicates no port assignment.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.117 COM4_BASE Parameter

The **COM4_BASE** parameter specifies the I/O port to be scanned for the existence of the fourth external serial port.

Values:

*nnn*h - I/O port number associated with the COM port. The value 0 indicates no port assignment.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.118 COM1_INIT Parameter

The **COM1_INIT** parameter specifies the initialization byte used to program the first external serial port.

The value is passed to the Initialize Serial Port function of INT 14h during POST.

Values:

*nn*h - Initialization byte specifying baud rate, parity, number of data bits, and number of stop bits in an encoded fashion as defined by the INT 14h standard initialization function. The default value of 11100011b initializes the serial port to 9600 baud, no parity, 8 data bits, and one stop bit.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.119 COM2_INIT Parameter

The **COM2_INIT** parameter specifies the initialization byte used to program the second external serial port.

The value is passed to the Initialize Serial Port function of INT 14h during POST.

Values:

*nn*h - Initialization byte specifying baud rate, parity, number of data bits, and number of stop bits in an encoded fashion as defined by the INT 14h standard initialization function. The default value of 11100011b initializes the serial port to 9600 baud, no parity, 8 data bits, and one stop bit.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.120 COM3_INIT Parameter

The **COM3_INIT** parameter specifies the initialization byte used to program the third external serial port.

The value is passed to the Initialize Serial Port function of INT 14h during POST.

Values:

*nn*h - Initialization byte specifying baud rate, parity, number of data bits, and number of stop bits in an encoded fashion as defined by the INT 14h standard initialization

function. The default value of 11100011b initializes the serial port to 9600 baud, no parity, 8 data bits, and one stop bit.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.121 COM4_INIT Parameter

The **COM4_INIT** parameter specifies the initialization byte used to program the fourth external serial port.

The value is passed to the Initialize Serial Port function of INT 14h during POST.

Values:

*nn*h - Initialization byte specifying baud rate, parity, number of data bits, and number of stop bits in an encoded fashion as defined by the INT 14h standard initialization function. The default value of 11100011b initializes the serial port to 9600 baud, no parity, 8 data bits, and one stop bit.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.

7.2.122 MFG_COM_BASE Parameter

The **MFG_COM_BASE** parameter specifies the base I/O port of the UART to be used by Manufacturing Mode. The UART must be 8250 compatible.

The UART does not have to be one of the standard ones assigned to COM1, COM2, COM3, or COM4, but this is commonly the case.

OPTION_SUPPORT_MFGMODE must be enabled for this parameter to be useful.

Values:

*nnn*h - Base I/O port of the UART to be used by Manufacturing Mode.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

CONFIG_MFG_BAUD - Encoded baud rate to be used by Manufacturing Mode.

MFG_INT_VECT - Interrupt vector used by the UART.

MFG_EOI_PORT - Interrupt controller port to be used to acknowledge serial interrupts during Manufacturing Mode.

MFG_EOI_CMD - End-Of-Interrupt command to be issued to interrupt controller during Manfacturing Mode.

7.2.123 MFG_INT_VECT Parameter

The **MFG_INT_VECT** parameter specifies the interrupt vector number associated with the UART to be used by Manufacturing Mode. The interrupt vector is needed to support interrupt-driven receives of RS-232 data from the host.

OPTION_SUPPORT_MFGMODE must be enabled for this parameter to be useful.

Values:

*nn*h - Interrupt vector number associated with the UART to be used by Manufacturing Mode.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

CONFIG_MFG_BAUD - Encoded baud rate to be used by Manufacturing Mode.

MFG_COM_BASE - Base I/O port of UART to be used by Manufacturing Mode.

MFG_EOI_PORT - Interrupt controller port to be used to acknowledge serial interrupts during Manufacturing Mode.

MFG_EOI_CMD - End-Of-Interrupt command to be issued to interrupt controller during Manfacturing Mode.

7.2.124 CONFIG_MFG_BAUD Parameter

The **CONFIG_MFG_BAUD** parameter specifies the baud rate as an encoded number to be used by Manufacturing Mode.

OPTION_SUPPORT_MFGMODE must be enabled for this parameter to be useful.

Values:

0 - 115k baud.
1 - 56k baud.
2 - 38.4k baud.
3 – 28.8k baud.
4 – 19.2k baud.
5 – 9600 baud.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

MFG_COM_BASE - Base I/O port of UART to be used by Manufacturing Mode.

MFG_INT_VECT - Interrupt vector associated with UART to be used by Manufacturing Mode.

MFG_EOI_PORT - Interrupt controller port to be used to acknowledge serial interrupts during Manufacturing Mode.

MFG_EOI_CMD - End-Of-Interrupt command to be issued to interrupt controller during Manfacturing Mode.

7.2.125 MFG_EOI_PORT Parameter

The **MFG_EOI_PORT** parameter specifies interrupt controller's command port that can be used to dismiss an interrupt during serial communications in Manufacturing Mode.

This parameter is useful for situations where the COM port being used generates an interrupt via a nonstandard interrupt controller.

For designs using COM1 or COM2, the primary interrupt controller at I/O port 20h is used, unless COM1 or COM2 are CPU UARTs.

OPTION_SUPPORT_MFGMODE must be enabled for this parameter to be useful.

Values:

xxxh - I/O port associated with interrupt controller's command register.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

CONFIG_MFG_BAUD - Baud rate associated with Manufacturing Mode.

MFG_COM_BASE - Base I/O port of UART to be used by Manufacturing Mode.

MFG_INT_VECT - Interrupt vector associated with UART to be used by Manufacturing Mode.

MFG_EOI_CMD - End-Of-Interrupt command to be issued to interrupt controller during Manfacturing Mode.

7.2.126 MFG_EOI_CMD Parameter

The **MFG_EOI_CMD** parameter specifies interrupt controller's EOI command to be used to dismiss an interrupt during serial communications in Manufacturing Mode.

This parameter is useful for situations where the COM port being used generates an interrupt via a nonstandard interrupt controller.

For designs using COM1 or COM2, the primary interrupt controller at I/O port 20h is used, unless COM1 or COM2 are CPU UARTs. The non-specific EOI command for this controller (or any 8259) is 20h.

OPTION_SUPPORT_MFGMODE must be enabled for this parameter to be useful.

Values:

*xx*h - 8-bit command to be written to interrupt controller's command port as End-Of-Interrupt command.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

CONFIG_MFG_BAUD - Baud rate associated with Manufacturing Mode.

MFG_COM_BASE - Base I/O port of UART to be used by Manufacturing Mode.

MFG_INT_VECT - Interrupt vector associated with UART to be used by Manufacturing Mode.

MFG_EOI_PORT - Interrupt controller port to be used to acknowledge serial interrupts during Manufacturing Mode.

7.2.127 CONFIG_MFG_BUFSIZE Parameter

The **CONFIG_MFG_BUFSIZE** parameter specifies the size of the packet buffer to be used by Manufacturing Mode. This parameter effectively specifies the maximum message size that can be transferred between the host and the target.

This parameter does not affect the circular buffer size, which is simply a buffer to handle differences in speeds of the target and host.

OPTION_SUPPORT_MFGMODE must be enabled for this parameter to be useful.

Values:

n - Size of message buffer in bytes (must be greater than or equal to 768).

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

7.2.128 CONFIG_MFG_CBSIZE Parameter

The **CONFIG_MFG_CBSIZE** parameter specifies the size of the circular buffer to be used by Manufacturing Mode. The circular buffer is used for interrupt-driven receives of bytes from the host. This parameter may specify a value less than the message buffer size, since the bytes are assembled in the message buffer, and only staged in the circular buffer.

OPTION_SUPPORT_MFGMODE must be enabled for this parameter to be useful.

Values:

n - Size of circular buffer in bytes (values greater than 32 recommended; 64 typical).

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

7.2.129 CONFIG_MFG_TIMEOUT Parameter

The **CONFIG_MFG_TIMEOUT** parameter specifies the number of 18.2Hz timer ticks to wait for the reception of a character before the message being received is deemed to be timed-out (and therefore discarded).

Increasing this parameter makes the target more forgiving when working with links that may become disconnected frequently. Decreasing this parameter makes the target respond more quickly to errors so that the operation can be retried.

OPTION_SUPPORT_MFGMODE must be enabled for this parameter to be useful.

Values:

n - Number of 18.2Hz ticks to wait for a byte until a timeout occurs.

Related Parameters:

OPTION_SUPPORT_MFGMODE - Enable Manufacturing Mode support.

7.2.130 CONFIG_CON_REDIR_STD Parameter

The **CONFIG_CON_REDIR_STD** parameter specifies the device used for standard POST and DOS console I/O. A value of 0 indicates the PC keyboard and video display, whereas nonzero values indicate the COM port number associated with the redirected I/O.

The redirection feature itself is enabled with the **OPTION_SUPPORT_CON_REDIRECTOR** option, which must be enabled for the I/O to be redirected over a serial port.

The **OPTION_VIDEO_DUPLICATE** option can be enabled to duplicate the redirected output to the standard video screen. Be aware that any VGA BIOS extension in the system is likely to hook the INT 10h vector and make it impossible for the console redirection code to receive control. In the lab environment, this can be solved by using a monochrome or color adapter.

Values:

0 - Do not redirect standard I/O over serial port; instead, use keyboard and video display.
1 - Redirect standard I/O over COM1.
2 - Redirect standard I/O over COM2.
3 - Redirect standard I/O over COM3.
4 - Redirect standard I/O over COM4.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.
OPTION_SUPPORT_VIDEO - Enable video controller support.
OPTION_SUPPORT_KEYBOARD - Enable PC keyboard support.
COM1_BASE - I/O base of COM1 UART.
COM2_BASE - I/O base of COM2 UART.
COM3_BASE - I/O base of COM3 UART.
COM4_BASE - I/O base of COM4 UART.

7.2.131 CONFIG_CON_REDIR_DEBUG Parameter

The **CONFIG_CON_REDIR_DEBUG** parameter specifies the device used the BIOS debugger's console I/O. A value of 0 indicates the PC keyboard and video display, whereas nonzero values indicate the COM port number associated with the redirected I/O.

The redirection feature itself is enabled with the **OPTION_SUPPORT_CON_REDIRECTOR** option, which must be enabled for the I/O to be redirected over a serial port.

The **OPTION_VIDEO_DUPLICATE** option can be enabled to duplicate the redirected output to the standard video screen. Be aware that any VGA BIOS extension in the system is likely to hook the INT 10h vector and make it impossible for the console redirection code to receive control. In the lab environment, this can be solved by using a monochrome or color adapter.

Values:

0 - Do not redirect debugger I/O over serial port; instead, use keyboard and video display.
1 - Redirect debugger I/O over COM1.
2 - Redirect debugger I/O over COM2.
3 - Redirect debugger I/O over COM3.
4 - Redirect debugger I/O over COM4.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.
OPTION_SUPPORT_VIDEO - Enable video controller support.
OPTION_SUPPORT_KEYBOARD - Enable PC keyboard support.
COM1_BASE - I/O base of COM1 UART.
COM2_BASE - I/O base of COM2 UART.
COM3_BASE - I/O base of COM3 UART.
COM4_BASE - I/O base of COM4 UART.

7.2.132 CONFIG_CON_REDIR_SETUP Parameter

The **CONFIG_CON_REDIR_SETUP** parameter specifies the device used the BIOS SETUP screen's console I/O. A value of 0 indicates the PC keyboard and video display, whereas nonzero values indicate the COM port number associated with the redirected I/O.

The redirection feature itself is enabled with the **OPTION_SUPPORT_CON_REDIRECTOR** option, which must be enabled for the I/O to be redirected over a serial port.

The **OPTION_VIDEO_DUPLICATE** option can be enabled to duplicate the redirected output to the standard video screen. Be aware that any VGA BIOS extension in the system is likely to hook the INT 10h vector and make it impossible for the console redirection code to receive control. In the lab environment, this can be solved by using a monochrome or color adapter.

Values:

0 - Do not redirect standard I/O over serial port; instead, use keyboard and video display.
1 - Redirect SETUP I/O over COM1.
2 - Redirect SETUP I/O over COM2.
3 - Redirect SETUP I/O over COM3.
4 - Redirect SETUP I/O over COM4.

Related Parameters:

OPTION_SUPPORT_SERIAL - Enable serial I/O support.
OPTION_SUPPORT_VIDEO - Enable video controller support.
OPTION_SUPPORT_KEYBOARD - Enable PC keyboard support.
COM1_BASE - I/O base of COM1 UART.
COM2_BASE - I/O base of COM2 UART.
COM3_BASE - I/O base of COM3 UART.
COM4_BASE - I/O base of COM4 UART.

7.2.133 BIOS_HDWR Parameter

The **BIOS_HDWR** parameter specifies the class of machine (in desktop PC terms) that best describes the target.

The standard value of **BIOS_MODEL_AT** is used to describe ISA configurations. This value is placed immediately after the power-on JMP statement and is inspected by some utility programs and operating systems.

The most useful information that can be derived from the model byte is the processor type and the type of keyboard controller that is available. *Some software, such as* `HIMEM.SYS`*, uses this information to determine the machine type.*

Port 92h is available on all of the PS/2-compatible models, and now on many PC/AT-compatible machines which do not have PS/2 MCA busses.

The PS/2 model 80 is a 386-based machine that, unlike the other PS/2 models, supports a 32-bit address space and a way to switch to real mode with an instruction instead of a reboot sequence.

The PC Jr contains polled Floppy I/O and therefore has a very strange I/O model.

Systems that have an 8042 keyboard controller and do not have port 92h should stick to the PC/AT model byte. General Software recommends that you use the default value, **BIOS_MODEL_AT**, if you have hardware that reasonably resembles a desktop 386 machine or better, with an ISA, PCI, or local bus design.

Values:

BIOS_MODEL_PC - IBM PC compatible.
BIOS_MODEL_XT - IBM PC/XT compatible.
BIOS_MODEL_JR - IBM PC Jr compatible.
BIOS_MODEL_AT - IBM PC/AT compatible (recommended).
BIOS_MODEL_PS2_30 - IBM PS/2 Model 30 compatible.
BIOS_MODEL_CVT - IBM PC Convertable compatible.
BIOS_MODEL_PS2_80 - IBM PS/2 Model 80 compatible.

Related Parameters:

BIOS_HDWR_SUB - Submodel byte.

7.2.134 BIOS_HDWR_SUB Parameter

The **BIOS_HDWR_SUB** parameter specifies the subclass of machine (in desktop PC terms) that best describes the target.

This information is rarely used by application or system programs, but is provided so that the OEM can strictly emulate a model/submodel combination on a target.

The main difference between the XT and AT submodel bytes is that the XT indicates that protected mode is not supported, and there is no 8042 keyboard controller. The AT indicates that protected mode is available, and an 8042 keyboard controller exists.

Values:

BIOS_SUBMODEL_AT - IBM AT compatible (recommended for 286 and above).
BIOS_SUBMODEL_XT - IBM XT compatible (recommended for 186 and below).

Related Parameters:

BIOS_HDWR - Model byte.

7.2.135 DEBUG_CMDBUF_LEN Parameter

The **DEBUG_CMDBUF_LEN** parameter specifies the size of the type-in buffer used by the debugger when accepting commands from the keyboard.

This value is typically 128 bytes and can be reduced if more memory is needed from the 1KB Extended BIOS Data Area.

Values:

n - Number of bytes to reserve for the debugger's command input buffer.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable debugger support.

7.2.136 DEBUG_MAX_BREAKPOINTS Parameter

The **DEBUG_MAX_BREAKPOINTS** parameter specifies the number of simultaneous breakpoints the debugger can manage at any given time.

Each breakpoint requires space from the Extended BIOS Data Area to support.

Values:

n - Number of simultaneously-defined breakpoints supported by the debugger.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable debugger support.
DEBUG_MAX_BKPT_CMD_LEN - Size of command buffer/each record.

7.2.137 DEBUG_MAX_BKPT_CMD_LEN Parameter

The **DEBUG_MAX_BKPT_CMD_LEN** parameter specifies size of the buffer reserved in each debugger breakpoint record for an optional ASCII command string to be executed at the time the breakpoint occurs.

Each breakpoint requires space from the Extended BIOS Data Area to support, and increasing the command buffer for breakpoints negatively impacts the available space in the EBDA.

Values:

n - Number of simultaneously-defined breakpoints supported by the debugger.

Related Parameters:

OPTION_SUPPORT_DEBUGGER - Enable debugger support.
DEBUG_MAX_BREAKPOINTS - Number of breakpoint records to support.

7.2.138 CONFIG_WINCE_ENTRY Parameter

The **CONFIG_WINCE_ENTRY** parameter specifies the physical address of the start of the Windows CE ROM image to be loaded by the **BOOT_WINCE** boot action.

Values:

0xxxxxxxxh - 32-bit Media Address specifing start of Windows CE ROM image.

Related Parameters:

OPTION_SUPPORT_WINCE - Enable Windows CE support.
CONFIG_WINCE_VIDEO - Specify initial video mode for Windows CE.
CONFIG_WINCE_PORT - Specify COM port for Windows CE kernel uploads.
CONFIG_WINCE_BAUD - Specify COM port baud rate for kernel uploads.
CONFIG_WINCE_PCI - Specify PCI configuration method for Windows CE kernel.

7.2.139 CONFIG_WINCE_VIDEO Parameter

The **CONFIG_WINCE_VIDEO** parameter specifies the initial Windows CE-defined video mode to be selected by the BIOS before transferring control to the Windows CE kernel.

Values:

0 - Use 320 x 200 x 256 mode.
n - Other modes as defined by Microsoft.

Related Parameters:

OPTION_SUPPORT_WINCE - Enable Windows CE support.
CONFIG_WINCE_ENTRY - Specify location of Windows CE ROM image.
CONFIG_WINCE_PORT - Specify COM port for Windows CE kernel uploads.
CONFIG_WINCE_BAUD - Specify COM port baud rate for kernel uploads.
CONFIG_WINCE_PCI - Specify PCI configuration method for Windows CE kernel.

7.2.140 CONFIG_WINCE_PORT Parameter

The **CONFIG_WINCE_PORT** parameter specifies the COM port used by the Windows CE kernel to communicate with the host PC during development.

Values:

0 - no COM port.
1 - COM1.
2 - COM2.
3 - COM3.
4 - COM4.

Related Parameters:

OPTION_SUPPORT_WINCE - Enable Windows CE support.
CONFIG_WINCE_ENTRY - Specify location of Windows CE ROM image.
CONFIG_WINCE_VIDEO - Specify initial video mode for Windows CE kernel.
CONFIG_WINCE_BAUD - Specify COM port baud rate for kernel uploads.
CONFIG_WINCE_PCI - Specify PCI configuration method for Windows CE kernel.

7.2.141 CONFIG_WINCE_BAUD Parameter

The **CONFIG_WINCE_BAUD** parameter specifies the baud rate that the Windows CE kernel should use to communicate with the host PC during development.

Values:

COM_BAUD_110 - 110 baud.
COM_BAUD_150 - 150 baud.
COM_BAUD_300 - 300 baud.
COM_BAUD_600 - 600 baud.
COM_BAUD_1200 - 1200 baud.
COM_BAUD_2400 - 2400 baud.
COM_BAUD_4800 - 4800 baud.
COM_BAUD_9600 - 9600 baud.
COM_BAUD_19K - 19.2K baud.
COM_BAUD_28K - 28.8K baud.
COM_BAUD_38K - 38.4K baud.
COM_BAUD_56K - 56K baud.
COM_BAUD_115K - 115K baud.

Related Parameters:

OPTION_SUPPORT_WINCE - Enable Windows CE support.
CONFIG_WINCE_ENTRY - Specify location of Windows CE ROM image.
CONFIG_WINCE_VIDEO - Specify initial video mode for Windows CE kernel.
CONFIG_WINCE_PORT - Specify COM port for kernel uploads.
CONFIG_WINCE_PCI - Specify PCI configuration method for Windows CE kernel.

7.2.142 CONFIG_WINCE_PCI Parameter

The **CONFIG_WINCE_PCI** parameter specifies the method that the Windows CE kernel will use to configure PCI. The details of this parameter are beyond the scope of this Adaptation Kit. Refer to your Windows CE ETK for more information.

Values:

See Windows CE ETK.

Related Parameters:

OPTION_SUPPORT_WINCE - Enable Windows CE support.
CONFIG_WINCE_ENTRY - Specify location of Windows CE ROM image.
CONFIG_WINCE_VIDEO - Specify initial video mode for Windows CE kernel.
CONFIG_WINCE_PORT - Specify COM port for kernel uploads.
CONFIG_WINCE_BAUD - Specify baud rate for COM port for Windows CE kernel.

7.2.143 CONFIG_CFGBOX_MONO_ATTRIB Parameter

The **CONFIG_CFGBOX_MONO_ATTRIB** parameter specifies the hexadecimal value to be used as an attribute byte for POST's configuration box, when using monochrome display adapters.

Values:

See IBM PC documentation (default is 0fh, white on black).

Related Parameters:

OPTION_SUPPORT_CONFIGBOX - Enable configuration box support.
CONFIG_CFGBOX_COLOR_ATTRIB - Specify attribute for color display adapters.

7.2.144 CONFIG_CFGBOX_COLOR_ATTRIB Parameter

The **CONFIG_CFGBOX_COLOR_ATTRIB** parameter specifies the hexadecimal value to be used as an attribute byte for POST's configuration box, when using color display adapters.

Values:

See IBM PC documentation (default is 1eh, yellow on blue).

Related Parameters:

OPTION_SUPPORT_CONFIGBOX - Enable configuration box support.
CONFIG_CFGBOX_MONO_ATTRIB - Specify attribute for mono display adapters.

7.2.145 CONFIG_DELAY_ADD Parameter

The **CONFIG_DELAY_ADD** parameter specifies an additive value to be used by the DELAY macro within the core BIOS code to increase the number of "JMP $+2" instructions generated by this macro.

Values:

n – Specifies number of extra "JMP $+2" instructions to use in DELAY macro.

Related Parameters:

CONFIG_DELAY_MULTIPLY – Scales count of instructions by a factor.

7.2.146 CONFIG_DELAY_MULTIPLY Parameter

The **CONFIG_DELAY_MULTIPLY** parameter specifies a multiplicative value to be used by the DELAY macro within the core BIOS code to scale the number of "JMP $+2" instructions generated by this macro.

Values:

n – Specifies a scaling factor of extra "JMP $+2" instructions to use in DELAY macro.

Related Parameters:

CONFIG_DELAY_ADD – Specify incremental number of instructions for DELAY macro.

7.2.147 CONFIG_DELAY_IO Parameter

The **CONFIG_DELAY_IO** parameter specifies an I/O port used by the DELAY_IO macro within the core BIOS to slow-down CPU loops to 8Mhz. By default, this value is 80h (a port usually implemented in ISA logic that corresponds to the progress codes issued by POST). However, if port 80h is not available, another port can be chosen.

Values:

n – Specifies the I/O address of an 8-bit port that can be used for dummy reads and writes to slow down CPU loops. If set to 0, no I/Os will be used.

Related Parameters:

None.

7.2.148 CONFIG_SPLASH_VMODE Parameter

The **CONFIG_SPLASH_VMODE** parameter specifies the encoded command code and mode parameter to be passed to INT 10h in the AX CPU register by the Splash Screen Manager when setting the video mode. The default value is 0012h, which is the value for a Set Mode command for 640x480x16 color standard VGA mode.

If you have a custom VGA BIOS or you have implemented your own graphics interface,you could set this parameter to reflect your custom INT 10h interface. Since the standard "Set Video Mode" function is AH=00h, and because the AL register contains the video mode for this function, all the values below are in the range 0000h-00ffh. In theory, the OEM could declare an alternate function code for INT 10h, and specify it by selecting values higher than 00ffh.

Other parameters relating to the splash screen configuration must be properly configured in order for the graphics system to work properly in the specified mode (see Related Parameters). When selecting modes, be careful to follow these guidelines:

- Interlaced modes are not supported
- Only monochrome, 16 color, and 256 color modes are supported
- The window to display memory cannot be larger than 64k bytes (65,536)
- VESA modes are not supported

Values:

0dh – Specifies 320x200, 16 colors.
0eh – Specifies 640x200, 16 colors.
0fh – Specifies 640x350, 2 colors (black and white).

10h – Specifies 640x350, 16 colors.
11h – Specifies 640x480, 2 colors (black and white).
12h – Specifies 640x480, 16 colors.
13h – Specifies 320x200, 256 colors.

Related Parameters:

OPTION_SUPPORT_SPLASHSCR - Enable splash screen support.
OPTION_SUPPORT_EXTRES - Enable External Resource Manager support.
CONFIG_SPLASH_WIDTH – Specify display device width in pixels.
CONFIG_SPLASH_WBYTES – Specify video frame buffer width in bytes.
CONFIG_SPLASH_HEIGHT – Specify video display height in raster lines.
CONFIG_SPLASH_COLORS – Specify number of colors supported by video mode.
CONFIG_SPLASH_SEG – Specify segment address for graphics workspace.
CONFIG_SPLASH_BOOTS – Specify limit for booting with disabled splash screen.
SPLASH_TABLE – Specify graphic resources to be used.

7.2.149 CONFIG_SPLASH_WBYTES Parameter

The **CONFIG_SPLASH_WBYTES** parameter is used by the low-level drawing routines to calculate the starting position in display memory for each raster line.

For 16 color modes that are only 320 pixels wide, the correct value for this parameter would be 40. For all of the 640xX 16 color and monochrome modes, the correct value for this parameter would be 80. For the 320x200x256 color mode, the correct value is 320.

This value must reflect the actual way that the display controller has arranged memory, and does not reflect how many pixels are actually displayed either by the controller or by the monitor or the LCD panel (the graphics controller may have special hardware registers that control the number of pixels displayed, regardless of the internal geometry of the display memory.

Values:

40 – Mode 0dh.
80 – Modes 0eh-12h.
320 – Mode 13h.

Related Parameters:

OPTION_SUPPORT_SPLASHSCR - Enable splash screen support.
OPTION_SUPPORT_EXTRES - Enable External Resource Manager support.
CONFIG_SPLASH_VMODE – Specify video mode for graphical front-end.
CONFIG_SPLASH_WIDTH – Specify display device width in pixels.
CONFIG_SPLASH_HEIGHT – Specify video display height in raster lines.
CONFIG_SPLASH_COLORS – Specify number of colors supported by video mode.
CONFIG_SPLASH_SEG – Specify segment address for graphics workspace.
CONFIG_SPLASH_BOOTS – Specify limit for booting with disabled splash screen.
SPLASH_TABLE – Specify graphic resources to be used.

7.2.150 CONFIG_SPLASH_HEIGHT Parameter

The **CONFIG_SPLASH_HEIGHT** parameter is used by the splash screen display routines as the definition for how many raster lines there are for the selected video mode. This height is used in turn by the proportional graphics engine to position graphics on the screen.

Values:

200 – Modes 0dh-0eh.
350 – Modes 0fh-10h.
480 – Modes 11h-13h.

Related Parameters:

OPTION_SUPPORT_SPLASHSCR - Enable splash screen support.
OPTION_SUPPORT_EXTRES - Enable External Resource Manager support.
CONFIG_SPLASH_VMODE – Specify video mode for graphical front-end.
CONFIG_SPLASH_WIDTH – Specify display device width in pixels.
CONFIG_SPLASH_WBYTES – Specify video frame buffer width in bytes.
CONFIG_SPLASH_COLORS – Specify number of colors supported by video mode.
CONFIG_SPLASH_SEG – Specify segment address for graphics workspace.
CONFIG_SPLASH_BOOTS – Specify limit for booting with disabled splash screen.
SPLASH_TABLE – Specify graphic resources to be used.

7.2.151 CONFIG_SPLASH_COLORS Parameter

The **CONFIG_SPLASH_COLORS** parameter is used by the splash screen display routines to determine how many colors are supported in the selectd video mode. It is used both on a low level, as a means of determining the number of bit planes in use, and on a higher level to determine how much memory to reserve for palette selection.

In EMBEDDED BIOS 4.3, only palette based and monochrome modes are supported. Four-color CGA modes are not supported and true color modes are not supported.

Values:

2 – Monochrome (2 color) modes.
16 – 16 color modes.
256 – 256 color modes.

Related Parameters:

OPTION_SUPPORT_SPLASHSCR - Enable splash screen support.
OPTION_SUPPORT_EXTRES - Enable External Resource Manager support.
CONFIG_SPLASH_VMODE – Specify video mode for graphical front-end.
CONFIG_SPLASH_WIDTH – Specify display device width in pixels.
CONFIG_SPLASH_WBYTES – Specify video frame buffer width in bytes.
CONFIG_SPLASH_HEIGHT – Specify video display height in raster lines.
CONFIG_SPLASH_SEG – Specify segment address for graphics workspace.
CONFIG_SPLASH_BOOTS – Specify limit for booting with disabled splash screen.
SPLASH_TABLE – Specify graphic resources to be used.

7.2.152 CONFIG_SPLASH_SEG Parameter

The **CONFIG_SPLASH_SEG** parameter is used by the splash screen display routines during POST to specify the segment address of an area of memory below 1MB that will be used as scratch space for buffering graphics resources.

The amount of memory that will be needed by the splash screen system may vary depending on the size of the splash screens involved, so users shoud take the size of the RLE graphics they will be extracting into account when determining the starting location of this temporary memory.

The default value of 7000h should provide enough space to extract RLE graphic files up to 128KB in size. Unless you have limited memory in your system, you should not need to modify this value. If **CONFIG_MAX_LOW_MEMORY** is set abnormally below 640, then you should change this parameter.

Values:

nnnnh – Hexadecimal segment address of the scratch segment.

Related Parameters:

OPTION_SUPPORT_SPLASHSCR - Enable splash screen support.
OPTION_SUPPORT_EXTRES - Enable External Resource Manager support.
CONFIG_MAX_LOW_MEMORY – Maximum low memory supported.
CONFIG_SPLASH_VMODE – Specify video mode for graphical front-end.
CONFIG_SPLASH_WIDTH – Specify display device width in pixels.
CONFIG_SPLASH_WBYTES – Specify video frame buffer width in bytes.
CONFIG_SPLASH_HEIGHT – Specify video display height in raster lines.
CONFIG_SPLASH_COLORS – Specify number of colors supported by video mode.
CONFIG_SPLASH_BOOTS – Specify limit for booting with disabled splash screen.
SPLASH_TABLE – Specify graphic resources to be used.

7.2.153 CONFIG_SPLASH_BOOTS Parameter

The **CONFIG_SPLASH_BOOTS** parameter is used by POST to manage the number of consecutive boots for which the splash screen may be disabled with a Setup Screen setting. After the specified number of boots, the splash screen is automatically reenabled.

This feature is used for evaluation platforms containing a preinstalled evaluation copy of EMBEDDED BIOS. Although it is possible for the user of an evaluation platform to disable the graphical display, it cannot be disabled indefinitely without an explicit setting of this parameter in the build.

Values:

0 – The splash screen cannot be disabled in Setup.
1-127 – Number of consecutive boots for which the splash screen may be disabled.
128 – The splash screen will not be automatically enabled.

Related Parameters:

OPTION_SUPPORT_SPLASHSCR - Enable splash screen support.
OPTION_SUPPORT_EXTRES - Enable External Resource Manager support.

CONFIG_MAX_LOW_MEMORY – Maximum low memory supported.
CONFIG_SPLASH_VMODE – Specify video mode for graphical front-end.
CONFIG_SPLASH_WIDTH – Specify display device width in pixels.
CONFIG_SPLASH_WBYTES – Specify video frame buffer width in bytes.
CONFIG_SPLASH_HEIGHT – Specify video display height in raster lines.
CONFIG_SPLASH_COLORS – Specify number of colors supported by video mode.
CONFIG_SPLASH_SEG – Specify segment address for graphics workspace.
SPLASH_TABLE – Specify graphic resources to be used.

7.2.154 SPLASH_TABLE Table

The **SPLASH_TABLE** macro is used to define the graphics used by the graphical POST system, commonly referred to as the Splash Screen feature. This table describes a time-ordered sequence of graphics to be displayed, and associates them with internal system events.

The graphical POST system uses the run-time services of the EMBEDDED BIOS External Resource Manager to retrieve graphics by their *Graphics Resource ID*, a 16-bit identifying number specified in the .IDF file read by GSMERGE when combining various components of the composite BIOS. This is the step where graphics files are associated with Resource IDs.

The splash table is specified in a tabular format with **SPLASH_TABLE** entries. Each line in the table specifies a new graphical component of the total graphical POST sequence, and begins with the identifying macro command, **SPLASH_TABLE**. Each line contains exactly four (4) operands, as in the following *hypothetical example*:

```
SPLASH_TABLE EVENT_SPLASH_INIT, RESOURCE_ID_SPLASH, SPLASH_CENTER, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_INIT, RESOURCE_ID_BOOTMSG, SPLASH_CENTER, 9800
SPLASH_TABLE EVENT_SPLASH_MSG1, RESOURCE_ID_ADVERT1, SPLASH_CENTER, SPLASH_TOP
SPLASH_TABLE EVENT_SPLASH_MSG2, RESOURCE_ID_ADVERT2, SPLASH_CENTER, SPLASH_TOP
SPLASH_TABLE EVENT_SPLASH_MSG3, RESOURCE_ID_ADVERT3, SPLASH_RIGHT, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_MSG3, RESOURCE_ID_ADVERT4, SPLASH_LEFT, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_MSG3, 0FF10h, SPLASH_LEFT, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_MSG3, RESOURCE_ID_ADVERT5, SPLASH_LEFT, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_ICON, SPLASH_ICON_RIGHT+32, 131, 9868 ; define progress bar
```

The first operand specifies an event code that is associated with the entry. Multiple entries may be associated with the same event code, in which case they are all activated when that event occurs in the system. (Note: only one instance of **EVENT_SPLASH_ICON** is permitted, however.) The following events are defined by the architecture:

EVENT_SPLASH_INIT (00h) – The initial call to the splash screen.
EVENT_SPLASH_ICON (01h) – Progress bar Icon location/ordering request.
RESERVED (02h-0fh) – Reserved for future expansion.
EVENT_SPLASH_MSG1 (10h) – First graphic after initial splash screen.
EVENT_SPLASH_MSG2 (11h) – Second graphic after initial splash screen.
EVENT_SPLASH_MSG3 (12h) – Third graphic after initial splash screen.
EVENT_SPLASH_MSG4 (13h) – Fourth graphic after initial splash screen.
EVENT_SPLASH_MSG5 (14h) – Fifth graphic after initial splash screen.
EVENT_SPLASH_MSG6 (15h) – Sixth graphic after initial splash screen.

The second operand specifies the graphic resource ID to be associated with the event. When the event occurs, all of the graphics defined in the **SPLASH_TABLE** with a matching event code are displayed in the order they occur. If this parameter is specified to be a value of the form

0ffxxh, then the system will delay for *xxh* 55ms timer ticks instead of displaying a graphic. This permits a timed delay between successive graphics associated with the same event, so that the user gets a chance to see them. An example of this is shown in the table above, where two different graphics images (**RESOURCE_ID_ADVERT4** and **RESOURCE_ID_ADVERT5**) are displayed in response to **EVENT_SPLASH_MSG3**, but separated by a delay of 10h timer ticks (approximately 16*55ms=880ms, or about 9/10 of a second).

The third operand specifies the horizontal component of the location at which the associated graphic will be drawn. If **SPLASH_CENTER** (5000) is specified, the graphic will be centered on the screen. If **SPLASH_LEFT** (0) is specified, the graphic will be left-justified. If **SPLASH_RIGHT** (10000) is specified, the graphic will be right-justified. Any other value will be used as a virtual position within this framework (think of 5000 as 50.00 percent of the screen, 0 as 0.00 percent of the screen, and 10000 as 100.00 percent of the screen, so that a new number like 7500 would represent 3/4th of the way across the screen.

The fourth operand specifies the vertical component of the location at which the associated graphic will be drawn. If **SPLASH_CENTER** (5000) is specified, the graphic will be centered between the top and bottom of the screen. If **SPLASH_TOP** (0) is specified, the graphic will be displayed at the top of the screen. If **SPLASH_BOTTOM** (10000) is specified, the graphic wil be displayed at the bottom of the screen. Any other value will be used as a virtual position within this framework (see the description for the 3rd operand, above).

For the **EVENT_SPLASH_ICON** table entry, the second, third, and fourth operands have different meanings. This table entry, when specified, includes the graphical progress bar as a visible component of the graphical POST system. The second parameter defines both the X and Y travel directions and a scalar displacement, used in each direction specified, of each icon with respect to its predecessor. The third parameter specifies the starting X location in the range 0-10000, and the fourth parameter specifies the starting Y location in the range 0-10000, of the first icon in the graphical progress bar.

In the above example, some entries share the same event in the table. When those events are triggered, more than one graphic is drawn. This feature provides the ability to perform animation by drawing successive graphics, not necessarily in the same location.

A different feature is the ability to use the same graphic ID in different entries in the table. This allows reuse of the graphic for different situations, saving the need to duplicate the graphic physically in the build.

Related Parameters:

OPTION_SUPPORT_SPLASHSCR - Enable splash screen support.
OPTION_SUPPORT_EXTRES - Enable External Resource Manager support.
CONFIG_MAX_LOW_MEMORY – Maximum low memory supported.
CONFIG_SPLASH_VMODE – Specify video mode for graphical front-end.
CONFIG_SPLASH_WIDTH – Specify display device width in pixels.
CONFIG_SPLASH_WBYTES – Specify video frame buffer width in bytes.
CONFIG_SPLASH_HEIGHT – Specify video display height in raster lines.
CONFIG_SPLASH_COLORS – Specify number of colors supported by video mode.
CONFIG_SPLASH_SEG – Specify segment address for graphics workspace.
CONFIG_SPLASH_BOOTS – Specify limit for booting with disabled splash screen.

7.2.155 POWER_DEVID (Power Management) Table

The **POWER_DEVID** macro is used to define a tree of device dependencies for the power manager (see Chapter 15 for further information about the power manager). Always at the top of the tree is the CPU itself. The CPU is the parent of all 1st-tier devices underneath it, such as Super I/O controllers, Flash arrays, PCMCIA controllers, and the like. Similarly, 1st-tier devices become parents of the devices they control, such as IDE drives and UARTs in the case of Super I/O controllers, PCMCIA cards in the case of PCMCIA controllers, and so on. EMBEDDED BIOS has a limit of eight (8) levels in its power management tree, which is more than adequate for anticipated designs.

The power management device tree is specified in a tabular format with **POWER_DEVID** entries. Each line in the table specifies a new device that will be participating in the system's power management, and begins with the identifying macro command, **POWER_DEVID**. Each line contains exactly four (4) operands, as in the following *hypothetical example*:

```
;       Power management device tree definition:
;       The POWER_DEVID entry for the CPU MUST be FIRST!
;
;               Device  Module: Parent: Setup text:
;
  POWER_DEVID   CPU,    Board,  CPU,    "Cpu"
  POWER_DEVID   IDE_0,  Ide,    SuperIo,"IDE drive 0"
  POWER_DEVID   IDE_1,  Ide,    SuperIo,"IDE drive 1"
  POWER_DEVID   SUPERIO,Board,  CPU,    "Super I/O"
  POWER_DEVID   PCMCIA, Board,  CPU,    "PCMCIA"
```

The first operand specifies the symbolic name of the participating device. These device names must have legal MASM or TASM symbol syntax, and should really be short names to keep the table simple. These symbols are case-sensitive, and are referred-to by the parent field in other entries of the table.

The second operand specifies the software component, usually a module name, that is responsible for management of the device. This operand is prepended to the string `PwrLvl` to produce a final name of a procedure in the BIOS that is responsible for managing the device's power level. This routine (see Chapter 15 for calling conventions) is called by the core BIOS's power management system at the appropriate time to instruct the module to change the device's power state. Because this operand specifies a name, and not an ordinal, it is possible to add OEM-defined device types to the system. General Software has provided the following types in the core BIOS:

`Board`	OEM Board Personality Module
`Ide`	IDE Hard Drives
`Media`	Media Control Layer (All RFD Devices)
`MtdRam`	RAM MTD
`MtdRom`	ROM MTD
`MtdAmd8_1`	AMD Flash 8-Bit 1-Way MTD
`MtdAmd8_2`	AMD Flash 8-Bit 2-Way MTD
`MtdAmd8_4`	AMD Flash 8-Bit 4-Way MTD
`MtdAmd16_1`	AMD Flash 16-Bit 1-Way MTD
`MtdAtm8_1`	Atmel Flash 8-Bit 1-Way MTD
`MtdBulk_1`	Bulk Erase Flash 8-Bit 1-Way MTD
`MtdInt16_1`	Intel Flash 16-Bit 1-Way MTD
`MtdInt16_2`	Intel Flash 16-Bit 2-Way MTD

`MtdInt8_1`	Intel Flash 8-Bit 1-Way MTD
`MtdInt8_2`	Intel Flash 8-Bit 2-Way MTD
`MtdInt8_A`	Intel Flash 28F016 in 8-Bit mode (1-Way) MTD
`MtdInt8_B`	Intel Boot Block Flash MTD
`MtdInt16A_1`	Intel 28F016/28F032 MTD

Note that the above list includes some "real" devices, and some "pseudo" devices. For example, the Flash parts managed by the MtdInt16_1 Media Technology Driver are very real. The IDE drives managed by the Ide module are also real. The Board module corresponds to real hardware if the OEM chooses to write the **BoardPwrLvl** routine to handle power management requests, and that routine must do whatever makes sense to manage the board's "power". Pseudo devices such as the one called Media are actually place-holders. It is necessary to allow these intermediate devices (this one routes Flash requests to the underlying MTDs), to play a role in managing power, so that they receive notification that they should suspend or resume the processing of their client's requests if the power is suspended or resumed, respectively.

In later versions of EMBEDDED BIOS, additional power management devices may be provided. For information about the current list of supported power management devices, contact General Software.

The third operand specifies the device name of the device's parent. For IDE drives, this is typically a Super I/O controller (the OEM would need to define the appropriate **SuperIoPwrLvl** routine in the Board Personality Module that is responsible for managing the Super I/O controller in the system). For UARTs, this might also be a Super I/O controller.

The fourth operand specifies an ASCII string in quotes that will appear in the power management SETUP screens so that the user can configure timeouts and enable and disable power management on a device basis. These strings should be kept simple and short, so as to fit within the space constraints of the SETUP screen system and also to be clear to the user.

Related Parameters:

OPTION_SUPPORT_POWERMAN - Enable power management support.

7.2.156 MEDIA_REGION (Media Management) Table

The EMBEDDED BIOS Media Control Layer MCL provides a centralized, uniform, access to all Flash and related storage devices in the system for its clients, which include the Resident Flash Disk, the Debugger's Flash commands, and Manufacturing Mode. Whereas traditional Flash file systems only support a single device type in a system, the MCL supports many types of media in the same system, and handles dispatching to associated Media Technology Drivers (MTDs) transparently to its clients.

This functionality of the MCL is largely data-driven, based on a table created with the **MEDIA_REGION** macro in the project file.

The **MEDIA_REGION** macro is used to define the system's address space for the purpose of routing Flash I/O requests associated with a specific 32-bit address to the correct MTD. The MCL scans the table, starting at the first specified record, until it finds a record that contains the media address of interest, or until it reaches the end of the table, to determine the MTD that will handle the request.

The media table is specified in a tabular format with **MEDIA_REGION** entries. Each line in the table specifies a new address range that maps to a particular MTD. In the event that address ranges overlap, then the MCL will find the first region that contains a given media address. Each line contains exactly three (3) operands, as in the following *hypothetical example*:

```
;               Starting   Ending      Technology
;               Phys Addr  Phys Addr   Driver Name
;               ---------  ----------  -----------
  MEDIA_REGION  00000000h, 0000dffffh, Ram
  MEDIA_REGION  000e0000h, 0000fffffh, Bulk8_1
  MEDIA_REGION  00100000h, 0007fffffh, Ram
  MEDIA_REGION  00800000h, 009ffffffh, Amd8_2
  MEDIA_REGION  00a00000h, 003ffffffh, Bulk8_1
```

The first operand specifies the 32-bit address associated with the first byte in the region to be defined. Note that on some processors, such as the AMD SC400 series, this address is not necessarily a bus address, but might correspond to the address configured for one of the processor's external chip select lines (ROMSEL0, for example). In this case, the Chipset Personality Module or Board Personality Module must make a decision during system initialization about where in the address space the chip select shall respond, and then the **MEDIA_REGION** entry for the device attached to the chip select must be coded properly so that the same addressing scheme is used.

The second operand specifies the 32-bit address associated with the last byte in the region to be defined. This address must be equal to or greater than the starting address as specified by the first operand.

The third operand specifies the name of the Media Technology Driver to which the MCL will route any requests related to this defined region. This operand is prepended with the string `MtdSvc` to produce a final name of a procedure in the BIOS that is responsible for handling I/O requests for the media type. This routine (see Chapter 13 for calling conventions) is called by the core BIOS's MCL, and never by any other software in the system.

Because the third operand specifies the name of the MTD, and not an ordinal, it is possible to add OEM-defined media types to the system. General Software has provided the following types in the core BIOS:

`Ram`	Read/Write RAM (SRAM & DRAM)
`Rom`	Read-Only (any read-only ROM, Flash, SRAM or DRAM)
`Amd8_1`	AMD Flash, 8-bit devices, 1-way interleaved
`Amd8_2`	AMD Flash, 8-bit devices, 2-way interleaved
`Amd8_4`	AMD Flash, 8-bit devices, 4-way interleaved
`Amd16_1`	AMD Flash, 16-bit devices, 1-way interleaved
`Atm8_1`	Atmel Flash, 8-bit devices, 1-way interleaved
`Bulk_1`	Bulk Erase Flash, 8-bit devices, 1-way interleaved
`Int16_1`	Intel Flash, 16-bit devices, 1-way interleaved
`Int16_2`	Intel Flash, 16-bit devices, 2-way interleaved
`Int8_1`	Intel Flash, 8-bit devices, 1-way interleaved
`Int8_2`	Intel Flash, 8-bit devices, 2-way interleaved
`Int8_A`	Intel 28F016 Flash, 8-bit mode, 1-way interleaved
`Int8_B`	Intel Boot Block Flash, 8-bit mode, 1-way interleaved
`Int16A_1`	Intel Flash, 28F016/28F032 devices, 1-way interleaved
`Int16A_2`	Intel Flash, 28F016/28F032 devices, 2-way interleaved
`Nand8_1`	AMD/Toshiba NAND Flash, 8-bit, 1-way interleaved

It is very important that the correct Flash driver be used for a given Flash array. Flash arrays are characterized by the device technology (AMD, Intel, Bulk, RAM, etc.), their data path width, and their interleave factor.

The first part of any MTD's name is its device technology. This makes it easy to determine which parts (AMD, Intel, or whatever) are being used in a given region.

The second part of the MTD's name is the data path width. This is determined by the Flash parts themselves, and how they are configured with strapping pins. For example, the Intel 28F008 is an 8-bit Flash part because it has an 8-bit data bus (D0-D7). The Intel 28F016 can be either an 8-bit or a 16-bit part, depending on how a package pin is strapped in hardware.

The third part of the MTD's name is the interleave factor. This is determined by the number of Flash parts ganged together, so as to widen the data path. For example, two Intel 28F008 parts can be ganged together, so that D0-D7 of the first part form the top 8 bits of a 16-bit data word, and D0-D7 of the second part form the bottom 8 bits of the same word. This technique is called interleaving, and in this example, the interleave factor is two (2). When a Flash array is not composed of ganged parts in this manner, the interleave factor is said to be one (1).

Note that in the event that the MCL cannot find an entry in the table that contains a given media address, then the request is passed to the RAM MTD, which treats the address space as though it had Random Access Memory (RAM), capable of being byte-addressable, with read and write operations available. Erase operations are simulated in the RAM MTD by resetting each byte within a block of **CONFIG_RFDDISK_KBBLKSIZE** to the value, FFh.

In later versions of EMBEDDED BIOS, additional MTDs may be provided. For information about the current list of supported MTDs, contact General Software.

The OEM can add additional MTDs to support special media by adding the appropriately named MTD entrypoint routine in the Board Personality Module; MTDs need not be implemented in separate assembly modules or in the SYSTEM (core BIOS) directory.

Related Parameters:

OPTION_SUPPORT_MCL - Enable Flash (and MCL) support.

7.2.157 FILE_SYSTEM (INT 13h Drive Management) Table

The EMBEDDED BIOS File System Control Layer (FSCL) provides a centralized, uniform, access to all INT 13h mass storage devices in the system for its clients, which include the operating system, application software, and Manufacturing Mode.

Hereafter, the term file system will be used to mean a disk driver or its emulator. The term "file system driver" (FSD) will be used to mean the code that receives I/O requests from FSCL to either manage the device or emulate it.

The FSCL architecture provides a way for file system drivers (FSDs), including those supporting floppy disks, IDE drives, ROM disks, RAM disks, Flash disks, and OEM-defined drivers, to participate in the system in a cooperative way. File systems can be mapped to specific BIOS unit numbers by the OEM using the SETUP screen system, transparently to the drivers themselves.

FSCL initializes each participating file system during POST, and routes INT 13h I/O requests to the appropriate FSD, based on this BIOS unit mapping.

The architecture provides for each file system to provide access to multiple devices in the same class within the same system. This allows support for up to four real physical floppy drives, four real physical IDE drives, and a virtually unlimited number of ROM, RAM, and Flash disks.

The architecture also permits FSDs to support both soft-style (floppy format) and hard-style (hard disk partitioned) file system layouts. The purpose of this feature is to provide the OEM with a choice of floppy-format or partitioned ROM, RAM, and Flash disks, although the idea can be logically extended to treating real IDE drives as floppy units, and real floppy drives as partitioned media, all transparently to the operating system.

This functionality of the FSCL is largely data-driven, based on a table created with the **FILE_SYSTEM** macro in the project file.

The **FILE_SYSTEM** macro is used to define the specific file systems that will be supported in the system. As previously mentioned, a given FSD may support multiple file systems. These file systems, as defined by the **FILE_SYSTEM** macro, are then mapped to drives in the SETUP screen, according to the user's needs. Not all of the entries in the **FILE_SYSTEM** table need be selected by the user. Only those enabled will actually be initialized by FSCL. The **FILE_SYSTEM** table entries represent the possible file systems that the BIOS will support.

When FSCL receives INT 13h requests for a specific drive, they are routed to the FSD that is handling the file system for the drive. The dispatching mechanism indexes into the **FILE_SYSTEM** table to locate the FSD associated with the file system itself.

The file system table is specified in a tabular format with **FILE_SYSTEM** entries. Each line in the table specifies a new file system that is governed by a particular FSD. Each line contains exactly five (5) operands, as in the following *hypothetical example*:

```
;             Type  Device    Start Addr  Length      SETUP name (unique)
;             ----  ------    ----------  ----------  -------------------
FILE_SYSTEM   Soft, Floppy,           0h,         0h, "Floppy 0"
FILE_SYSTEM   Soft, Flash,    080000000h,    400000h, "4MB Flash Disk 0"
FILE_SYSTEM   Hard, Ide,              0h,         0h, "IDE Drive 0"
FILE_SYSTEM   Hard, Ide,              1h,         0h, "IDE Drive 1"
```

The first operand specifies the type of file system (soft or hard). Soft file systems are configured by the BIOS to respond as floppies to the operating system; that is, they are associated with unit numbers in the range 00h-7fh (bit 7 clear). Hard file systems are configured by the BIOS to respond as hard disks to the operating system; that is, they are associated with unit numbers in the range 80h-ffh (bit 7 set). Soft file systems are never partitioned, whereas hard file systems are always partitioned.

The second operand specifies the file system driver (FSD) to be associated with the file system. There is a set of standard FSDs provided in the core BIOS, and the OEM can add new FSDs if needed. The following is a list of built-in file systems supported by the core BIOS:

Floppy	True floppy disk drives (360K, 1.2M, 720K, 1.44M, 2.88M)
Ide	IDE hard drives and relatives (ATA cards as well)
Cdrom	CD-ROM drives with bootable "El Torito" CD-ROM media
Rom	ROM disk driver (read-only, sectors direct-mapped to memory)
Ram	RAM disk driver (read/write, sectors direct-mapped to memory)

Flash	Flash disk driver (read/write, sectors movable in memory)
Doc2000	Disk-On-Chip 2000 file system driver placeholder for bootability
User	User-defined file system driver in Board Personality Module

OEM-defined file systems may be added in the system by assigning them a unique name (say, User), adding an entry in the **FILE_SYSTEM** table with that name, and then naming the entrypoint of the new file system according to the naming conventions described in the chapter on File System Drivers.

The third operand identifies the location of the underlying media for the file system, to the FSD. For FSDs that emulate drives with memory (ROM, RAM, or Flash disks), the starting media address of the memory array is specified here. This is illustrated in the example above with the entry for a Flash file system called "4MB Flash Disk 0", which starts at media address 80000000h.

For FSDs that need to identify physical equipment, this field may be divided into several bitfields. For IDE drives, bit 0 indicates whether the physical drive is a master or slave device, and bit 1 indicates whether the controller I/O base address is 1f0h (0) or 170h (1). For Floppy drives, this field is simply the floppy drive unit number, from 0 to 3.

The fourth operand provides additional information about the file system to the FSD, and this information is FSD-specific. For FSDs that emulate drives with memory (ROM, RAM, or Flash disks), the size of the memory array is specified here in bytes. In the example above, the 4MB Flash Disk is assigned a length field of 400000h, or 4MB.

For FSDs that identify physical equipment like floppy disks and IDE drives, this field is not used.

The fifth operand is the human-readable name assigned to the file system, for purposes of display in the SETUP screens and in operator prompts (such as when the user is prompted to verify an RFD, for example). This name should not exceed 16 characters, or the SETUP screen may not be displayed correctly.

Related Parameters:

OPTION_SUPPORT_DISKIO - Enable FSCL support.

7.2.158 LOAD_IMAGE (Windows CE Bootability) Table

The EMBEDDED BIOS CE Ready software can load and launch a copy of Windows CE, or any other operating system software, as it is stored in a file on any disk that the BIOS recognizes. This includes floppy disks, IDE drives, ROM disks, RAM disks, and Flash disks.

When booting Windows CE with the CE Ready feature, the core BIOS scans a table built from **LOAD_IMAGE** entries in the OEM's project file. This table lists the different filenames that should be scanned for in the load attempt. The file also describes the contents of each file, so that the core BIOS knows where to load the image into RAM, and how to transfer control to it.

When the BIOS attempts to boot Windows CE from a disk, it scans this table from beginning to end, searching the disk's root directory for the named file. If found, it transfers control to it according to the specifications in that **LOAD_IMAGE** table entry.

The image table is specified in a tabular format with **LOAD_IMAGE** entries. Each line in the table specifies a possible image to be loaded. Each line contains exactly four (4) operands, as in the following *hypothetical example*:

```
;             Filename          Type   Load Addr   Entrypoint
;             ----------------  -----  ----------  ----------
LOAD_IMAGE    "NK.BIN",         WinCe,         0h,         0h
LOAD_IMAGE    "RAW.COM",        Raw,   000010000h, 000010100h
LOAD_IMAGE    "OS.IMG",         Raw,   000080000h,         0h
```

The first operand specifies the name of the file to be searched for in the boot drive's root directory. The BIOS attempts variations on this file, so that closely-matching filenames are also detected. For example, NK.BIN also matches NK1.BIN, NKXYZ.BIN, etc.

The second operand specifies the type of image to be loaded. This allows the BIOS to determine how to interpret the contents of the image itself. For example, the NK.BIN file resulting from the Microsoft Windows CE build has a certain format that EMBEDDED BIOS parses specifically for Windows CE. This type is called **WinCe**. Another type, **Raw**, specifies that the image is not to be interpreted in any way, but is a simple binary image that should be copied without changing it. Other types may be available at dates later than this publication; contact General Software for details.

The third operand specifies the starting physical address of the memory into which the image is to be copied. For Windows CE builds, the starting physical address is contained in the file, and so the dummy value 0h is used in the table. For **Raw** files, this value must be below 1MB.

The fourth operand specifies the entrypoint, or physical address to which the BIOS jumps, after loading the image into memory. Note that the entrypoint is not specified as an address relative to the load address, but is itself a physical address. For Windows CE builds, this is specified as 0h because the entrypoint is encoded in the NK.BIN file itself. For **Raw** images, if this value is zero (0h), then no jump will take place, as in the third example above. In this instance, the contents of the OS.IMG file is loaded into low memory at physical address 80000h, and then no jump takes place. If the entrypoint is specified as a nonzero value, then the value is used as a jump address.

Related Parameters:

OPTION_SUPPORT_WINCE - Enable Windows CE loader support.
OPTION_WINCE_ENTRY - Windows CE entrypoint for Windows CE in ROM.
OPTION_WINCE_VIDEO - Windows CE video mode.
OPTION_WINCE_PORT - Windows CE COM port.
OPTION_WINCE_BAUD - Windows CE COM port baud rate.
OPTION_WINCE_PCI - Windows CE PCI initialization method.

7.2.159 PCI_ROM Configuration Table

While EMBEDDED BIOS can automatically detect and map ROM images on PCI devices, some PCI devices have embedded ROM images that are not detectable by the standard PCI initialization sequence. The **PCI_ROM** configuration table provides a way for the OEM to specify embedded PCI ROM images that must be mapped by the BIOS that are not to be autodetected.

Warning: Do not use this table to predefine PCI devices in a normal PCI system. The standard option ROMs on PCI devices are automatically detected by the core BIOS. This table is only used to specify option ROMs that do not show up in PCI bus autodetection.

The PCI devices with associated embedded ROM images to be mapped are specified in a tabular format with **PCI_ROM** macro entries. Each line in the table specifies a ROM image to be mapped for a particular function of a device on a bus. Each line contains exactly four operands to specify all of these things, as in the following *hypothetical example*:

```
;               Bus  Device  Function  Map Address
;               ---  ------  --------  -----------
PCI_ROM           0,    12h,      42h,      c0000h
```

The first operand specifies the number of the bus to which the device is attached. In this example, the bus number is 0. Bus number 0 is often used to mean the bus controller's address. You must specify the correct number for your system here.

The second operand specifies the number of the device on the bus. In this example, the device number is 12h.

The third operand specifies the function number of the particular device that is associated with the ROM image. The hypothetical function number in the example is 42h. Note that each PCI device may have one or more functional units in a system.

The fourth operand specifies the physical address where the option ROM should be mapped by the BIOS. In this example, physical address C0000h is specified, which translates to segment C000h, where the video BIOS extension is normally located.

Related Parameters:

OPTION_SUPPORT_PCI - Enable PCI support.

7.2.160 RELOCATE_FEATURE Configuration Table

The **RELOCATE_FEATURE** configuration table provides the way for OEMs to move specific components of the EMBEDDED BIOS core BIOS into segments other than F000h for builds greater than 64KB in size. Moving components into the other (lower) segments (E000h, D000h, or even C000h) makes room for other core BIOS features to be enabled. By providing a way for the OEM to selectively move components, maximum flexibility is passed along to the OEM.

This feature works closely in conjunction with the **OPTION_BIOS_KBSIZE** parameter. By default, this parameter is set to 64, providing an output file size of 64KB. To create system BIOSes with more features than can fit in the 64KB footprint, some of those features must be moved to a lower segment. When **OPTION_BIOS_KBSIZE** is set to a larger value (say, 128), the next lower segment (E000h) is opened up and made available for placement of core features. Code movement cannot be automatically handled by the build process because it is only at link time, not compile time, that the total size requirements of selected features can be determined, yet the segmentation for the selected features must be determined at compile time, not link time. The **RELOCATE_FEATURE** table provides the way to specify which components of the BIOS should be moved into lower segments, such as E000h.

The system BIOS can range from 16KB to 256KB in size. External considerations, such as the integration of a VGA BIOS into the final build, are beyond the scope of this section. Combination of external files not withstanding, the **OPTION_BIOS_KBSIZE** parameter is used to select the overall footprint size and automatically make segments at F000h, E000h, D000h, and C000h available as appropriate for the size. Then, the **RELOCATE_FEATURE** is used to move features into the segments made available by **OPTION_BIOS_KBSIZE**.

The following features may be moved away from the default (F000h) segment into other segments, thereby gaining space for other features. Others may become available since this manual was published; contact General Software for details.

Feature Name	Component	Approximate Codespace
DEBUG	BIOS Debugger	6KB-10KB, depending on options
SETUP	All Setup Screens	8KB-12KB, depending on options
SPLASH	All Splash Screen Code	4KB-10KB, depending on options

The features to be moved to other segments are specified in a tabular format with **RELOCATE_FEATURE** macro entries. Each line in the table specifies a feature to be moved. Each line contains exactly two operands to specify the feature name and its segment, as in the following *hypothetical example*:

```
;                  Feature           Location
;                  ----------------  -----------
RELOCATE_FEATURE   DEBUG,            SEG_E000      ; put debugger in E000h segment.
RELOCATE_FEATURE   SETUP,            SEG_D000      ; put setup system in D000h segment.
RELOCATE_FEATURE   SPLASH,           SEG_D000      ; put splash screen at D000h.
```

The first operand specifies the name of the feature to be moved. Not all features are movable; movement requires core BIOS support in terms of segmentation and the way calls are made into and out of a given feature's code modules. In many cases, the core BIOS features are compact enough that adding movability would add a substantial percentage to their size, reducing pay-off.

The second operand specifies the name of the segment to which the feature is to be moved. Legal segment names are **SEG_E000** (for values of **OPTION_BIOS_KBSIZE** between 65 and 128), **SEG_D000** (for values of **OPTION_BIOS_KBSIZE** between 129 and 192), and **SEG_C000** (for values of **OPTION_BIOS_KBSIZE** between 193 and 256).

Related Parameters:

OPTION_BIOS_KBSIZE – Specifies footprint size of BIOS in kilobytes.

Chapter 8

STEP-BY-STEP BIOS ADAPTATION

This chapter examines the issues related to adapting the standard BIOS to a specific platform. The following topics are covered in this chapter:

1. The project concept;
2. Selecting the best starting point;
3. Determining what needs to be changed;
4. Building a BIOS; and
5. Getting through POST.

8.1 The Project Concept

A BIOS engineer often needs to work on two or more BIOSes at the same time. For example, it is highly recommended that the first BIOS an engineer builds be for a standard evaluation board. Once that BIOS is working, which should be a fairly trivial task, the engineer would then begin development of a BIOS for the first iteration of the real hardware.

Usually this first iteration will be a non-form-factor bread-board design. If, for example, the end product was to be a cellular smart phone or PDA, then the final design will be packaged in a space with dimensions of not more than 1.5" by 3" by 0.5" and will quite likely use flexible multi-layer printed circuits. Such a package is extremely hard to work with from a troubleshooting perspective. It is also likely to be mechanically fragile. In the long run it is often cheaper and quicker to build the first version of the design using a conventional 1/16" rigid multi-layer printed circuit board with sufficient spacing to do rework and with headers for all signals that may need to be scoped or examined with a logic analyzer.

During this part of the process, the BIOS engineer will likely be switching back and forth between the standard evaluation board and the breadboard. Once the bread-board design is mostly up and running, layout of the form-factor design can begin. Often, as the result of experiences with the bread-board, changes to the circuitry will be made at this time. Once form-factor prototypes become available, the BIOS engineer will most probably still be switching back and forth between two BIOSes: the BIOS for the bread-board and the BIOS for the form-factor design.

This BIOS kit employs a scheme called the project concept to make it easier for the BIOS engineer to work with several different BIOSes simultaneously. Each of the three BIOS's envisaged in the example above would be different projects.

Suppose that the standard evaluation board was manufactured by the Super Duper Chip Company and featured their 386 single chip embedded micro-processor. The evaluation board might well be called a SDC386EV. It would be logical to use that as a project name.

When you purchased the BIOS kit, you probably selected some "personality modules" for it. Given that the SDC386EV is a standard evaluation board which is supported by the BIOS kit, two personality modules would be available for it. One would be known as a "board module" and be called SDC386EV (i.e., the same name as the evaluation board) and the other would be known as a "chipset module" and be called SDC386 (after the high integration embedded micro-processor on the board). These modules will have been developed in close cooperation with the Super Duper Chip Company and contain fixes and workarounds for chip and board anomalies. Developing your own chipset and board personality modules from ground zero will be a time consuming and, probably, frustrating process. In this chapter we will assume that you did purchase these two personality modules.

Having installed the BIOS kit and the personality modules, you will have the following subdirectories.

```
C:\
 └──EBIOS43
       │
       ├── BOARDS
       │    └── SDC386EV
       │
       ├── CHIPSETS
       │    └── SDC386
       │
       ├── CPUS
       │    └── NOCPU
       │
       ├── INC
       │
       ├── TOOLS
       │
       ├── UTIL
       │
       ├── RESOURCE
       │    ├── ADS
       │    ├── ICONS
       │    ├── IDF
       │    └── SPLASH
       │
       ├── PROJECTS
       │    └── SDC386EV
       │
       └── SYSTEM
            └── OBJ
                 └── SDC386EV
```

Note that the two subdirectories in bold type are the personality modules.

The subdirectory that is underscored is a holding directory in which object modules created by the build process (see Chapter 5) reside.

The italicized subdirectory is known as the Project Directory. It contains a file named `SDC386EV.INC`. This file is known as the Project file. The project include file contains the name of the board personality module, the chipset personality module and the values of any options and parameters which have been changed from their default values.

The following is a sample project include file.

```
;***   SDC386EV.INC -- Embedded BIOS Project File for the SDC386EV System.
;
;1.     Functional Description.
;       This include file defines CONFIG.INC & OPTIONS.INC overrides.
;
;2.     Modification History.
;       S. E. Jones   00/02/14.     #4.3, original.
;
;3.     NOTICE: Copyright (C) 1992-2000 General Software, Inc.

;       Required values:

CPUCLASS        equ     <NOCPU>
CHIPSET         equ     <SDC386>
BOARD           equu    <SDC386EV>

BIOS_LICENSEE EQU       'Unlicensed Demonstration Copy'

;       CONFIG.INC overrides:

CONFIG_CPU_TYPE             =       CPU_386
CONFIG_MAX_EXT_MEMORY       =       (64-1)*16 ; 64MB limit

CONFIG_CMOS_FLOPPY          =       DRIVE_144

;       OPTIONS.INC overrides:
```

The text in bold type specifies the personality modules.

Once you have built and tested a BIOS for the SDC386EV evaluation board (following the procedures contained in Chapter 5), you are ready to start on your second BIOS: the one for the bread-board system.

The first step is to choose a project name for this BIOS. Let us suppose that the hardware has the code-name RAINIER. A reasonable project name for bread-board BIOS would be RAINIER1.

The next step is to create the sub-directories and initial files for this new RAINIER1 project. This can be done with the following DOS commands.

```
MD C:\EBIOS43\SYSTEM\OBJ\RAINIER1

MD C:\EBIOS43\BOARDS\RAINIER1
```

```
COPY C:\EBIOS43\BOARDS\SDC386EV\SDC386EV.ASM
       C:\EBIOS43\BOARDS\RAINIER1\RAINIER1.ASM

COPY C:\EBIOS43\BOARDS\SDC386EV\SDC386EV.INC
       C:\EBIOS43\BOARDS\RAINIER1\RAINIER1.INC

MD C:\EBIOS43\PROJECTS\RAINIER1

COPY C:\EBIOS43\PROJECTS\SDC386EV\SDC386EV.INC
       C:\EBIOS43\PROJECTS\RAINIER1\RAINIER1.INC
```

The last step of the process of creating a new project is to edit the file `C:\EBIOS43\PROJECTS\RAINIER1\RAINIER1.INC` to change the name of the board personality module to RAINIER1.

You are now ready to start development of the BIOS for the bread-board design. A subsequent section of this chapter provides some hints regarding the changes that you may need to make.

Once the breadboard BIOS is up and running, another project could be created for the form-factor design. RAINIER2 might be a good name for it.

8.2 Selecting the Best Starting Point

As was noted in the previous section, implementing board or chipset modules from ground zero is a very time consuming process. Thus it becomes important to choose the best starting point.

Most hardware designers will pick a reference design that illustrates the use of the micro-processor chip that they had selected. The hardware designer will usually purchase an evaluation board that incorporates that reference design. In almost all cases, the best starting point for the BIOS engineer is the personality modules for that same evaluation board.

In some cases there is no evaluation board that uses the same set of chips as does the design for which the BIOS is being developed. For example, the chosen micro-processor may not include PCMCIA support, and the hardware design engineer may have selected a PCMCIA controller chip which is totally different from the one used in the reference design. In this case the personality modules for the reference design, while useful from many other aspects, are not helpful for creating PCMCIA code.

In this situation, the BIOS engineer should check other reference designs to see if there is one that uses the PCMCIA controller chip in question, even if the reference design uses a different micro-processor. If such a reference design is found, the board personality module for that design may be acquired and the relevant code ported from it.

If such a reference design cannot be found, it may be that a reference design and personality module do exist for an earlier version of the chip. Or it may be that they exist for a somewhat similar chip from another vendor.

In most cases it is cheaper in the long run to start from some tested and proven personality module rather than to start from nothing. Taking into account wages, benefits, taxes, lab space costs, utilities, and depreciation, a BIOS engineer usually costs an employer at least $1 per minute. It does not take too long to spend as much as it would have cost to buy a personality module.

8.3 Determining What Needs to be Customized

As important as determining what should be changed is understanding what should not be changed. The sub-directories C:\EBIOS43\INC and C:\EBIOS43\SYSTEM contain what is known as "the core". The core should not be changed unless there is no other way to make the BIOS work. If you change the core you will make it very difficult for you to pick up fixes and new features for your BIOS from General Software.

The architecture of the BIOS anticipates that changes will be made only to personality modules and project include files.

Assuming that the hardware design for which a BIOS is need is based on a reference design and evaluation board for which personality modules have been purchased, the most common changes required fall into these categories:

1. Project include file changes from standard defaults;
2. General purpose pin assignments (GPIOs);
3. Power control;
4. Watch dog timer;
5. PCI interrupt mapping; and
6. Super I/O programming.

8.3.1 Project File Symbol Overrides

There are many options and parameters which can be changed quite simply. Some of these options and parameters are intended to "tune" the BIOS to a specific hardware platform while others enable and disable features that may or may not be relevant to a specific hardware design. Most of these options and parameters are in the files OPTIONS.INC and CONFIG.INC. These files are in the directory C:\EBIOS43\INC. These two files should be carefully examined to determine if they contain the correct values, but these two files themselves SHOULD NEVER BE CHANGED. Instead, any line that ought to have a different value should be copied into the project include file and changed there.

8.3.2 General Purpose Package Pin Assignments

Most micro-processor chips intended for use in embedded systems contain pins that can be directly controlled by software. These are often known as "general purpose pins". The use of these pins varies greatly from design to design. The BIOS engineer will usually have to work closely with the hardware design engineer to ensure that the BIOS uses these general purpose pins correctly.

The architecture of the BIOS expects that all software that directly controls general purpose pins is confined to personality modules. Violation of this rule may make it very hard to pick up bug fixes later in the life of the product.

8.3.3 Defining Power Control

Control of power in embedded systems is probably the least standardized part of the system. If your system includes any kind of software controlled power management, you will probably have to modify some power related code.

Power control software can be divided into two parts: the APM routines; and the power manager.

The APM routines are contained in the personality modules. Their purpose is to directly interface with the hardware. They are typically called by the power manager to change the power consumption level of the hardware (i.e., to power some component on or off, or to change its speed) and to query the state of the power system, for example to check the level of charge of a battery, or to determine the presence or absence of AC power.

The APM routines do not normally make any decision concerning the power state of the system.

Two sets of options exist with respect to the power manager. They are to either use or not use the BIOS's built-in power manager; and to use or not use an external power manager.

If the built-in power manager is to be used, then appropriate parametric values must be set up in the project include file.

An external power manager is typically implemented as a device driver (usually called `POWER.SYS`) and loaded via `CONFIG.SYS`. An external power manager is typically required when power control decisions are based on events that are more complex than timer ticks, for example when there are user operated power control switches, or when it is necessary to respond to a change in availability of AC power.

8.3.4 Watchdog Timer

Many embedded systems incorporate a watchdog timer scheme which is intended to reboot the system if it hangs. In general it is up to the application program to use the INT 15h APIs to inform the BIOS when the watchdog timer should be enabled, disabled, and "kicked". While the watchdog timer is enabled, it must be kicked every so often or the system will be rebooted.

When the application program issues watchdog APIs, the personality modules are notified. If the watchdog timer feature is to be used, the BIOS engineer must put code in a personality module to enable, disable, and kick the underlying hardware.

8.3.5 PCI Interrupt Mapping

Targets that have one or more PCI busses will need some adaptation at the Board Personality Module level, in order to specify which PCI interrupt lines (INTA, INTB, INTC, INTD) are wired to which devices (or "slots"). This prevents the core from generically assigning an interrupt to a slot that cannot support that interrupt.

Additionally, there are some PCI tuning parameters that govern how resources are assigned during PCI initialization. These issues are discussed in a separate PCI chapter.

8.3.6 Super I/O Programming

Targets with embedded Super I/O controllers usually require some custom OEM-supplied code to program the Super I/O part to the desired configuration. For example, COM ports, LPT ports, FDC, HDC, keyboard and mouse controller, and so on, are all logical components of a Super I/O part that the OEM may wish to selectively enable or disable, or map to a specific I/O address or

IRQ. Because General Software cannot know in advance about how the Super I/O is wired into the target, this programming is necessarily unique to the OEM's Board Personality Module.

8.4 Building the BIOS

There are two possible ways to build a BIOS. One is to use an interactive Windows program called BIOStart, which not only builds a BIOS but also provides an interactive method of updating options and parameters in the project include file. BIOStart is described in Chapter 6.

The other way to build a BIOS is via a DOS-compatible program called GSMAKE. This method is intended to be used only by BIOS engineers. The GSMAKE method is described in Chapter 5.

Using the RAINIER1 project as an example, having successfully built a BIOS by either of these methods, a 64k file named `RAINIER1.ABS` will have been created in the sub-directory `C:\EBIOS43\PROJECTS\RAINIER1`. The first few times that you build a BIOS you should perform two checks of the file RAINIER1.ABS to make sure that it looks reasonable.

Firstly, enter the following DOS command.

```
DIR C:\EBIOS43\PROJECTS\RAINIER1\RAINIER1.ABS
```

Note: The size of the .ABS file depends on the OPTION_BIOS_KBSIZE configuration parameter. If the parameter is in the range 1KB-64KB, then the .ABS file will be 64KB in size, with any padding at the front (so as to force the boot vector at the top of the ROM space). When the parameter is increased to the range 65KB-127KB, the size of the .ABS file will be 128KB. Similar rounding-up occurs at 196KB and 256KB, the largest size available.

Secondly, enter the following DOS command.

```
DEBUG C:\EBIOS43\PROJECTS\RAINIER1\RAINIER1.ABS
```

DEBUG should respond with a prompt as follows.

```
-
```

Enter the following DEBUG command.

```
D 100
```

Debug should respond with something similar to the following, provided you've generated a simple, 64KB BIOS. (If you've built a BIOS of a different size, then you need to account for padding of the file from the beginning of the file).

```
0C9E:0100  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30   0000000000000000
0C9E:0110  30 37 2F 30 34 2F 30 30-28 43 29 31 39 39 32 2D   07/04/00(C)1992-
0C9E:0120  32 30 30 30 20 47 65 6E-65 72 61 6C 20 53 6F 66   2000 General Sof
0C9E:0130  74 77 61 72 65 2C 20 49-6E 63 2E 20 76 34 2E 33   tware, Inc. v4.3
0C9E:0140  2E 31 67 00 00 00 FF FF-FF FF FF FF 34 0B 3C 0B   .1g.........4.<.
0C9E:0150  44 0B 4C 0B 54 0B 5C 0B-64 0B 6C 0B FF FF 42 61   D.L.T.\.d.l...Ba
0C9E:0160  73 69 63 20 43 4D 4F 53-20 43 6F 6E 66 69 67 75   sic CMOS Configu
0C9E:0170  72 61 74 69 6F 6E 00 53-68 61 64 6F 77 20 43 6F   ration.Shadow Co
```

Exit from DEBUG by entering `Q (for Quit)`.

Assuming that the above checks produced acceptable results, you should now load the BIOS into the system by one of the mechanisms described in Chapter 5.

8.5 Getting Through POST

POST is an abbreviation for Power-On Self-Test. POST is the first code in the BIOS to be executed after a CPU reset occurs. POST not only tests the hardware but it also initializes it. When POST completes either the operating system (usually DOS) or an application program is loaded, or a menu is displayed allowing the user to determine how to proceed.

Most of the difficult debugging problems faced by a BIOS engineer involve failures during POST. Most of POST must execute before the video and keyboard sub-systems can be used. The first part of POST executes before DRAM is operational.

8.5.1 Using the Speaker to Report POST Failures

On a traditional PC a failure during POST is reported to the user via a number of beeps on the PC's speaker. The user is expected to count the number of beeps and guess at a likely cause of the failure based on that number. While this BIOS kit supports this method, it is almost always useless to the BIOS engineer as a debugging tool.

8.5.2 Using POST Codes to Report POST Failures

During POST, most traditional PC's generate an 8-bit number, called a Post Code, as POST executes. Each time a new Post Code is generated it is written to port 80h. If POST fails to complete properly, the last Post Code written to port 80h may provide a clue as to the nature of the failure. A short 8-bit ISA card, which will capture and display the last value written to port 80h, is available from several vendors. These cards are known as Post Code Cards or Port 80 Cards.

This BIOS kit supports Post Code Cards and routinely displays codes to port 80h as POST executes. The meaning of each code is briefly stated in the file `POST.INC`, which is in the sub-directory `C:\EBIOS43\INC`.

It is highly recommended that bread-board designs incorporate a built-in Post Code display.

If there is neither a built-in Post Code display nor an 8-bit ISA compatible connector, a logic analyzer may be required to capture Post Codes.

While Post Codes provide a more precise scheme than counting speaker beeps, Post Codes often do little more than narrow down the point of failure.

8.5.3 Using a Serial Port to Report POST Failures

This BIOS kit supports an optional serial port based method of reporting POST failures. It is known as POSTCODECOM and it can be enabled by inserting the following line in the project include file.

```
OPTION_SUPPORT_POSTCODES_COM = 1
```

If this option is enabled, short, somewhat descriptive, ASCII strings are sent to a serial port as each major block of code in POST is executed.

A PC connected to the system under test via a null modem or crossover cable and running a terminal emulator program such as PROCOMM can be used to capture and view these ASCII strings. By default the serial port at 3F8h is used with settings of 9600,8,N,1.

The BIOS engineer can easily include additional **POSTCODECOM** strings to help localize the point of failure.

8.5.4 Using the Graphical POST Test Icons

When a graphical POST interface is enabled, and a progress bar is available, the system will display icons showing the status of major devices tested by POST, such as low memory, extended memory, floppy disks, hard drives, and CD-ROM devices. The OEM can add OEM-defined devices and icons to this system. Note that the graphical POST system actually provides more information than the legacy POST display for disk devices.

8.5.5 Attempt to Boot an Operating System (DOS)

Once a logic analyzer, Post Code Display, and/or serial port link have been set up, you are almost ready to try booting.

If your test system includes an LCD display of the type which is sensitive to the order and timing of the LCD power controls, you should temporarily disconnect it to protect it from damage should POST fail at an unfortunate place.

Now go ahead and install the BIOS (if you have not already done so) and apply power to the system. Be prepared to shut the power off quickly if there is any indication of excessive heating (i.e., smoke, or a "hot" smell). Be very careful not to touch any hot components, nor to allow them to come into contact with any flammable or volatile materials.

Observe the system for indications of activity.

8.5.6 When Nothing Happens . . .

If nothing happens, start by checking that the BIOS was properly programmed into a Flash or EPROM and that the chip is properly installed in its socket. Check that all power connectors are correctly installed. Remove all plug in cards (except the Post Code card, if any) and all DRAM. Disconnect all peripherals (except the **POSTCODECOM** serial link, if any), and try again.

If nothing still happens, with the help of a hardware design engineer, identify a pin on the micro-processor chip that is under direct software control. Attach an oscilloscope to this pin, or use a logic probe to monitor the pin. It may be necessary to lift the pin or cut foils to prevent a fault or external load from holding the pin high or low. Note that some output pins are tied high or low externally and sensed during power-on reset to enable or disable features in the micro-processor silicon. Add code to the routine BoardInit0 in the board personality module to toggle this pin on and off at some frequency that can be conveniently scoped. Remember that the CPU could be running as slow as one tenth its nominal speed because it is running with default speed and wait state settings at this time.

Assuming that there are no coding errors, if no square wave is seen at the scope then either the wrong board personality module is being included in the BIOS, or there is a BIOS build problem, or the hardware is broken. If no BIOS errors are found, attach a logic analyzer to the address, data, and read strobe (probably output enable) pins of the Flash/EPROM and check for reasonable operation. Note that because the CPU may be pre-fetching instructions, you cannot easily determine which instructions are actually executed versus those that are pre-fetched and discarded.

If the logic analyzer indicates that only a few instructions were executed, then there may well be a bus contention, bus loading, or bus wiring problem. If no instructions are executed there may be a clocking, reset, or power problem.

PART II

BIOS FEATURES

This part of the EMBEDDED BIOS reference documentation discusses the major architectural features of EMBEDDED BIOS on an individual basis, and offers information about how to configure and use them in practical ways.

Chapter 9

THE INTEGRATED BIOS DEBUGGER

This chapter describes the operation of the *integrated BIOS debugger*. The purpose of the debugger is to provide BIOS-level debugging facilities to the BIOS customization and hardware development team and to the operating system and application engineers involved in bringing up the operating system or application software on the target. It is not intended for use as the only debugging tool for application programmers; it is mainly used for ensuring that the application software or source-level debugger is loaded properly in the system, and that all the BIOS services are available to the higher layer software.

9.1 How to Use the Debugger

The debugger can be activated in four ways.

1. On a PC-compatible platform, the BIOS debugger can be invoked through the console by pressing the keyboard's CTRL and LEFT-SHIFT keys simultaneously. Breaking into the debugger in this way suspends the execution stream in the system until it is resumed with the "G" (go) command. As part of the standard configuration options, the OEM can configure the debugger to communicate through a serial port rather than the console, if desired.

2. The debugger can also receive programmed control from an "**INT 3**" instruction in the DOS code, application code, or BIOS code. This can be useful in debugging EMBEDDED BIOS adaptations running on new hardware that aren't yet booting the operating system. It can also be used to check-out new hardware by manipulating I/O ports with debugger commands.

3. From the SETUP main menu, the **ENTER SYSTEM BIOS DEBUGGER** selection will enter the debugger. After use, typing the "G" (go) command will return to the SETUP screens.

4. As a boot action, as a last-ditch effort if the operating system cannot be booted from the appropriate drives or out of ROM, and the Manufacturing Mode link cannot be established.

To enable the debugger in your EMBEDDED BIOS adaptation, set **OPTION_SUPPORT_DEBUGGER** to 1. The debugger contains quite a few commands and also contains a disassembler, which includes a full opcode table (see related options of the form, **OPTION_DEBUG_*xxx***). This can take up quite a bit of space, and it may require that you increase the size of the BIOS itself by increasing **OPTION_BIOS_KBSIZE** to 64.

To enable access to the debugger through the SETUP screen system, enable **OPTION_SETUP_DEBUGGER**.

As with the SETUP system, the debugger can be configured to redirect its input and output over an OEM-defined serial port. To redirect debugger I/O over an RS232 serial line, enable **OPTION_SUPPORT_CON_REDIRECTOR**, and set **CONFIG_CON_REDIR_DEBUG** to the serial port number (1=COM1, 2=COM2, etc.) to use for remote debugging.

9.2 Debugger Command Syntax

Nearly all debugger commands are specified as a single abbreviated word such as "BIOSDATA", "REBOOT", or "G", followed possibly by expressions separated by tabs, spaces, or commas. Depending on the command’s function, the address operand may default to the "next approriate address" or it may be required in the event that there is no "next appropriate address". Command names are case-insensitive, as are the names of registers in operands.

9.2.1 Operand Types

Commands all take different operand types, depending on their function. For example, the command that outputs a 32-bit double word to a 32-bit I/O port requires a 32-bit datum, whereas the command that dumps memory uses an address of the memory area to dump.

The debugger accepts 8-bit, 16-bit, and 32-bit operands, as needed for a given command. In addition, real-mode (segment:offset) addresses, and 32-bit physical addresses, are often given. The Flash programming commands require a 32-bit address formed by an xxxx:yyyy syntax that looks like it should be a real-mode address, but is in fact a special way to enter a 32-bit physical Flash media address in two components, separated by a semicolon.

The 8-bit, 16-bit, and 32-bit operands are built from expressions.

Real-mode addresses (often called 16:16 addresses) are composed of two 16-bit expressions separated by a colon, where the 16-bit expression on the left hand side of the colon represents the segment, and the 16-bit expression on the right hand side represents the offset.

Physical addresses are indicated by a percent sign (%) followed by a 32-bit expression.

9.2.2 Expressions

The basic components of expressions are simple values, such as hexadecimal constants, or register names. For example, the constants 0, 1234, 17, DEADBEEF (an interesting-looking 32-bit hexadecimal number), and register names AX, BX, CX, DX, SI, DI, SP, BP, FL, CS, DS, ES, SS, FS, and GS are all simple values. The 8-bit register names are not valid simple values. The 32-bit register names EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP, are also valid when the

CONFIG_CPU_TYPE parameter has been set to **CPU_386** or above. Whenever an expression is called for, these values can be used alone.

In addition, parentheses may be used (**without any intermixed white space**) to specify a simple expression consisting of an operation to be performed on two values. For example, it is possible to take the sum, difference, logical AND, or logical OR of two values, or shift one value by the number of bits specified by the second value. Finally, anywhere a value may be specified, a simple expression may be specified.

This recursive definition leads to the following examples of 16-bit expressions that might be used in debugger operands:

1234	(constant value)
BX	(contents of BX general register)
(BX+1234)	(contents of BX plus 1234h)
(AX-2345)	(contents of AX minus 2345h)
(CX&55AA)	(contents of CX ANDed with 55aah)
(BP|2)	(contents of BP ORed with 0002h)
(FL>1)	(contents of FL shifted right one bit)
(ES<AX)	(contents of ES shifted left AX bits)

Here are some more complex 16-bit expressions:

(BX+(SI-23))	(add BX to the difference of SI and 23h)
((AX&7FFF)|(BX&8000))	(bottom 15 bits of AX ORed with top bit of BX)

Here is a more formal definition, using a modified BNF, of expression syntax:

```
<hex value>     ::=   <hex digit> | <hex digit> <hex value>

<reg16>         ::=   AX | BX | CX | DX | SI | DI | SP |
                      BP |  DS | ES | CS | SS | FS | GS | FL

<reg32>         ::=   EAX | EBX | ECX | EDX |
                      ESI | EDI | ESP | EBP | EFL

<register>      ::=   <reg32> | <reg16>

<value>         ::=   <hex value> | <register>

<operator>      ::=   '+' | '-' | '&' | '|' | '>' | '<'

<expr>          ::=   <value> | ( <expr> <operator> <expr> )
```

9.2.3 Addresses

Some debugger commands, such as U[nassemble] and D[ump bytes], allow an address to be specified. When this is the case, the address can be a real-mode address or a physical address.

Real-mode addresses consist of two 16-bit expressions separated by a colon without intervening whitespace. For example, F000:1234 specifies offset 1234h with respect to real mode segment F000h. Register names, and expressions involving constants and expressions, are supported. For

example, F000:(BX+52) specifies an offset calculated from the contents of the BX CPU register summed with the hexadecimal constant, 52h.

Physical addresses are specified by a 32-bit expression prefixed by a percent sign (%). The following are examples of 32-bit physical addresses:

`%00800000`	(address of first byte at 8MB boundary)
`%00100000`	(address of first byte of extended memory)
`%00000000`	(address of first byte of low memory)
`%FFFFFFFF`	(address of last byte in Pentium-class machine)

Flash media commands use a special form of addressing to indicate 32-bit physical addresses. This form consists of two 16-bit expressions separated by a colon, as with real-mode addresses. However, the two 16-bit expressions do not correspond to a segment:offset pair. Instead, the first 16-bit expression becomes the high 16 bits of the 32-bit address, and the second 16-bit expression becomes the low 16 bits of the 32-bit address. The following are examples of Flash addresses:

`0000:0000`	(address of first byte of low memory)
`0010:0000`	(address of first byte of extended memory)
`0080:0000`	(address of first byte at 8MB boundary)
`FFFF:FFFF`	(address of last byte in Pentium-class machine)

9.3 Command Reference

This section describes the individual debugger commands.

9.3.1 ? Command

The "?" command evaluates its operand as an expression and prints the resulting value.

Command Syntax:

`?` *Expression*

Parameters:

Expression - A required expression as specified earlier in this chapter.

Sample Output Display:

```
1234
```

9.3.2 + Command

The "+" command advances the instruction pointer (IP) by one byte. This command is useful when skipping over instructions.

Command Syntax:

```
+
```

Parameters:

none.

Sample Output Display:

none.

9.3.3 - Command

The "-" command backs up the instruction pointer (IP) by one byte. This command is useful when an instruction should be reexecuted.

Command Syntax:

```
-
```

Parameters:

none.

Sample Output Display:

none.

9.3.4 BC Command

The BC command allows the developer to clear an execution breakpoint.

Command Syntax:

`BC` *BreakpointNumber*

Parameters:

BreakpointNumber - A required expression parameter that specifies the number of the breakpoint to be cleared. The number of a breakpoint can be displayed with the BL command, and is displayed after the system processes a BP command.

Sample Output Display:

```
Breakpoint #0 cleared.
```

9.3.5 BIOSDATA Command

The BIOSDATA command allows the developer to inspect the major low-memory fields in the system at segment 40H.

Command Syntax:

```
BIOSDATA
```

Parameters:

none.

Sample Output Display:

```
--- BIOS Data Area at 0040:0000 ---

COM Ports: COM1=03F8 COM2=02F8 COM3=0000 COM4=0000
LPT Ports: LPT1=0378 LPT2=0000 LPT3=0000
Device Flags: 4427 BOOT_DRIVE MATH_COPROC MOUSE
Memory Size: 635 kb
Kbd Status:  General=00h, Extended=00
Disk Status: Floppy=00h, Fixed=40h
18Hz Ticks:  0000:04BB
Reset Flag:  0000

--- Extended BIOS Data Area at 9EC0:0000 ---

CMOS Cache:  9EC0:034E
```

9.3.6 BL Command

The BL command allows the developer to list the breakpoints that are currently active. If a breakpoint has a command string associated with it, the command string is displayed. Those breakpoints with no command string have no command string display.

Command Syntax:

```
BL
```

Parameters:

none.

Sample Output Display:

```
#0 - 0500:1d49      "U CS:IP;G"
#1 - 12d9:ef7c      "CSW 32 1A;R AX 1234;T"
#2 - 12e9:459a
```

9.3.7 BP Command

The BP command allows the developer to set an execution breakpoint at a specified address. Multiple breakpoints may be set at any given time. Please note that breakpoints work by storing an INT 3 instruction at the specified location; this is impossible in read-only memory. Breakpoints may only be set in RAM.

Command Syntax:

`BP` *Address* [“*CommandString”*]

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of a breakpoint to be set.

CommandString - An optional parameter, enclosed in double quotes, that specifies a sequence of commands separated by semicolons to be executed when the breakpoint occurs. If this parameter is not specified, then the standard breakpoint command is the R (register dump) command.

Sample Output Display:

```
Breakpoint #0 saved.
```

9.3.8 CIS Command

The CIS command allows the developer to display a portion of the memory address space with the formatting of a Card Information Structure as found in the configuration space of PCMCIA cards.

This command is intended for use in debugging embedded applications that have a dedicated PCMCIA card that must be configured for use in the target without card or socket services.

Command Syntax:

`CIS` *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of memory space to be formatted as a CIS.

Sample Output Display:

Dependent on PCMCIA Card Type.

9.3.9 CONSOLE Command

The CONSOLE command allows the developer to redirect the debugger's input and output to another device. Available devices are:

CON - the system keyboard and video display monitor
COM1 - the first communications port at 3f8h
COM2 - the second communications port at 2f8h

Command Syntax:

`CONSOLE` *Device*

Parameters:

Device - A required parameter that specifies the new console to redirect debugger output to, and to redirect debugger input from.

Sample Output Display:

none.

9.3.10 CSR Command

The CSR ("chip set read") command allows the developer to display the value held in a chipset register. If no chipset is configured for the BIOS adaptation, then this command cannot function properly.

Note that some chipset registers are write-only, and some chipsets (or their equivalents on high-integration CPUs such as the SC520) may have registers that read-out different values than the values written to them (bits flip, and some may stay high or low).

This command is very useful in conjunction with CSW to test chipset configuration values before building a new BIOS with best-guess values.

This debugger command calls the **CsReadReg** Chipset Personality Module function to perform the actual I/O to the chipset.

Command Syntax:

`CSR` *RegisterIndex*

Parameters:

RegisterIndex - A required expression that specifies the index of the register in the chipset to be read.

Sample Output Display:

```
1234h
```

9.3.11 CSW Command

The CSW ("chip set write") command allows the developer to set a chipset register to a specific value. If no chipset is configured for the BIOS adaptation, then this command cannot function properly.

Note that some chipset registers are write-only, and some chipsets (or their equivalents on high-integration CPUs such as the SC520) may have registers that read-out different values than the values written to them (bits flip, and some may stay high or low).

This command is very useful in conjunction with CSR to test chipset configuration values before building a new BIOS with best-guess values.

This debugger command calls the **CsWriteReg** Chipset Personality Module function to perform the actual I/O to the chipset.

Command Syntax:

`CSW` *RegisterIndex RegisterValue*

Parameters:

RegisterIndex - A required expression that specifies the index of the register in the chipset to be read.

RegisterValue - A required expression that specifies the value to be stored in the chipset register. Note that some chipsets use 8-bit values, and others use 16-bit values. See your chipset's programming documentation for details.

Sample Output Display:

None.

9.3.12 D Command

The D command allows the developer to display memory at the specified address, or at the address immediately following the last byte displayed with the last D, DB, DW, or DD command.

Command Syntax:

`D` *Address*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode or 0:32 physical address of memory to be displayed in the default format. If not specified, then the address is assumed to be the address immediately following the last byte displayed by the last D, DB, DW, or DD command.

Sample Output Display:

```
0040:0000 f8 03 00 00 00 00 00 00:bc 03 78 03 00 00 00 00  o.......L.x.....
0040:0010 7d 82 00 80 02 00 00 00:00 00 2c 00 2c 00 13 1f  }e.C......,.,...
0040:0020 13 1f 3f 35 0d 1c 03 2e:64 20 0d 1c 6f 18 75 16  ..>5....d...o.u.
0040:0030 74 14 0d 1c 62 30 6c 26:0d 1c 3f 35 0d 1c 01 00  t...b0l&..?5....
0040:0040 24 00 04 00 00 00 01 06:02 07 50 00 00 40 00 00  #.........P..@..
0040:0050 0b 18 00 00 00 00 00 00:00 00 00 00 00 00 00 00  <...............
0040:0060 07 00 00 b4 03 29 30 03:00 00 c8 00 b1 93 01 00  ....).0......o..
0040:0070 00 00 00 00 00 01 81 00:14 14 14 14 01 01 01 01  ......u.........
```

9.3.13 DA20 Command

The DA20 command allows the developer disable the A20 gate using the method configured in the BIOS adaptation.

Command Syntax:

```
DA20
```

Parameters:

none.

Sample Output Display:

```
A20 gate disabled.
```

9.3.14 DB Command

The DB command allows the developer to set the default memory display format to *bytes*, and then to display memory at the specified address, or at the address immediately following the last byte displayed with the last D, DB, DW, or DD command.

Command Syntax:

`DB` *Address*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode or 0:32 physical address of memory to be displayed in bytes. If not specified, then the address is assumed to be the address immediately following the last byte displayed by the last D, DB, DW, or DD command.

Sample Output Display:

```
0040:0000 f8 03 00 00 00 00 00 00:bc 03 78 03 00 00 00 00  o.......L.x.....
0040:0010 7d 82 00 80 02 00 00 00:00 00 2c 00 2c 00 13 1f  }e.C......,.,...
0040:0020 13 1f 3f 35 0d 1c 03 2e:64 20 0d 1c 6f 18 75 16  ..>5....d...o.u.
0040:0030 74 14 0d 1c 62 30 6c 26:0d 1c 3f 35 0d 1c 01 00  t...b0l&..?5....
0040:0040 24 00 04 00 00 00 01 06:02 07 50 00 00 40 00 00  #.........P..@..
0040:0050 0b 18 00 00 00 00 00 00:00 00 00 00 00 00 00 00  <...............
```

```
0040:0060 07 00 00 b4 03 29 30 03:00 00 c8 00 b1 93 01 00  ....).0......o..
0040:0070 00 00 00 00 00 01 81 00:14 14 14 14 01 01 01 01  ......u.........
```

9.3.15 DCACHE Command

The DCACHE command allows the developer disable CPU (L1) and chipset (L2) cache in the system using the methods configured in the BIOS adaptation. This can be used to determine if the cache is working properly.

Command Syntax:

```
DCACHE
```

Parameters:

none.

Sample Output Display:

```
Cache disabled.
```

9.3.16 DD Command

The DD command allows the developer to set the default memory display format to *doublewords*, and then to display memory at the specified address, or at the address immediately following the last byte displayed with the last D, DB, DW, or DD command. Data displayed in this format is in big-endian format (the numbers are real hexadecimal numbers that have been formatted by the debugger by swapping the low and high bytes of each word, and the low and high words of each doubleword).

Command Syntax:

`DD` *Address*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode or 0:32 physical address of memory to be displayed in doublewords. If not specified, then the address is assumed to be the address immediately following the last byte displayed by the last D, DB, DW, or DD command.

Sample Output Display:

```
0090:0000     6483:b3ea 4300:0004 7279:706f 7468:6769
0090:0010     2943:2820 3839:3120 6547:2039 6172:656e
0090:0020     6f53:206c 6177:7466 2000:6572 2020:2020
0090:0030     2020:2020 2020:2020 2020:2020 2020:2020
0090:0040     454c:4946 4346:0053 4200:5342 4546:4655
0090:0050     4300:5352 544e:554f 4400:5952 434b:5349
0090:0060     4548:4341 4552:4200 5600:4b41 4649:5245
0090:0070     5346:0059 4544:0044 4543:4956 4d4f:4300
```

9.3.17 DW Command

The DW command allows the developer to set the default memory display format to *words*, and then to display memory at the specified address, or at the address immediately following the last byte displayed with the last D, DB, DW, or DD command. Data displayed in this format is in big-endian format (the numbers are real hexadecimal numbers that have been formatted by the debugger by swapping the low and high bytes of each word).

Command Syntax:

`DW` *Address*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode or 0:32 physical address of memory to be displayed in words. If not specified, then the address is assumed to be the address immediately following the last byte displayed by the last D, DB, DW, or DD command.

Sample Output Display:

```
0090:0000     b3ea 6483 0004 4300 706f 7279 6769 7468
0090:0010     2820 2943 3120 3839 2039 6547 656e 6172
0090:0020     206c 6f53 7466 6177 6572 2000 2020 2020
0090:0030     2020 2020 2020 2020 2020 2020 2020 2020
0090:0040     4946 454c 0053 4346 5342 4200 4655 4546
0090:0050     5352 4300 554f 544e 5952 4400 5349 434b
0090:0060     4341 4548 4200 4552 4b41 5600 5245 4649
0090:0070     0059 5346 0044 4544 4956 4543 4300 4d4f
```

9.3.18 E Command

The E command allows the developer to change a series of consecutive 8-bit storage locations in memory.

Command Syntax:

`E` *Address Value1 [Value2 [Value3...]]*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode or or 0:32 physical address where the first byte in the sequence is to be stored. Subsequent bytes (if specified) are stored in consecutively higher bytes in memory.

Value1, Value2, etc. - A required set of one or more expressions that specify the hexadecimal 8-bit values to be stored at the specified address in memory.

Sample Output Display:

none.

9.3.19 EA20 Command

The EA20 command allows the developer enable the A20 gate using the method configured in the BIOS adaptation.

Command Syntax:

```
EA20
```

Parameters:

none.

Sample Output Display:

```
A20 gate enabled.
```

9.3.20 ECACHE Command

The ECACHE command allows the developer enable CPU (L1) and chipset (L2) cache in the system using the methods configured in the BIOS adaptation. This can be used to determine if the cache is working properly.

Command Syntax:

```
ECACHE
```

Parameters:

none.

Sample Output Display:

```
Cache enabled.
```

9.3.21 EFL Command

The EFL command allows the developer to erase a block of sectored Flash supported by the Flash device driver enabled in the core BIOS, if available.

This command uses the debugger's parsing routines that allow entry of 16:16 (real-mode) addresses, although the address that is actually being entered is a 32-bit physical address. The address is specified in two 16-bit parts, separated by a colon. This address format is purely for convenience and has nothing to do with 16:16 segment:offset addressing.

Command Syntax:

`EFL` *HighPhysAddr*:*LowPhysAddr*

Parameters:

HighPhysAddr - The top 16 bits of a 32-bit physical address that points to the first byte of a Flash block to be erased.

LowPhysAddr - The bottom 16 bits of a 32-bit physical address that points to the first byte of a Flash block to be erased.

Sample Output Display:

```
Flash block erased.
```

9.3.22 G Command

The G command allows the developer to resume execution from within the debugger.

Command Syntax:

`G` *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of a breakpoint to be set *before* execution begins at the current CS:IP address. If not specified, no breakpoint will be set.

Sample Output Display:

none.

9.3.23 HELP Command

The HELP command allows the developer to display a summary of commands that are supported by the debugger.

Command Syntax:

```
HELP
```

Parameters:

none.

Sample Output Display:

Short summary of available commands.

9.3.24 I Command

The I command allows the developer to issue a read to a byte-wide I/O port in the system. The value read from the port is displayed on the console.

Command Syntax:

`I` *IoAddress*

Parameters:

IoAddress - A required expression that specifies the hexadecimal I/O port to read the 8-bit quantity from.

Sample Output Display:

```
12
```

9.3.25 ID Command

The ID command allows the developer to issue a read to a dword-wide I/O port in the system. The value read from the port is displayed on the console.

Command Syntax:

`ID` *IoAddress*

Parameters:

IoAddress - A required expression that specifies the hexadecimal I/O port to read the 32-bit quantity from.

Sample Output Display:

```
12345678
```

9.3.26 IW Command

The IW command allows the developer to issue a read to a word-wide I/O port in the system. The value read from the port is displayed on the console.

Command Syntax:

`IW` *IoAddress*

Parameters:

IoAddress - A required expression that specifies the hexadecimal I/O port to read the 16-bit quantity from.

Sample Output Display:

```
1234
```

9.3.27 LFL Command

The LFL command allows the developer to lock a block of sectored Flash supported by the Flash device driver enabled in the core BIOS, if available.

This command uses the debugger's parsing routines that allow entry of 16:16 (real-mode) addresses, although the address that is actually being entered is a 32-bit physical address. The address is specified in two 16-bit parts, separated by a colon. This address format is purely for convenience and has nothing to do with 16:16 segment:offset addressing.

Command Syntax:

`LFL` *HighPhysAddr*:*LowPhysAddr*

Parameters:

HighPhysAddr - The top 16 bits of a 32-bit physical address that points to the first byte of a Flash block to be locked.

LowPhysAddr - The bottom 16 bits of a 32-bit physical address that points to the first byte of a Flash block to be locked.

Sample Output Display:

```
Flash block locked.
```

9.3.28 MASK Command

The MASK command allows the developer to specify a bit mask, called the "enabled" bit mask, that is used by Embedded DOS-ROM internal debugging macros (XPRINTF) at run-time, on certain platforms.

XPRINTF statements in the Embedded DOS-ROM kernel specify a bit mask that is ORed with the "enabled" bitmask. If any bits match, then the XPRINTF statement is executed.

This feature allows a developer to add or modify code to the Embedded DOS-ROM kernel and place actual debugging code in the kernel, tied to developer-assigned bits in this bit mask. Then, these bits can be selectively enabled or disabled using the Embedded BIOS debugger with this command.

Command Syntax:

`MASK` *BitMask*

Parameters:

BitMask - A required expression that specifies a 16-bit value containing a bit pattern to be used by the XPRINTF macros in debug-aware Embedded DOS-ROM kernel builds.

Sample Output Display:

None.

9.3.29 MODE Command

The MODE command allows the developer to change the mode of the current video output device. This works by issuing an INT 10h, function 00h, specifying the operand's value as the video mode.

Most commonly, this feature is used to reset the video mode after some graphics program has run, so that debugger output is visible on the screen. For example, if a graphics program, such as Windows, has painted the screen in some graphics mode, and CTRL-SHIFT has been used to break into the debugger, then the debugger's output won't be visible as text, but as a dot spray on the screen. Typing "MODE 7" would cause the debugger to reset the video card (and monitor) to mode 7, which is the standard monochrome mode.

Command Syntax:

`MODE` *VideoMode*

Parameters:

VideoMode - A required expression that specifies a new video mode to set on the current video display.

Sample Output Display:

None.

9.3.30 O Command

The O command allows the developer to issue a write to a byte-wide I/O port in the system. The value written to the port is specified as the second parameter.

Command Syntax:

`O` *IoAddress Value [... Value]*

Parameters:

IoAddress - A required expression that specifies the hexadecimal I/O port to write the 8-bit quantity to.

Value - A required expression that specifies the hexadecimal 8-bit value to write to the I/O port. If more than one *Value* is specified, then each value is written to the I/O port in the specified order, with interrupts disabled and no intervening I/O cycles.

Sample Output Display:

none.

9.3.31 OD Command

The OD command allows the developer to issue a write to a dword-wide I/O port in the system. The value written to the port is specified as the second parameter.

Command Syntax:

`OD` *IoAddress Value [... Value]*

Parameters:

IoAddress - A required expression that specifies the hexadecimal I/O port to write the 32-bit quantity to.

Value - A required expression parameter that specifies the hexadecimal 32-bit value to write to the I/O port. If more than one *Value* is specified, then each value is written to the I/O port in the specified order, with interrupts disabled and no intervening I/O cycles.

Sample Output Display:

none.

9.3.32 OW Command

The OW command allows the developer to issue a write to a word-wide I/O port in the system. The value written to the port is specified as the second parameter.

Command Syntax:

`OW` *IoAddress Value [... Value]*

Parameters:

IoAddress - A required expression that specifies the hexadecimal I/O port to write the 16-bit quantity to.

Value - A required expression that specifies the hexadecimal 16-bit value to write to the I/O port. If more than one *Value* is specified, then each value is written to the I/O port in the specified order, with interrupts disabled and no intervening I/O cycles.

Sample Output Display:

none.

9.3.33 PCID Command

The PCID command allows the developer to display the current configuration of the PCI subsystem, including all devices and bridges recognized by the BIOS.

Command Syntax:

```
PCID
```

Parameters:

None.

Sample Output Display:

```
PCI Device Table.
Bus Dev Func VendID DevID Class              Irq
 00  00  00   8086   7190  Host Bridge
 00  01  00   8086   7191  PCI-to-PCI Bridge
 00  07  00   8086   7110  ISA Bridge
 00  07  01   8086   7111  IDE Controller
 00  07  02   8086   7112  Serial Bus          5
 00  07  03   8086   7113  PCI Bridge
 00  12  00   1011   0014  Ethernet            10
 00  13  00   1101   9500  SCSI Controller     9
```

9.3.34 PCIR Command

The PCIR command allows the developer to read from the PCI configuration space associated with a device, function, and bus specified by the user. The debugger assumes a byte read to be the default when the system is started, otherwise it will use the last used value (i.e. B, W, or D).

Command Syntax:

`PCIR` *Index [Function [Device [Bus]]]*

Parameters:

Index - A required expression that specifies the index into the configuration space from which the data will be read.

Function - An optional expression that specifies the device's function number associated with the information to be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Device - An optional expression parameter that specifies the number of the device from which the data will be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Bus - An optional expression parameter (ranging from 0 to 255) that specifies the device's bus number associated with the information to be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Sample Output Display:

```
12
```

9.3.35 PCIRB Command

The PCIRB command allows the developer to read an 8-bit byte from the PCI configuration space associated with a device, function, and bus specified by the user.

Command Syntax:

`PCIRB` *Index [Function [Device [Bus]]]*

Parameters:

Index - A required expression that specifies the index into the configuration space from which the 8-bit byte will be read.

Function - An optional expression that specifies the device's function number associated with the information to be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Device - An optional expression parameter that specifies the number of the device from which the data will be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Bus - An optional expression parameter (ranging from 0 to 255) that specifies the device's bus number associated with the information to be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Sample Output Display:

```
12
```

9.3.36 PCIRW Command

The PCIRW command allows the developer to read a 16-bit word from the PCI configuration space associated with a device, function , and bus specified by the user.

Command Syntax:

`PCIRW` *Index [Function [Device [Bus]]]*

Parameters:

Index - A required expression that specifies the index into the configuration space from which the 16-bit word will be read.

Function - An optional expression that specifies the device's function number associated with the information to be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Device - An optional expression parameter that specifies the number of the device from which the data will be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Bus - An optional expression parameter (ranging from 0 to 255) that specifies the device's bus number associated with the information to be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Sample Output Display:

```
1234
```

9.3.37 PCIRD Command

The PCIRD command allows the developer to read a 32-bit doubleword from the PCI configuration space associated with a device, function , and bus specified by the user.

Command Syntax:

`PCIRD` *Index [Function [Device [Bus]]]*

Parameters:

Index - A required expression that specifies the index into the configuration space from which the 32-bit doubleword will be read.

Function - An optional expression that specifies the device's function number associated with the information to be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Device - An optional expression parameter that specifies the number of the device from which the data will be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Bus - An optional expression parameter (ranging from 0 to 255) that specifies the device's bus number associated with the information to be read. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Sample Output Display:

```
12345678
```

9.3.38 PCIW Command

The PCIW command allows the developer to write to the PCI configuration space associated with a device and function specified by the user. The debugger assumes a byte write to be the default when the system is started, otherwise it will use the last used value (i.e. B, W, or D).

Command Syntax:

`PCIW` *Index Data [Function [Device [Bus]]]*

Parameters:

Index - A required expression that specifies the index into the configuration space where the data will be written.

Data - A required expression that specifies the 8-bit, 16-bit, or 32-bit data to be written.

Function - An optional expression that specifies the device's function number associated with the information to be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Device - An optional expression parameter that specifies the number of the device to which the data will be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Bus - An optional expression parameter (ranging from 0 to 255) that specifies the bus of the device to which the data will be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Sample Output Display:

none.

9.3.39 PCIWB Command

The PCIWB command allows the developer to write an 8-bit byte to the PCI configuration space associated with a device and function specified by the user.

Command Syntax:

`PCIWB` *Index Data [Function [Device [Bus]]]*

Parameters:

Index - A required expression that specifies the index into the configuration space where the 8-bit byte will be written.

Data - A required expression that specifies the 8-bit hexadecimal data to be written.

Function - An optional expression that specifies the device's function number associated with the information to be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Device - An optional expression parameter that specifies the number of the device to which the data will be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Bus - An optional expression parameter (ranging from 0 to 255) that specifies the bus of the device to which the data will be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Sample Output Display:

none.

9.3.40 PCIWW Command

The PCIWW command allows the developer to write a 16-bit word to the PCI configuration space associated with a device and function specified by the user.

Command Syntax:

`PCIWW` *Index Data [Function [Device [Bus]]]*

Parameters:

Index - A required expression that specifies the index into the configuration space where the 16-bit word will be written.

Data - A required expression that specifies the 16-bit hexadecimal data to be written.

Function - An optional expression that specifies the device's function number associated with the information to be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Device - An optional expression parameter that specifies the number of the device to which the data will be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Bus - An optional expression parameter (ranging from 0 to 255) that specifies the bus of the device to which the data will be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Sample Output Display:

none.

9.3.41 PCIWD Command

The PCIWD command allows the developer to write a 32-bit doubleword to the PCI configuration space associated with a device and function specified by the user.

Command Syntax:

`PCIWD` *Index Data [Function [Device [Bus]]]*

Parameters:

Index - A required expression that specifies the index into the configuration space where the 32-bit doubleword will be written.

Data - A required expression that specifies the 32-bit hexadecimal data to be written.

Function - An optional expression that specifies the device's function number associated with the information to be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Device - An optional expression parameter that specifies the number of the device to which the data will be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Bus - An optional expression parameter (ranging from 0 to 255) that specifies the bus of the device to which the data will be written. If not specified, debugger will assume an initial value of 0, or use the last supplied value.

Sample Output Display:

none.

9.3.42 R Command

The R command allows the developer to display the contents of the general register set using the display format last commanded with R32 or R16. If this is the first register display command, then the initial register display format is selected based on whether 386 registers are available on the target or not.

Command Syntax:

`R`

Parameters:

none.

Sample Output Display:

```
EMBEDDED BIOS Debugger [IN BIOS] Copyright (C) 2000 General Software
AX=0093  BX=007a  CX=0001  DX=3d26  SI=001e  DI=0000  BP=03b6
CS=f000  DS=0040  ES=157b  SS=157b  SP=037e  IP=ebc3  NV UP EI NG NA PO ZR NC
f000:ebc3         cli
```

9.3.43 R16 Command

The R16 command allows the developer to display the contents of the general register set using the 16-bit display format.

Command Syntax:

```
R16
```

Parameters:

none.

Sample Output Display:

```
EMBEDDED BIOS Debugger [IN BIOS] Copyright (C) 2000 General Software
AX=0093  BX=007a  CX=0001  DX=3d26  SI=001e  DI=0000  BP=03b6
CS=f000  DS=0040  ES=157b  SS=157b  SP=037e  IP=ebc3  NV UP EI NG NA PO ZR NC
f000:ebc3         cli
```

9.3.44 R32 Command

The R32 command allows the developer to display the contents of the general register set using the 32-bit display format.

Command Syntax:

```
R32
```

Parameters:

none.

Sample Output Display:

```
EMBEDDED BIOS Debugger [IN BIOS] Copyright (C) 2000 General Software
EAX = 12345678  CS:EIP = F000:00000149  EFL = 001c213A
EBX = 00000001  SS:ESP = 02C0:00007FFE  EBP = 0000199C
ECX = 179D248E  DS:ESI = 74AB:00000511  FS  = 0000
EDX = 5555AAAA  ES:EDI = F000:0000E000  GS  = 0000
F000:00000149   cli
```

9.3.45 R32X Command

The R32X command allows the developer to display the contents of the general register set using the 32-bit display format, along with the special CPU control register set available on 386 and

above CPUs. The display of CR0 is especially useful to determine the CPU's cache and FPU management policies.

Command Syntax:

```
R32X
```

Parameters:

none.

Sample Output Display:

```
Embedded BIOS Debugger Breakpoint Trap
EAX = 0000000D  CS:EIP = E000:00002281  EFL = 00000296  NG nz .. AC .. PE .. nc
EBX = 756E000A  SS:ESP = 0000:00001FF2  EBP = 00001FDA  .. nt IOPL0 nv up EI ..
ECX = 6C65001A  DS:ESI = E000:00000BAB  FS  = 2504      .. .. id vp vi al vm rf
EDX = 49650C1B  ES:EDI = 9EC0:00002141  GS  = 0000
CR0 = 00000012  CR2 = 00000000  CR3 = 00000000  CR4 = 00000000
E000:00002281   retn
```

9.3.46 RC Command

The RC command allows the developer to read the contents of battery-backed CMOS memory. Either one byte of CMOS may be displayed, or the entire CMOS contents may be displayed.

Command Syntax:

`RC` [*CmosIndex*]

Parameters:

CmosIndex - An optional expression that specifies the CMOS address (actually, an index into the part) of the CMOS memory to be displayed. If no index is supplied, then the entire contents of CMOS are displayed.

Sample Output Display:

```
Addr  CMOS memory contents...
0000: 00 00 00 00 00 00 01 01 : 01 80 00 00 00 80 00 00
0010: 40 00 00 00 31 80 02 00 : 04 00 00 00 00 00 00 00
0020: 00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 f7
0030: 00 00 19 03
```

9.3.47 RD Command

The RD command allows the developer to issue an INT 13h read command so that hard drives, floppy disks, and their emulators, may be tested in the debugger environment. Operands of the RD command specify arguments that are normally passed in registers to INT 13h.

Command Syntax:

`RD` *DriveNo SectorNo HeadNo TrackNo Address*

Parameters:

DriveNo - An 8-bit expression that specifies the INT 13h unit number associated with the device to be read. For example, 0 is the first floppy, 1 is the second floppy, 80 is the first hard drive, and 81 (hexadecimal) is the second hard drive.

SectorNo - An 8-bit expression that specifies the sector number to be read. Sector numbers start with 1 and continue to the last sector number per track. For example, a 1.44MB diskette has sector numbers ranging from 1 to 18 (12 hexadecimal).

HeadNo - An 8-bit expression that specifies the head number to be read. Head numbers start with 0 and continue to the last head number. For example, a 1.44MB diskette has head numbers ranging from 0 to 1.

TrackNo - A 16-bit expression that specifies the track number to be read. Track numbers start with 0 and continue to the last track per cylinder. For example, a 1.44MB diskette has track numbers ranging from 0 to 79 (4f hexadecimal).

Address - A 16:16 real-mode address (physical addresses are not permitted) that specifies the memory location where the 512 byte sector will be transferred.

Sample Output Display:

```
Drive 00h, Sector 01h, Head 01h, Track 0042h read, status=00h.
```

9.3.48 RDMSR Command

The RDMSR command allows the developer to read a model specific register associated with an Intel Pentium or above CPU. Details about MSRs are beyond the scope of this document; consult your Intel documentation for details about what MSRs are available for your CPU.

Command Syntax:

`RDMSR` *RegisterNo*

Parameters:

RegisterNo – A 32-bit expression that specifies the model specific register number to read from. Consult your Intel documentation for details. This value is passed to a real RDMSR instruction in the ECX register.

Sample Output Display:

```
MSR register 12345678 = 23456789.abcdef01.
```

9.3.49 REBOOT Command

The REBOOT command allows the developer to reboot the system without removing power to the machine. This command causes the core BIOS to execute the OEM-defined reboot sequence, which may involve only the CPU, port 92h, the Chipset Personality Module, or the CPU Personality Module.

Note that the CPU restart vector on 286 and above processors is *not* F000:FFF0. These CPUs actually execute out of the very top of their physical address space, which in some cases is occupied by a boot loader such as the one provided by CyberQuest. Even though the CPU is executing out of memory above the 1MB address mark, it is still executing in real mode, not protected mode (really, real mode). Once the CS register is reloaded with any value, the CPU disables the upper address lines, and typically, continues to execute at the 8086-compatible reboot address, F000:FFF0.

On 286 and above CPUs, EMBEDDED BIOS enables the A20 line before rebooting the system. This allows special boot loaders to execute.

Command Syntax:

```
REBOOT
```

Parameters:

none.

Sample Output Display:

none.

9.3.50 RFL Command

The RFL command allows the developer to read data from a block of sectored Flash supported by the Flash device driver enabled in the core BIOS, if available. The data are displayed in words, because some Flash arrays only support word accesses.

This command uses the debugger's parsing routines that allow entry of 16:16 (real-mode) addresses, although the address that is actually being entered is a 32-bit physical address. The address is specified in two 16-bit parts, separated by a colon. This address format is purely for convenience and has nothing to do with 16:16 segment:offset addressing.

If the operand is not specified, then reading will continue where the last RFL command left off.

Command Syntax:

`RFL` [*HighPhysAddr*:*LowPhysAddr*]

Parameters:

HighPhysAddr - The top 16 bits of a 32-bit physical address that points to the first word of a Flash block to be read.

LowPhysAddr - The bottom 16 bits of a 32-bit physical address that points to the first word of a Flash block to be read.

Sample Output Display:

```
03a0:0000     b3ea 6483 0004 4300 706f 7279 6769 7468
03a0:0010     2820 2943 3120 3839 2039 6547 656e 6172
03a0:0020     206c 6f53 7466 6177 6572 2000 2020 2020
03a0:0030     2020 2020 2020 2020 2020 2020 2020 2020
03a0:0040     4946 454c 0053 4346 5342 4200 4655 4546
03a0:0050     5352 4300 554f 544e 5952 4400 5349 434b
03a0:0060     4341 4548 4200 4552 4b41 5600 5245 4649
03a0:0070     0059 5346 0044 4544 4956 4543 4300 4d4f
```

9.3.51 SFL Command

The SFL command allows the developer to write a 16-bit pattern to a specified number of words in a Flash array. This is used in situations where a Flash block must be written with all zeroes before erasing it.

Command Syntax:

`SFL` *HighPhysAddr*:*LowPhysAddr Count Word*

Parameters:

HighPhysAddr - The top 16 bits of a 32-bit physical address that points to the first word of a Flash block to be written.

LowPhysAddr - The bottom 16 bits of a 32-bit physical address that points to the first word of a Flash block to be written.

Count - A required expression that specifies the number of words to write in hexadecimal.

Word - A required expression that specifies the 16-bit value to be stored in each word.

Sample Output Display:

```
Data written to Flash.
```

9.3.52 SIOR Command

The SIOR command allows the developer to read an 8 or 16-bit value (depending on the implementation of the BPM's **BoardSioReadReg** function) from a specified register on a Super I/O component managed by the board module. This command simplifies the task of reading these higher-level registers by eliminating the need for programming individual index and data registers at the I/O port level.

The organization of the register set on the Super I/O component varies with these parts, and the partitioning of the register space is left to the BPM implementor.

The resulting output is printed in 16-bit format; however, whether the Super I/O part supports 8 or 16-bit data registers determines whether all 16 bits (or simply the bottom 8 bits) are actually meaningful.

Command Syntax:

`SIOR` *RegisterNo*

Parameters:

RegisterNo – An 8 or 16-bit register number (depending on the BPM implementation) that specifies which register on the Super I/O part to read data from.

Sample Output Display:

```
1234h
```

9.3.53 SIOW Command

The SIOW command allows the developer to write an 8 or 16-bit value (depending on the implementation of the BPM's **BoardSioWriteReg** function) to a specified register on a Super I/O component managed by the board module. This command simplifies the task of reading these higher-level registers by eliminating the need for programming individual index and data registers at the I/O port level.

The organization of the register set on the Super I/O component varies with these parts, and the partitioning of the register space is left to the BPM implementor.

Whether the Super I/O part supports 8 or 16-bit data registers determines whether all 16 bits (or simply the bottom 8 bits) supplied as the value parameter are actually used by the debugger.

Command Syntax:

`SIOW` *RegisterNo Value*

Parameters:

RegisterNo – An 8 or 16-bit register number (depending on the BPM implementation) that specifies which register on the Super I/O part to write data to.

Value – An 8 or 16-bit register number (depending on the BPM implementation) that specifies data to be written to the selected Super I/O register.

Sample Output Display:

```
none.
```

9.3.54 SO Command

The SO command allows the developer to redirect special debugging output from the XPRINTF macro in Embedded DOS-ROM to its own output device, such as CON, or COM1-COM4. For more information about XPRINTF debugging output see the section on the MASK command in this chapter.

Command Syntax:

SO *Device*

Parameters:

Device - A required parameter that specifies the new console to redirect Embedded DOS-ROM's XPRINTF output to. Supported device names are: CON, COM1, COM2, COM3, and COM4.

Sample Output Display:

none.

9.3.55 T Command

The T command allows the developer to trace through the current instruction and stop execution before the next one is executed. CALL and INT instructions are single-stepped by pushing into the called code; this command does not "step over" the instruction.

Command Syntax:

T

Parameters:

none.

Sample Output Display:

```
EMBEDDED BIOS Debugger [IN BIOS] Copyright (C) 2000 General Software
AX=0093  BX=007a  CX=0001  DX=3d26  SI=001e  DI=0000  BP=03b6
CS=f000  DS=0040  ES=157b  SS=157b  SP=037e  IP=ebc3  NV UP EI NG NA PO ZR NC
f000:ebc3         cli
```

9.3.56 TIME Command

The TIME command allows the developer to obtain a concrete CPU performance number associated with the target running the BIOS. The TIME command uses its operand value as a number of times to execute a lengthy loop of instructions that perform no useful work other than cause a delay before the prompt comes back.

As the operand's value increases, so does the time it takes for the TIME command to complete and return to the prompt. The relationship between the operand value and the time to complete the command is linear, making it possible to determine how much of a performance improvement certain changes in chipset programming, etc. is incurred.

Here is a simple way of measuring performance improvements:

1. Start with a simple system before the modifications. Suppose, for the sake of argument, that you are interested in how the CPU's incoming clock divisor (manipulated through some chipset register) affects CPU performance. Boot to the debugger, and type "TIME 10". We've chosen 10 here because it is a good starting point. Measure how long this takes.

2. Probably, 10 turned out to be too short or too long. However, you'll definitely know that, because your measurement will be hard to make in the short case, or difficult to wait for, in the long case. Come up with a better number (*n*) and run TIME on it. Use your stopwatch to measure how much time it takes. For real accuracy, we recommend some interval on the order of 30 seconds or so, to account for delays in starting and stopping the watch. Record the number of seconds it took to perform the TIME command. Call that *x*, here.

3. Now manipulate your chipset registers with the CSR and CSW commands.

4. Run the same TIME command with *n* as its parameter. Record the number of seconds it took to perform this TIME command. Call that *y*, here.

5. Now compute the performance improvement as (*y/x)*.

Command Syntax:

`TIME` *DelayFactor*

Parameters:

DelayFactor - A 16-bit expression specifying the amount of "work" to perform. There are many factors which, combined with this factor, cause the TIME command to delay a certain amount of time. *DelayFactor* is a linear parameter, which means that time taken to perform the command increases linearly with an increase in *DelayFactor* itself.

Sample Output Display:

none.

9.3.57 TORAM Command

The TORAM command copies the BIOS into low memory at **CONFIG_FLASH_CODESEG** and transfers control to the BIOS there. This allows the BIOS to run from RAM during tests involving reconfiguring chipset parameters that relate to the BIOS ROM.

Command Syntax:

`TORAM`

Parameters:

none.

Sample Output Display:

none.

9.3.58 TOROM Command

The TOROM command effectively cancels the TORAM command, causing execution to resume from the original location in ROM. If the BIOS was already running from ROM, then no operation takes place.

Command Syntax:

```
TOROM
```

Parameters:

none.

Sample Output Display:

none.

9.3.59 U Command

The U command allows the developer to display the contents of memory as a series of consecutive machine instructions. The instructions are formatted as 16-bit or 32-bit, depending on the last unassembly command, whether U, U16, or U32.

By default, the U command unassembles at the current CS:IP address after a debugger break-in. Subsequent U commands display the next few instructions, and so on. Specifying a new address with the U command causes subsequent U commands to display the instructions following the last U command.

Command Syntax:

`U` *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode or 0:32 physical address of the first instruction to be decoded and displayed. If not specified, the display will start with the first instruction that follows the one last displayed in a U command.

Sample Output Display:

```
033f:620b        mov     di, [0068]
033f:620f        mov     [di+06], ss
033f:6212        mov     [di+04], sp
```

```
033f:6215         mov     [di+02], fffd
033f:6219         call    61b7h
033f:621c         bkpt
033f:621d         retn
033f:621e         push    ds
```

9.3.60 U16 Command

The U16 command allows the developer to display the contents of memory as a series of consecutive machine instructions. The instructions are displayed in 16-bit format (16 bit instruction offsets, etc.)

By default, the U command unassembles at the current CS:IP address after a debugger break-in. Subsequent U commands display the next few instructions, and so on. Specifying a new address with the U command causes subsequent U commands to display the instructions following the last U command.

Command Syntax:

`U16` *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode or 0:32 physical address of the first instruction to be decoded and displayed. If not specified, the display will start with the first instruction that follows the one last displayed in a U command.

Sample Output Display:

```
033f:620b         mov     di, [0068]
033f:620f         mov     [di+06], ss
033f:6212         mov     [di+04], sp
033f:6215         mov     [di+02], fffd
033f:6219         call    61b7h
033f:621c         bkpt
033f:621d         retn
033f:621e         push    ds
```

9.3.61 U32 Command

The U32 command allows the developer to display the contents of memory as a series of consecutive machine instructions. The instructions are displayed in 32-bit format (32 bit instruction offsets, etc.)

By default, the U32 command unassembles at the current CS:IP address after a debugger break-in. Subsequent U commands display the next few instructions, and so on. Specifying a new address with the U-type command causes subsequent U commands to display the instructions following the last U command.

Command Syntax:

`U32` *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode or 0:32 physical address of the first instruction to be decoded and displayed. If not specified, the display will start with the first instruction that follows the one last displayed in a U command.

Sample Output Display:

```
033f:0000620b        mov     di, [00000068]
033f:0000620f        mov     [di+0006], ss
033f:00006212        mov     [di+0004], sp
033f:00006215        mov     [di+0002], fffffffd
033f:00006219        call    61b7h
033f:0000621c        bkpt
033f:0000621d        retn
033f:0000621e        push    ds
```

9.3.62 UFL Command

The UFL command allows the developer to update an area of Flash from another area of memory (such as the BIOS area at F000:0000).

This command copies the contents of memory specified by the 16:16 real mode address to the physical address.

The Flash must be erased before the update will work, because this command does not automatically erase the Flash before writing to it.

Command Syntax:

`UFL` *HighPhysAddr*:*LowPhysAddr Count SourceAddress*

Parameters:

HighPhysAddr - The top 16 bits of a 32-bit physical address that points to the first word of a Flash block to be written.

LowPhysAddr - The bottom 16 bits of a 32-bit physical address that points to the first word of a Flash block to be written.

Count - A required expression that specifies the number of words to copy in hexadecimal.

SourceAddress - Specifies the 16:16 real-mode address of an area of memory to be copied to the Flash.

Sample Output Display:

```
Flash Updated.
```

9.3.63 V Command

The V command allows the developer to display the contents of an interrupt vector by its number and save the address for a U command so that the code pointed to by that interrupt vector can be disassembled.

The V command is implemented solely to save the OEM time during debugging. The same results can be achieved with the DD command to display the interrupt vector table.

Command Syntax:

`V` *VectorNumber*

Parameters:

VectorNumber - A 16-bit expression that specifies a vector number from 00h to ffh, inclusive.

Sample Output Display:

```
Interrupt Vector 03h Contents:
033f: 620b         mov     di, [00000068]
```

9.3.64 WATCH Command

The WATCH command allows the developer to enable watchpoints inside the core BIOS flagged with **INTENTRY** and **INTEXIT** macro instructions in the source code.

All of the major interrupt service handlers in the BIOS (such as those for INT 10h, INT 11h, and so on) call these macros, one for entry and one for exit. Using the WATCH command, the developer can cause these macros to invoke the debugger's register dump facility to see the general registers on entry and exit to those interrupt handlers. This allows debugging of new BIOS code or analysis of requests made by higher-layer software such as DOS or Windows.

The WATCH command accepts one or more interrupt numbers as operands. If no operands are specified, then the current list of interrupts being watched is displayed. If operands are specified, then their watch status is toggled. So for example, to enable the watchpoint for the INT 10h service, "`WATCH 10`" would be specified. To disable the same watchpoint, the same command would be issued again.

Command Syntax:

`WATCH` *[IntNo [...IntNo]]*

Parameters:

IntNo - A 16-bit expression that specifies a BIOS service interrupt number to watch. Several of these may be specified as arguments.

Sample Output Display:

```
Watchpoint list:  10  11  15  19
```

9.3.65 WC Command

The WC command allows the developer to write a byte to battery-backed CMOS memory at the specified index.

Command Syntax:

`WC` *CmosIndex Value*

Parameters:

CmosIndex - A required expression that specifies the CMOS address (actually, an index into the part) of the CMOS memory to write to.

Value - A required expression that specifies the value to be stored in the specified CMOS location.

Sample Output Display:

none.

9.3.66 WCOMx Command

The WCOMx command allows the developer test a serial port by writing a hexadecimal value to a specified COM port a specified number of times. With large repeat values, the same character can be written out effectively continuously, so that serial ports can be tested with a logic analyzer, remote terminal software, or logic probe.

A period is printed on the primary debugging console to show the progress of writing to the UART, although the actual output goes out the specified device, which is typically not the debug output device.

Command Syntax:

`WCOMx` *ByteToWrite RepeatCount*

Parameters:

x - 1 for COM1, or 2 for COM2.

ByteToWrite - A required 16-bit expression that specifies the value to be written to the output data port of the UART.

RepeatCount - A required 16-bit expression that specifies the number of times to write the value to the UART in succession.

Sample Output Display:

```
Writing to COM1..........
```

9.3.67 WD Command

The WD command allows the developer to issue an INT 13h write command so that hard drives, floppy disks, and their emulators, may be tested in the debugger environment. Operands of the WD command specify arguments that are normally passed in registers to INT 13h.

Command Syntax:

`WD` *DriveNo SectorNo HeadNo TrackNo Address*

Parameters:

DriveNo - An 8-bit expression that specifies the INT 13h unit number associated with the device to be written. For example, 0 is the first floppy, 1 is the second floppy, 80 is the first hard drive, and 81 (hexadecimal) is the second hard drive.

SectorNo - An 8-bit expression that specifies the sector number to be written. Sector numbers start with 1 and continue to the last sector number per track. For example, a 1.44MB diskette has sector numbers ranging from 1 to 18 (12 hexadecimal).

HeadNo - An 8-bit expression that specifies the head number to be written. Head numbers start with 0 and continue to the last head number. For example, a 1.44MB diskette has head numbers ranging from 0 to 1.

TrackNo - A 16-bit expression that specifies the track number to be written. Track numbers start with 0 and continue to the last track per cylinder. For example, a 1.44MB diskette has track numbers ranging from 0 to 79 (4f hexadecimal).

Address - A 16:16 real-mode address (physical addresses are not permitted) that specifies the memory location where the 512 byte sector will be copied from.

Sample Output Display:

```
Drive 00h, Sector 01h, Head 01h, Track 0042h written, status=00h.
```

9.3.68 WFL Command

The WFL command allows the developer to write words of data to a block of sectored Flash supported by the Flash device driver enabled in the core BIOS, if available. The data are written in words, because some Flash arrays only support word accesses.

This command uses the debugger's parsing routines that allow entry of 16:16 (real-mode) addresses, although the address that is actually being entered is a 32-bit physical address. The address is specified in two 16-bit parts, separated by a colon. This address format is purely for convenience and has nothing to do with 16:16 segment:offset addressing.

Multiple data words may be specified on the command line, indicating that these words should be written to consecutive word addresses.

Command Syntax:

`WFL` *HighPhysAddr*:*LowPhysAddr Word1* [*Word2*] [*Word3*]...

Parameters:

HighPhysAddr - The top 16 bits of a 32-bit physical address that points to the first word of a Flash block to be written.

LowPhysAddr - The bottom 16 bits of a 32-bit physical address that points to the first word of a Flash block to be written.

Sample Output Display:

```
Data written to Flash.
```

9.3.69 WP Command

The WP command allows the developer to set a data watchpoint on a 16-bit storage area at the specified address. While the watchpoint is set, the processor enters *trace mode*, allowing the debugger to check the status of the storage area after the execution of each instruction to see if it has changed.

While it slows execution considerably (10x or more), a watchpoint can be very useful for finding instructions that are trashing memory.

Command Syntax:

`WP` *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of a 16-bit storage location in memory to be monitored. If not specified, the active watchpoint (if any) is cleared.

Sample Output Display:

```
Watchpoint saved.
```

9.3.70 WRMSR Command

The WRMSR command allows the developer to write a 64-bit value to a model specific register associated with an Intel Pentium or above CPU. Details about MSRs are beyond the scope of this document; consult your Intel documentation for details about what MSRs are available for your CPU.

Command Syntax:

`WRMSR` *RegisterNo HighValue LowValue*

Parameters:

RegisterNo – A 32-bit expression that specifies the model specific register number to write to. Consult your Intel documentation for details. This value is passed to a real WRMSR instruction in the ECX register.

HighValue – A 32-bit expression that specifies the high 32 bits of the data to be written to the MSR. This value is passed to the WRMSR instruction in the EDX register.

LowValue – A 32-bit expression that specifies the low 32 bits of the data to be written to the MSR. This value is passed to the WRMSR instruction in the EAX register.

Sample Output Display:

```
Setting MSR register 12345678 = 23456789.abcdef01.
```

9.4 PRINTF Output Formatting Macro

The integrated BIOS debugger provides output-formatting services in the style of the C-language *printf()* library function for use in debugging an adaptation of EMBEDDED BIOS. These services are available through a **PRINTF** macro in modules of the BIOS.

A PRINTF macro is defined in MACROS.INC for output formatting within the BIOS itself. PRINTF provides unconditional output, as is used by the system initialization code that displays the sign-on banner.

The PRINTF macros function very similarly to the C library's *printf* function. The remainder of this chapter discusses how to use the PRINTF macro and explains all of the formatting options.

The basic PRINTF macro syntax is as follows:

```
label   PRINTF <fmtstr> [, <arg1 [,argn]>]
```

The label field is used by the assembler and can be used to transfer control to the PRINTF statement. PRINTF doesn't do anything with the label itself.

The formatting string, *fmtstr*, is any sequence of characters that your assembler will accept as a string. The angle brackets surrounding the formatting string are used by the assembler to group the string's characters together, even if the string contains commas and other separators. Be aware that the PRINTF macro actually uses a DB statement in its expansion and surrounds the formatting string with single quotes; consequently, you must use two single quotes in succession whenever you wish to have one single quote printed in the string.

The formatting string is the basic template for the output to be performed. If no characters are present in the formatting string, then no output will be performed, regardless of the parameters specified in the argument list. The following is an example of a PRINTF statement that prints "Hello World.\n":

```
        PRINTF <Hello World!\n>
```

9.4.1 Literal Specifications

Notice that, as with the C-library *printf* function, PRINTF accepts literal characters, including the following:

`\n`	newline (CR/LF pair)
`\r`	carriage return (CR only)
`\t`	tab to next tab stop (1, 9, 17, etc.)
`\b`	bell character; beeps using BIOS
`\\`	display backslash character
`\$`	dollar sign (normally, $ is a formatting escape)

9.4.2 Format Specifications

Using PRINTF to output strings with literals is a basic function. Of course, you probably also have data that needs to be formatted in many ways, because you will most lkely hav ethe data in binary form in a register or in memory. To display data such as binary words and strings using PRINTF, we add two more components to the PRINTF macro calls. First, we add an argument list after the print formatting string. Second, we introduce formatting specifiers inside the formatting string.

The PRINTF macro accepts either one parameter or two parameters. If one parameter (enclosed in angle brackets) is specified, then that parameter is assumed to be a formatting string. If two parameters (both enclosed in their own angle brackets) are specified, separated by a comma, then the first parmaeter is assumed to be a formatting string, and the second parameter is assumed to contain a variable length list of arguments to be printed.

The argument list may contain zero, one, or more items to be formatted. The number of data items actually printed is a function of the print formatting string, and not the argument list. Hence the two parameters must be carefully and precisely coordinated.

Data items to be printed can be general processor registers, memory words, memory bytes, or strings in memory that are either fixed or variable length. Variable length strings may be terminated by a zero byte (00h) or a dollar sign ($).

The way that the data items are to be formatted is specified in your formatting string. All format specifications start with a dollar sign ($) and include a character after the dollar sign that indicates what type of formatting should be performed. The following table shows what formatting specifications are possible:

`$c`	prints bottom byte of word argument as raw character
`$u`	prints word argument as unsigned short number
`$d`	prints word argument as signed short number
`$x`	prints word argument as four hex digits
`$lu`	prints dword argument as unsigned long number
`$ld`	prints dword argument as signed long number
`$lx`	prints dword argument as eight hex digits
`$b`	prints bottom byte of word argument as two hex digits
`$s`	prints ASCIIZ string addressed by two word arguments
`$s$`	prints '$'-terminated string addressed by two word arguments

`$s[n]` prints fixed length string of n characters

9.4.2.1 $c Format Specification

The following example displays the character in the AL register as an ASCII (raw) character:

```
        PRINTF <The character in AL is $c.\n>, <ax>
```

Similarly, to print the contents of a byte located in memory, you would declare the byte with the DB statement, but actually specify a WORD in the PRINTF argument list:

```
OurChar DB      'A'             ; character to print.
        PRINTF <OurChar contains $c.\n>, <word ptr OurChar>
```

9.4.2.2 $b Format Specification

To display the contents of an 8-bit location in hexadecimal, the $b format specification must be used. While $c printed the quantity as a single character, the $b specification interprets the byte as a binary number from 0-255, and then formats the number in base 16. The result is two digits that can take on values from 00 to ff.

Because an 8-bit quantity is being displayed, the same rules for passing 8-bit arguments as defined in the $c formatting section apply here also. Therefore, you must fool the assembler and actually pass a word to the PRINTF macro to satisfy the macro expansion.

The following is an example call to display the contents of the CL register in hexadecimal format:

```
        PRINTF <The hex value in CL is $b.\n>, <cx>
```

9.4.2.3 $x Format Specification

The $x format specification is similar to $b, except that a 16-bit quantity is displayed in hexadecimal format instead of an 8-bit quantity.

The following example shows how to print the contents of the SI register using the PRINTF $x format specification:

```
        PRINTF <The hex value in SI is $x.\n>, <si>
```

Similarly, you can print the contents of a memory word with some form of the following example:

```
MyWord  DW      12345           ; a word in memory.
        PRINTF <MyWord in hex is $x.\n>, <MyWord>
```

Don't forget that the output is printed in hexadecimal. Therefore, this example doesn't print "12345", because that is the decimal value of the contents of MyWord. Instead, the $x format specification will display this value as "3039", because it is printed in base 16, not base 10.

9.4.2.4 $u Format Specification

The $u format specification is similar to $x, except that the 16-bit quantity is displayed in decimal (base 10) format instead of hexadecimal (base 16). The following example shows how to print the contents of the BX register using the PRINTF $u format specification:

```
PRINTF <The value in BX is $u.\n>, <bx>
```

9.4.2.5 $d Format Specification

The $d format specification is similar to $u, except that the 16-bit quantity is displayed in integer decimal (base 10) format instead of unsigned base 10 format. Thus, if the top bit in the word is set, then the value is treated as a 2's complement negative number, and is displayed after a minus sign to indicate that it is a negative quantity. Keep in mind that 16-bit words can hold positive nubmers in the range 0 to 32767 and negative numbers -1 to -32768. Thus, if you store the unsigned value 32768 in a word and then format it with the $d format specification, it will be printed as -1.

The following example shows how to print the contents of the AX register using the PRINTF $d format specification:

```
PRINTF <The value in AX is $d.\n>, <ax>
```

9.4.2.6 $lx Format Specification

The $lx format specification is similar to $x, except that a 32-bit quantity is displayed in hexadecimal format instead of a 16-bit quantity.

The following example shows how to print the contents of the 32-bit quantity in the register pair (DX:AX) using the PRINTF $lx format specification:

```
PRINTF <The hex value in DX:AX is $lx.\n>, <dx, ax>
```

9.4.2.7 $lu Format Specification

The $lu format specification is similar to $lx, except that the 32-bit quantity is displayed in decimal (base 10) format. The following example shows how to print the contents of the 32-bit quantity represented by the CX:BX register pair using the PRINTF $lu format specification:

```
PRINTF <The 32-bit value in CX:BX is $lu.\n>, <cx, bx>
```

9.4.2.8 $ld Format Specification

The $ld format specification is similar to $lu, except that the 32-bit quantity is displayed in integer decimal (base 10) format instead of unsigned format. Keep in mind that 32-bit dwords can hold positive numbers in the range 0 to 2^31-1 and negative numbers -1 to -2^31. Thus, if you store the unsigned value 4292967295 (the decimal equivalent of 2^31) in a longword and then format it with the $ld format specification, it will be printed as -1.

The following example shows how to print the contents of the DX:AX register pair using the PRINTF $ld format specification:

```
        PRINTF <The 32-bit value in DX:AX is $ld.\n>, <dx, ax>
```

9.4.2.9 $s Format Specification

The $s format specification allows you to display strings within your output. There are three forms of this format specification.

First, without any other modifiers, $s will simply output an ASCIIZ string (a variable length string containing a zero-byte at the end of it).

Second, by placing a dollar sign directly after the '$s', you can instruct PRINTF to display a variable length string terminated by a dollar sign instead of a zero byte. This string format is commonly found in DOS programs that use DOS function 09h, **DosConStrOutput**.

Third, you can use a format specification derived from the general form, '$s[n]', where the brackets tell PRINTF that the base 10 number inside the brackets is the length of the string.

Regardless of how the string is terminated or how long it is, its address must be fully specified in the argument list. Because strings are in general larger than one word, the address of the string instead of the string itself is passed in the argument list. And because the string may be located in any segment, both the segment and offset components of the string's address must be specified. As a result, you must specify two arguments (not one) in your argument list for every string formatter you use. The first argument is the segment address of the string, and the second argument is the offset address relative to that segment.

Finally, processors designed before the 80286 did not have a PUSH immediate instruction. As a consequence, it is not possible to push a segment value or an offset value of something without first putting it into a register, and then pushing the contents of the register. There is simply no instruction for pushing immediate data. We get around this processor limitation by simply storing string addresses in memory words or processor registers, and then passing the memory words or registers to PRINTF's argument list.

The following example shows how to print the contents of a string that is pointed to by the ES:DI register pair (there is nothing magic about ES and DI, it could have been AX:BX or BP:DX). The string is zero-byte terminated.

```
        PRINTF <The string contains "$s".\n>, <es, di>
```

This next example shows how to print the contents of a string that is statically declared as a memory array of bytes using the DB directive. The string is zero-byte terminated. We assume that MyString is in the data segment (addressable with DS).

```
MyString DB     'Hi there.', 0

        lea     ax, DGROUP:MyString
        PRINTF <The string is $s.\n>, <ds, ax>
```

If MyString had been assembled in the code segment, then the argument list <cs, ax> would have been appropriate. Remember that the LEA instruction is only one of several ways to get the address of MyString into the AX register. Another would be to use the MOV instruction with the OFFSET operator:

```
MyString DB     'Hi there.', 0

        mov     ax, OFFSET DGROUP:MyString
        PRINTF <The string contains "$s".\n>, <ds, ax>
```

Notice that we used the syntax "DGROUP:MyString". This indicates to the assembler that the offset component of the address is to be calculated relative to the group or segment called "DGROUP. DGROUP is not a magic name to the assembler, it is simply the most common name for the group of segments in the data group that most people use. If MyString had been in the code segment, and you were using CGROUP as a code group, then you would substitute "CGROUP" for "DGROUP" in the above example.

9.4.2.10 s Format Specification

So far, we have seen ways to print zero-byte terminated (ASCIIZ) strings with many different kinds of addressing. The s format specification allows the same addressing methods to be used, but simply defines the end of the string to be printed as the first occurrance of a dollar sign ($) in the string instead of a zero byte. here is an example where a $-terminated string is printed using the LEA-style addressing:

```
MyString DB     'Hi there.$'

        lea     ax, DGROUP:MyString
        PRINTF <The string contains "$s$".\n>, <ds, ax>
```

9.4.2.11 $s[n] Format Specification

Still a third way to define the end of a string to be printed with $s is to include the syntax, [n], following the $s specification. This tells PRINTF that the string is exactly n characters long, and that no characters in the string are to be treated as end-of-string terminators.

The following example shows how a fixed-length string can be printed with the PRINTF macro:

```
MyString DB     'abcdefghijklmnop'

        lea     ax, DGROUP:MyString
        PRINTF <The string contains "$s[16]".\n>, <ds, ax>
```

Chapter 10

THE BIOS POST INTERFACE

EMBEDDED BIOS can have a flexible interaction with the end-user, from a totally headless approach, to the common memory count-up display found on PCs, to a graphical POST with splash screen, optional icon progress bar and OEM-defined animations, which could be used by the OEM for announcements or advertisements, for example. This Chapter discusses how the user interface is configured from the project and IDF files.

10.1 Legacy POST Interface

Most desktop PCs provide the user with a text-based POST display that shows the BIOS vendor's copyright, a short description and encoded text about the nature of the platform, and then a memory count-up (perhaps with speaker clicks) that shows progress while memory is being tested. This interface has become so well-known that it expected on desktop computers.

This same interface can be used in embedded systems, but it may need to be more flexible. For example, POST messages themselves may need to be removed, or critical and soft errors (such as the detection of a missing keyboard) need to have different handling, or the entire output might need to be redirected over a serial port if it exists. The classic BIOS-based embedded system, such as a router or home gateway, often makes this basic interface available via an RS232 connection to test gear if it is present.

10.1.1 POST Messages

POST messages include all text-based characters displayed during POST's operation. This set includes the sign-on banner with General Software copyright, name of processor or chipset, the technical information displayed at the bottom of the screen, memory count-up display, and messages and queries such as "*press <DEL> to enter Setup*". These messages can be disabled by disabling **OPTION_SUPPORT_POSTMSGS** in the project file. Disabling these messages improves boot time.

The PCI subsystem also displays a table that shows the resource assignments of discovered PCI devices in the system. This table can be disabled by disabling **OPTION_SUPPORT_PCI_POSTMSGS**.

Soft errors (such as missing LPT port, memory mismatch, or invalid CMOS) can cause text displays during POST as well. These messages can be disabled by disabling **OPTION_SUPPORT_SOFT_ERR**.

The memory count-up is timed, to allow the user to see the count-up on faster boards. The delay used in the count-up is configurable in the project file with the **CONFIG_WAIT_COUNT** parameter, or it can be disabled by disabling **OPTION_MEMTEST_WAIT**. The **OPTION_MEMTEST_CLICK** option, when enabled, causes the PC speaker to click for each 64KB chunk of memory as it is tested.

Some options in the project file provide a way to control user queries during POST. There are optional queries to enter the debugger, enter the Setup screen, format RAM disks, format RFD disks, and verify RFD disk integrity. These can all be disabled in the project file to streamline POST and make an automated and unattended POST work without user intervention.

Finally, a “configuration box” can be displayed after POST has completed, but right before control is passed to the boot activities, such as loading the operating system from disk, or launching Manufacturing Mode or the debugger. This configuration box is painted using INT 10h services (which work when using console redirection as well) and is enabled or disabled with the **OPTION_SUPPORT_CONFIGBOX** parameter.

10.1.2 Critical Errors

Critical errors occur during POST before hardware is sufficiently initialized to allow the display of textual messages on the console device. Commonly on the desktop, critical errors are signaled by a beep on the PC speaker. The end user counts the number of beeps, or records a pattern of beeps, and then consults a manual to determine the nature of the pre-boot failure.

EMBEDDED BIOS provides critical errors, and provides for them to generate legacy PC speaker beeps, and/or other activities, including blinking of a floppy drive light, invocation of Manufacturing Mode, or execution of an OEM-written routine in the Board Personality Module (routine **BoardPostError**). This flexibility allows the OEM to implement any policy desired in the embedded target. By default, **OPTION_CRITICAL_BEEP** is enabled, and the other options are disabled, causing the BIOS to behave as expected for a legacy PC situation.

Critical errors are caused by error sources called hard errors. These sources can be enabled and disabled on an individual basis. Thus, the results of certain equipment tests can be ignored, allowing POST to continue even when the error occurs. These options, named **OPTION_HARDERR_*xxx***, where *xxx* is the name of the test, are selectable in the project file.

10.1.3 Soft Errors

Soft errors are like critical errors, but are generated after video is generally available, and so they generate textual messages instead of beeps. Like their critical error brothers, they are multi-sourced and configurable on an individual basis.

Soft error sources include LPT ports missing, memory size mismatch (legacy PC/AT soft error used to notify when installed memory has changed size), invalid CMOS contents (usually due to battery or power failure), as well as errors generated by the OEM’s code in the Board, Chipset, and CPU Personality Modules. The options for selectively enabling and disabling these error generators, are named **OPTION_SOFTERR_*xxx***, where *xxx* is the name of the test.

10.1.4 Console Redirection

Console redirection is an integrated feature of EMBEDDED BIOS that provides a way for INT 10h video output requests and INT 16h keyboard input requests to be routed over an RS232 connection, normally to terminal emulation software such as HyperTerminal or PROCOMM running on a host PC. Console redirection supports three functional channels of I/O, including POST/DOS, Setup, and the integrated debugger. Each of these channels is independently routable to the standard keyboard and video drivers (the traditional PC keyboard and screen) or any configured serial ports.

Console redirection is a highly-configurable component of EMBEDDED BIOS. The feature can be enabled independently of the selection of the drivers for the standard video and keyboard devices, allowing the feature to remain at the ready should the console need to be redirected on-the-fly at runtime or even when a specific component (such as Setup or the Debugger) are executing. To enable the feature, set **OPTION_SUPPORT_CON_REDIRECTOR** to 1.

There are many qualifiers for this feature, allowing it to be tailored to meet the policy needs of the OEM. The default I/O channels for console redirection are specified with **CONFIG_CON_REDIR_POST**, **CONFIG_CON_REDIR_DEBUG**, and **CONFIG_CON_REDIR_SETUP**. Additional options provide for the cancellation (i.e., dynamic resetting of these channels to 0, the standard keyboard and screen) under various conditions. To disable the console redirection if a serial timeout occurs, set **OPTION_CON_REDIR_TIMEOUT**. To disable the console redirection if a keypress is detected at the main keyboard, set **OPTION_CON_REDIR_CANCEL**. To disable the console redirection if no video device is present, set **OPTION_CON_REDIR_AUTO**. This last redirection cancellation feature causes a Board Pesonality Module routine, **BoardAutoRedirect**, to be called, allowing the OEM to change the way the BIOS detects the presence or absence of a video controller.

10.2 Graphical POST Interface

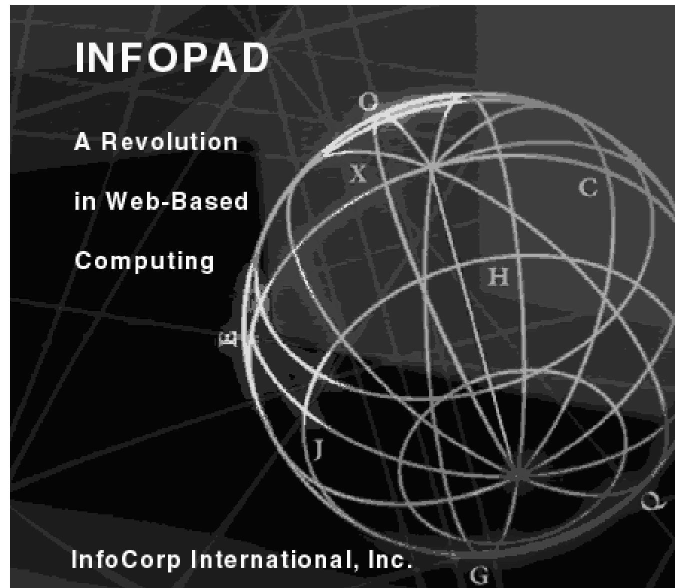
While desktop PCs must all operate substantially the same during the pre-boot environment in order to allow a huge user base to interact with them without any significant training, visually-intensive embedded PCs, like factory controllers and internet appliances, have quite a different goal. Most visual embedded systems need to establish distinctive qualities that separate them from other products on the market. This requirement makes it essential to have a non-legacy, graphical interface available during the pre-boot environment.

The graphical environment addresses many needs. First, it provides a less-technical indication to the user that the embedded system is working. Second, it allows the OEM to brand the system and control its visual look-and-feel, moving it away from a thinly-disguised PC. Third, it provides a way to graphically display the progress of the pre-boot environment, by displaying icons instead of error messages and memory count-up displays. Finally, it bridges the gap seamlessly from the moment the display is activated to the point where the application program takes over control of the screen.

The graphical POST environment can be enabled at the same time that the legacy POST messages, critical errors, soft errors, and configuration box options are enabled. This allows the user to select which type of POST interface will be used on the next boot with a setting in the Basic Setup screen. When the graphical POST system is activated, it automatically intercepts all INT 10h cursor updating and character display functions so that they don’t overlay the graphical

system. Then, when the first "set mode" INT 10h function is detected, the graphical POST releases control over the INT 10h service, allowing the operating system and/or the application to manage the display.

10.2.1 Splash Screens

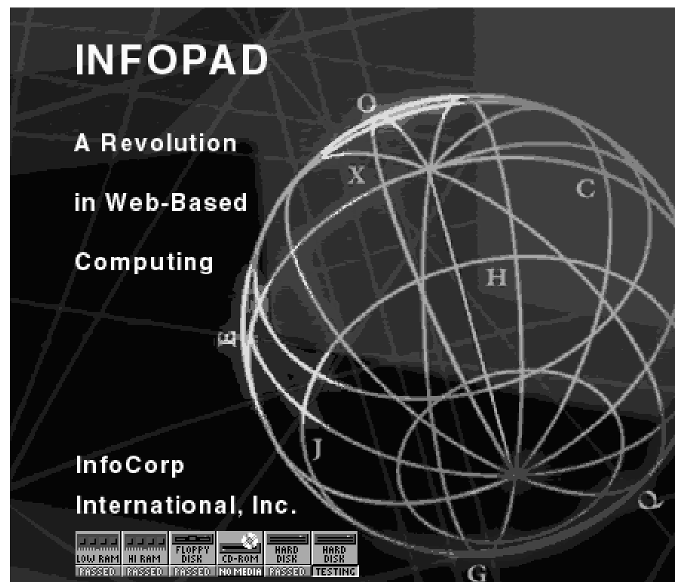


The most commonly-used feature of the graphical POST system is the splash screen. Using a statement in the **SPLASH_TABLE** in the project file, a bitmapped graphic may be selected from any of several graphic resources merged into the final BIOS image for display as early as possible during POST. This effectively makes the interface graphical, and allows the OEM to define the look-and-feel of the system, and brand it as well with completely custom graphics.

There are many design issues to consider when planning a splash screen to be displayed, including palette organization, and how the palette is shared among all the graphical images in the system. Additionally, there may be space constraints in the system that make it necessary to simplify the graphic in order to make it fit in the target's media address space.

The graphic above shows how a simple graphic and some text can be put together to create a splash screen that identifies the device, reinforces what it does, and brands it with the manufacturer's name. Legal notice: The graphic used here is copyright General Software, Inc. and cannot be used by anyone without written permission from General Software, Inc. The text is purely fictional and is not intended to represent any offerings by any third parties.

10.2.2 POST Progress Icons



Just as the text-based POST displays its progress in the form of critical errors, soft errors, and memory count-up, the graphical POST system provides a way to visually show the user how POST is proceeding, and the status of the POST tests. This is done by associating special events in POST (such as the start and end of a memory test, or disk test, etc.) with graphic resources, again using the **SPLASH_TABLE** table in the project file. The next graphic shows how the above graphic can support overlayed icons that show POST's progress.

These icons can be regularly spaced horizontally, vertically, or even at X/Y displacements to produce cascading effects. Commonly, they are placed at the bottom of the screen, or on the left-hand side. Note in the example design shown here, the design of the graphic's text had to be adjusted to fit the icon bar in the system.

OEMs can add their own icons to the system, or can replace the icons supplied by General Software. This can be useful to support specific markets, such as non-English speaking ones, or ones requiring totally graphic representation without any text.

10.2.3 Still and Animated Bitmaps

While the splash screen which we've already seen is an example of a still bitmap that spans the entire height and width of the screen, it is possible to display other bitmaps without covering the entire screen, and even overlay or redraw the same bitmaps in several different places on the screen. With these basic tools in hand, it is possible to create a splash screen that makes room for overlayed animated sequences, or advertisements, or moving objects that make the display more active while POST runs.

Probably the simplest paradigm for this is the addition of web-style advertisement banners to the splash screen. To add this, simply create room in the screen-sized splash screen for the advertisement banners to be placed. Then, specify in the **SPLASH_TABLE** those additional graphic resources and associate them with the rotating advertisement POST events. This will cause POST to select a different advertisement on each boot, rotating all of them in a sequence on successive boots.

To display all of the advertisements for each boot, the same technique is used, but instead of associating the advertisement graphics with the advertisement POST events, they are all associated with the initialization event, and specified in the proper sequence in the **SPLASH_SCREEN** table in the project file. Finally, delays may be inserted in between these graphics by coding a special resource ID as an object to be "displayed".

10.2.4 Configuration

The EMBEDDED BIOS graphical POST is configurable by the OEM in a number of ways. The OEM can specify certain graphical resources, or bitmaps, to be displayed in a specific order, and when certain events occur during POST. The central mechanism for managing these resources within the 16-bit BIOS build is the **SPLASH_TABLE** table, in the project file.

Once specified, the images may be created by the OEM using standard Windows .BMP development tools, such as Paint or Photoshop, to list two commonly-known tools. Careful planning and attention to procedure is necessary to manage the palette and limit resource size.

Finally, the graphics engine that draws the images can also be configured to accommodate certain variances in the hardware.

10.2.4.1 The SPLASH_TABLE Table

The **SPLASH_TABLE** table, specified in the project file, is used to define all the elements of the graphical POST system, including the splash screen, POST progress icons, banner advertisement graphics, and even the delays inserted between graphics drawn for the same

system event. Although a practical how-to discussion will be presented here, a more formal discussion of this project file statement is provided in Chapter 7.

The graphical POST system uses the run-time services of the EMBEDDED BIOS External Resource Manager to retrieve graphics by their *Graphics Resource ID*, a 16-bit identifying number specified in the .IDF file read by GSMERGE when combining various components of the composite BIOS. The GSMERGE step is where graphic resources, such as icons and splash screen bitmaps, get converted to the proper (internal RLE) format and merged into the final BIOS image. At the same time that this is done, GSMERGE creates an external object directory that EMBEDDED BIOS uses at runtime to find all the external resources. GSMERGE is therefore responsible for the link between Resource Ids and graphics files.

The splash table itself is specified in a tabular format with **SPLASH_TABLE** entries in the project file. Each line in the table specifies a new graphical component of the total graphical POST sequence, and begins with the identifying macro command, **SPLASH_TABLE**. Each line contains exactly four (4) operands, as in the following *hypothetical example*:

```
SPLASH_TABLE EVENT_SPLASH_INIT, RESOURCE_ID_SPLASH, SPLASH_CENTER, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_INIT, RESOURCE_ID_BOOTMSG, SPLASH_CENTER, 9800
SPLASH_TABLE EVENT_SPLASH_MSG1, RESOURCE_ID_ADVERT1, SPLASH_CENTER, SPLASH_TOP
SPLASH_TABLE EVENT_SPLASH_MSG2, RESOURCE_ID_ADVERT2, SPLASH_CENTER, SPLASH_TOP
SPLASH_TABLE EVENT_SPLASH_MSG3, RESOURCE_ID_ADVERT3, SPLASH_RIGHT, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_MSG3, RESOURCE_ID_ADVERT4, SPLASH_LEFT, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_MSG3, 0FF10h, SPLASH_LEFT, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_MSG3, RESOURCE_ID_ADVERT5, SPLASH_LEFT, SPLASH_CENTER
SPLASH_TABLE EVENT_SPLASH_ICON, SPLASH_ICON_RIGHT+32, 131, 9868 ; define progress bar
```

The first operand specifies an event code that is associated with the entry. Multiple entries may be associated with the same event code, in which case they are all activated when that event occurs in the system. (Note: Only one instance of **EVENT_SPLASH_ICON** is permitted, however.) The following events are defined by the architecture:

EVENT_SPLASH_INIT (00h) – The initial call to the splash screen.
EVENT_SPLASH_ICON (01h) – Progress bar Icon location/ordering request.
RESERVED (02h-0fh) – Reserved for future expansion.
EVENT_SPLASH_MSG1 (10h) – First boot advertisement after initial splash screen.
EVENT_SPLASH_MSG2 (11h) – Second boot advertisement after initial splash screen.
EVENT_SPLASH_MSG3 (12h) – Third boot advertisement after initial splash screen.
EVENT_SPLASH_MSG4 (13h) – Fourth boot advertisement after initial splash screen.
EVENT_SPLASH_MSG5 (14h) – Fifth boot advertisement after initial splash screen.
EVENT_SPLASH_MSG6 (15h) – Sixth boot advertisement after initial splash screen.

The second operand specifies the graphic resource ID to be associated with the event. When the event occurs, all of the graphics defined in the **SPLASH_TABLE** with a matching event code are displayed in the order they occur. If this parameter is specified to be a value of the form *0ffxxh*, then the system will delay for *xxh* 55ms timer ticks instead of displaying a graphic.

The third operand specifies the horizontal component of the location at which the associated graphic will be drawn. If **SPLASH_CENTER** (5000) is specified, the graphic will be centered on the screen. If **SPLASH_LEFT** (0) is specified, the graphic will be left-justified. If **SPLASH_RIGHT** (10000) is specified, the graphic will be right-justified. Any other value will be used as a virtual position within this framework (think of 5000 as 50.00 percent of the screen, 0 as 0.00 percent of the screen, and 10000 as 100.00 percent of the screen, so that a new number like 7500 would represent 3/4th of the way across the screen.

The fourth operand specifies the vertical component of the location at which the associated graphic will be drawn. If **SPLASH_CENTER** (5000) is specified, the graphic will be cetnered between the top and bottom of the screen. If **SPLASH_TOP** (0) is specified, the graphic will be displayed at the top of the screen. If **SPLASH_BOTTOM** (10000) is specified, the graphic wil be displayed at the bottom of the screen. Any other value will be used as a virtual position within this framework (see the description for the 3rd operand, above).

For the **EVENT_SPLASH_ICON** table entry, the second, third, and fourth operands have different meanings. This table entry, when specified, includes the graphical progress bar as a visible component of the graphical POST system. The second parameter defines both the X and Y travel directions and a scalar displacement, used in each direction specified, of each icon with respect to its predecessor. The third parameter specifies the starting X location in the range 0-10000, and the fourth parameter specifies the starting Y location in the range 0-10000, of the first icon in the graphical progress bar.

In the above example, some entries share the same event in the table. When those events are triggered, more than one graphic is drawn. This feature provides the ability to perform animation by drawing successive graphics, not necessarily in the same location.

A different feature is the ability to use the same graphic ID in different entries in the table. This allows reuse of the graphic for different situations, saving the need to duplicate the graphic physically in the build.

10.2.4.2 Graphical Resources

Although the External Resource Manager supports many types of external resources, the graphical POST system only supports a run-length-encoded file format designed by General Software (.RLE extension). This is not the same as the RLE compression format used when saving .BMP files in PhotoShop.

The Windows .BMP format was chosen for the graphical POST system because it is in widespread use in the industry. There are some features of that format that are not supported, however:

First, Microsoft documentation states that DDB (Device Dependent Bitmaps) are hardly used any more for graphics files. Therefore, they aren't supported. Only DIB (Device Independent Bitmap) files are supported.

To keep the decoding/encoding steps simple, only uncompressed .BMP files are supported. GSMERGE automatically performs RLE compression on these files before combining them with the BIOS image as external resources. Do not enable any kind of compression when saving .BMP files that will be used as input by GSMERGE for the graphical POST system.

Although the encoder in GSMERGE supports 16-bit and 24-bit images, there is no decoder for these formats in the graphical display drivers in the graphical POST system. This may be supported in the future.

Since images with more than one bit plane are rare, and because multiple planes are normally used in DDB, not DIB-based files, there is no support for them. If your image editor stores 16-color .BMP files using bit planes, simply save the file as a 256-color image instead. This will have no discernable impact on the size or support of the final .RLE file unless you actually use all 256 colors.

10.2.4.3 Creating Bitmaps and Icons

Bitmaps and icons are best created in a graphics design tool, such as PhotoShop. It is possible to create .BMP files compatible with EMBEDDED BIOS in Windows Paint, but it is difficult to do so, since Paint has no palette management tools. Thus, it is difficult to define 16 colors, which can match-up to the capabilities of the hardware in 640x480x16 mode.

Some general rules of thumb should be followed in order to keep the size of graphical images within reasonable bounds so they don't exceed the ROM space available in the target. These are not hard-and-fast rules, but simply recommendations. For example, if you absolutely must use the standard corporate graphics already pre-approved by a legal department, it can be accommodated by the system, provided it stays within the 16 color budget, and provided those colors are supported by the hardware in the mode used by the graphical POST system, and then provided all of the images can be combined with the other BIOS components to produce a final output file that can fit in the ROM component.

1. Avoid large numbers of colors. Using more than four colors will start to bloat the file size. If more than 16 colors are used, it will not be displayed properly by the graphical POST system.

2. When storing the length of each consecutive string of colors, the RLE compression bit packs the line length. Fewer bits are used to represent details on the right side of the screen. If you must include large amounts of text, try to place the text as close to the right edge of the screen as you can, or take care of it separately. You can also make images smaller by making them less wide.

3. Do not use image enhancements such as dithering, horizontal gradient shading, anti-aliasing, or other techniques that would increase the number of breaks (changes in color) in a given horizontal line. Vertical gradients do not increase the file size as much as horizontal ones do.

The most common embedded designs make use of 16-color modes. This is because all EGA/VGA devices support them through standard interfaces that have existed for some time. Out of all the non-VESA VGA modes, 16-color modes are the highest resolution and have been supported for the longest time. In the future, VESA support may be added, in which case 256-color mode may be supported at higher resolutions.

While there is no guarantee that a VGA card will support VESA, all VGA cards *must* support 640x480, 16-color mode, or they are no longer VGA compatible. Although VGA cards must also support the 320x200, 256-color mode, it is less desirable due to its lack of resolution.

Most EGA/VGA, 16-color modes use a palette of 16 colors selected from a total of 64 different hues. The 64 colors are selected using two bits each for red, green, and blue. This means that you will have four shades of red, green, and blue scaling up in 33 percent increments from no intensity (off) to full intensity. All the combinations of these are then available for use in the palette. Note that some colors may be significantly altered when translated to the hardware palette. This is particularly true of colors that use 50% red, green, or blue, since the rounding error is extreme for these cases in 16 color modes.

Unfortunately, since Windows provides eight bits for red, green, and blue for every entry in a plaette, you must keep in mind that your palette may be changed significantly when it is actually used. If you do not keep this in mind when designing your palette, you may notice some strange discrepancies between the colors you intended to have displayed, and the colors that are actually displayed at run time.

With the above as a technical background, here is a simple procedure for creating splash screen images using PhotoShop:

1. Create the image you wish to use as a splash screen. Place your logos and text, select your colors, and keep it simple. Once you finish creating the original splash screen, save it. Be sure to keep the original under a different name from your working copy, because you will want to open both the original image and the working copy at the same time in later steps.

2. On your working copy of the image, use flood fill or color adjustment to change your colors so they are significnatly different from each other. It's a good idea to be sure that the different colors in the working copy constrast sharply with each other before going to step 3, otherwise PhotoShop may anti-alias the image using similar colors. Don't worry if the colors you use for sharp contrast are the colors you want (or like) since you will be changing them back later.

3. On your working copy, click on "Mode" and select "Indexed Color". Under "Other", enter the number of colors you will be using plus one. Under "Palette" select "Adaptive" and under "Dither" select "None". Finally, click on [OK].

4. Open your original splash screen, and place it so it is visible when you select your working copy. Click on "Mode" and then on "Color Table". This will bring up a dialog box that will show all the colors currently used in the image. Select each entry in turn and use the color picker to adjust the palette entries in the color table, so that your working copy begins to resemble the original. When adjusting palette entries, be sure that the R, G, and B entries are 0, 85, 170, or 255. This will result in the colors in your color table turning out the closest to the final colors displayed on the splash screen (if your target is in 16-color mode).

5. Clean up the final image. In step 3, PhotoShop will have set some random pixels on diagonal lines and color boundaries in an attempt to reproduce something closer to the original image. You may need to clean these up so that the final image is less jagged. Of course, this only happens if you used anti-aliasing when designing the original art, so this step can be eliminated if you avoid anti-aliasing early-on in the process.

6. Save your final image as a Windows .BMP file. Select four bits per pixel (if you used less than 16 colors). Do not select RLE compression. Your images must be uncompressed .BMP files without RLE encoding. GSMERGE will convert your file to its own, internal RLE format, which incidently does not match the format of a Windows RLE-encoded .BMP file.

7. From the command prompt, you can run the CVTBMP utility to view your output as it will be displayed by the graphical POST system, before trying to create an IDF file that includes it. This utility works under DOS, and also in a full-screen DOS box under Windows. CVTBMP accepts two parameters. The first parameter is the name of the .BMP file with the .BMP extension explicitly provided. The second parameter is the name of a scratch file (you may choose to create an .RLE extension to remind you) that CVTBMP will write the actual compressed resource file just as GSMERGE would convert it to. This allows you to see just how large the file will be, for ROM space budgeting. Here is an example of invoking CVTBMP from the DOS prompt:

```
C> CVTBMP  splash1.bmp  splash1.rle
… graphic displays here; press space bar to exit graphic display …
C> DIR  splash1.rle
… display of size of splash1.rle here …
```

10.2.4.4 Creating Timed Sequences

Displaying an animated sequence of frames requires a delay to be inserted between the frames, so that the end user has a chance to see each frame before the next one is displayed. These delays can be specified in the SPLASH_TABLE by inserting special entries with a resource ID that is between 0FF00h and 0FFFFh. An example of this is shown in the table above, where two different graphics images (**RESOURCE_ID_ADVERT4** and **RESOURCE_ID_ADVERT5**) are displayed in response to **EVENT_SPLASH_MSG3**, but separated by a delay of 10h timer ticks (approximately 16*55ms=880ms, or about 9/10 of a second).

10.2.4.5 Configuring the Graphics Driver Software

The graphical POST system contains a graphics driver subsystem that is configurable with parameters in the project file. The following parameters may be tuned (see Chapter 7 for more detail) to support the exact hardware to be used in the embedded design.

Enable the system by enabling **OPTION_SUPPORT_SPLASHSCR**. This system requires that you also enable **OPTION_SUPPORT_EXTRES**, so that the External Resource Manager can find your graphical bitmaps as resources “external” to the 16-bit core BIOS compilation.

Use the **CONFIG_SPLASH_WIDTH** and **CONFIG_SPLASH_HEIGHT** to specify the dimensions of the display device itself in pixels and scan lines, respectively. These dimensions should not be confused with the frame buffer width, specified using the **CONFIG_SPLASH_WBYTES** parameter. Thus, it is possible to have a display device connected to a video controller that has a much wider frame buffer than can be displayed by the device.

The **CONFIG_SPLASH_COLORS** parameter is used to determine how to interpret the graphical resources. Only 16-color and 256-color modes are supported at this version. Do not attempt to set this parameter to values other than those supported.

The **CONFIG_SPLASH_SEG** parameter provides a way for the OEM to redefine the scratch segment used by the display routines when decompressing the RLE-encoded images. Normally, this is not necessary, but could become an issue with systems that have limited low memory (i.e., less than 640KB of low memory).

The **CONFIG_SPLASH_BOOTS** parameter provides a way for the OEM to limit the number of boots for which the graphical POST can be disabled in the setup screen system until it is restored again.

Chapter 11

PCI Subsystem

EMBEDDED BIOS provides support for managing Peripheral Component Interconnect (PCI) busses, devices, and bridges in an embedded system. This Chapter discusses the functions of the PCI subsystem and how it is configured from the project file and from the Board Personality Module.

11.1 Overview

EMBEDDED BIOS provides support for configuring and initializing devices on a PCI bus, and provides the industry standard PCI API functions for use by operating systems and device drivers.

During POST, a process called PCI enumeration scans the PCI bus hierarchy for PCI devices and PCI-PCI bridges. During PCI enumeration, each device and bridge is queried to determine the resources that it is requesting. A second pass through PCI bus scanning then allocates and maps memory, I/O, and IRQ resources so that the devices and bridges can perform their functions. As each device is configured and enabled, it is initialized so that it may begin servicing its device(s). As each bridge is configured and enabled, it provides the gateway through which other PCI devices and bridges may be visible on the other side of the bridge.

During the system’s steady-state operation, EMBEDDED BIOS can provide 16-bit and 32-bit services callable by application programs and operating systems for external management of PCI devices and bridges. The 16-bit services are built directly into the 16-bit core BIOS image. The 32-bit services are built as a separate component, using a 32-bit assembly and 32-bit linker. The 32-bit services module presents its services through another 32-bit component, the 32-bit BIOS directory service, which is used by 32-bit applications and operating systems to locate 32-bit BIOS services in general. The 16-bit services are always built when the PCI feature is enabled in EMBEDDED BIOS; however, the 32-bit services are optional and need only be included in the final binary on the target if applications or operating systems require them.

Configuration of PCI support in EMBEDDED BIOS is managed in two ways: build parameters and tables in the project file, as well as tables and custom code in the Board Personality Module. Build parameters help to guide the policies used during PCI enumeration for assigning resources to PCI devices. Tables in the project file are used to define external ROM images, combined

with the BIOS image, that support embedded PCI devices without on-board option ROMs. A table in the Board Personality Module describes the IRQ routing used in the board's design (barber pole issues, etc.) Optional code in the Board Personality Module allows the OEM to change the standard behavior of the PCI subsystem to meet the needs of an embedded design.

11.2 PCI Services

PCI services are made available to callers running in 16-bit real mode (or virtual 86 mode), 16-bit protected mode (using the same 16-bit code path) and 32-bit protected mode. These services provide functionality for reading and writing the PCI configuration space for a given bus, device, and function on the device in units of bytes, 16-bit words, or 32-bit doublewords. Additional services are provided to find a PCI device in the bus hierarchy, find a PCI device by its class code (i.e., video output device), and manage hardware interrupt routing.

The 16-bit PCI services are callable by 16-bit applications through the INT 1Ah function B1h. These services are not designed to be called from a 32-bit application or operating system. Instead, a separate 32-bit interface is available for 32-bit callers. This API is defined in the PCI Specification version 2.1.

The 32-bit PCI services are not handled by the 16-bit INT 1Ah function B1h. Instead, 32-bit callers can obtain the address of the 32-bit PCI services entry point by scanning the BIOS image from physical address E0000h to physical address FFFFFh, looking for "_32_" aligned on a 16-byte boundary. This method conforms to the industry standard, "32-bit BIOS Directory Services Specification".

Documentation for both PCI service APIs is widely-available; however, the PCI Specification is the original, official source.

11.3 The 32-Bit PCI Build Process

The PCI API supports both 16-bit and 32-bit callers. This introduces complexity to the system, in the source code, the build process, and at run time. Some OEMs may not require 32-bit PCI services; environment variables may be defined to disable or control the build process to eliminate these extra services. In order to provide entry points for 16-bit and 32-bit callers, two separate builds of source code are required which are merged together in a final step. Some source code is shared between both builds, and certain restrictions apply to coding methods in this dual-use code. The 16-bit build is performed first to build the majority of the BIOS and the 16-bit entry points for the PCI API. Typically, the 32-bit build is then performed, and the GSMERGE utility is invoked automatically by GSMAKE to merge the 32-bit BIOS components such as 32-bit PCI and 32-bit BIOS directory services with the 16-bit image, resulting in a final image to be used to program the BIOS boot ROM. Either Borland or Microsoft tool sets can be used to build the 16-bit and 32-bit components; however, they may not be mixed.

The 16-bit build produces a raw image of the 16-bit core BIOS with a name of the form, *proj*.ABS, where *proj* is the name fo the project (for more background on this process, see Chapters 4 and 5. Versions of EMBEDDED BIOS prior to 4.3 only build the *proj*.ABS file, which contained the entire image to be used to program the BIOS boot ROM.

An OEM may not require 32-bit PCI services if the target platform or operating system does not utilize them. Alternatively, the 32-bit binary may already exist and not require rebuilding. In these cases, the 32-bit build may be disabled by defining the **NOPCI32** environment variable prior to running GSMAKE for a given project. The actual contents of the environment variable

are not important, but by convention "YES" or "Y" is used to make the sense of the variable clear. The following command clears the environment variable, allowing the 32-bit PCI build to take place:

```
C> SET NOPCI32=
C>
```

The following command sets the environment variable, disabling the 32-bit PCI build:

```
C> SET NOPCI32=YES
C>
```

The GSMERGE utility is still executed to perform any functions that may be defined in the `.IDF` file, also located in the project directory. GSMERGE may be required to merge other components with the 16-bit core BIOS, including VGA binaries, OEM ROM extensions, splash screens, or other resources. If the `.IDF` file does not exist, then GSMERGE does not run. The OEM can force GSMERGE to ignore a `.IDF` file and not run at all by setting another environment variable, **NOGSMERGE**, to a non-null value. Thus, for purposes of building 32-bit PCI, setting **NOGSMERGE** also effectively implies **NOPCI32**. The actual contents of the environment variable are not important, but by convention "YES" or "Y" is used to make the sense of the variable clear. The following command clears the environment variable, allowing the GSMERGE portion of the build to take place:

```
C> SET NOGSMERGE=
C>
```

The following command sets the environment variable, disabling the final merge portion of the build:

```
C> SET NOGSMERGE=YES
C>
```

If the 32-bit build is performed, the PCIAPI32.DLL file is created (in Portable Executable format). The GSMERGE utility reads the `.IDF` file (Image Definition File), which contains statements that cause GSMERGE to read the PCIAPI32.DLL file and merge it into the final output file. In typical adaptations, the output from this process is the `.BIN` file, containing the patched `.ABS` image, the 32-bit BIOS directory services, and the 32-bit PCI services.

The 32-bit binary may already exist because of a prior build, or may be provided by General Software, one of its technology centers, or a third party. Since the 32-bit build requires more build tools that may need to run in a 32-bit environment such as Windows 95/98/NT, not all workstations may be able to execute them or have the tools installed. In this case, the `PCIAPI32.DLL` and `PCIAPI32.MAP` files (for example) should be placed in the project subdirectory and the NOPCI32 environment variable should be defined. When GSMAKE is subsequently executed, the 16-bit code will be updated and the provided DLL and MAP files will be used by GSMERGE without recompilation.

Some source files are shared between the 16-bit and 32-bit builds. This includes some of the Board Personality Module and Chipset Personality Module routines. The shared routines are kept in separate `.ASM` files which are included in the 16-bit and 32-bit board and chipset module builds. The dual-build routines for the BPM are kept in a file named `BRD1632.ASM` in the BPM's source directory. Similarly, the dual-build routines for the CSPM are kept in a file named `CS1632.ASM` in the CSPM's source directory. These dual-build files are subject to additional restrictions so that the code they contain can build correctly in both the 16-bit and 32-bit build

environments. This code must not make segment references or assume CS has any attribute besides execute-only. Among other things, this means that look-up tables must be used with care. The **BIOS_16** and **BIOS_32** symbols are defined for the respective build environments, so that the code may make use of conditional assembly by testing these symbols with IF statements. The **GSMERGE_LINKAGE** macro may be used to patch a 32-bit portion of the image with addresses to stored data in the 16-bit image.

11.4 Configuring PCI in the Project File

PCI support in general is enabled in the project file by setting **OPTION_SUPPORT_PCI**. Normally, **OPTION_SUPPORT_BIOS32** should also be enabled, to support the 32-bit BIOS directory services so operating systems may gain access to 32-bit PCI services. Setting both these options enables all the code paths to make PCI functional in the BIOS.

The PCI subsystem can be directed to display a table of PCI devices and resource assignments during POST; this display is enabled by setting **OPTION_SUPPORT_PCI_POSTMSGS**.

If the PCI subsystem encounters hard errors during POST, the BIOS can be configured to continue or halt. Setting **OPTION_HARDERR_PCI** causes the hard error to be fatal, whereas clearing it allows the POST process to continue, despite hard errors. The other PCI configuration parameters in the project file are qualitative.

The **CONFIG_PCI_ROM_MAP** parameter specifies the starting addres of an area in the address space that can be used by the PCI ROM scan code to map option ROMs temporarily into the address space while copying them to their final destination below the 1MB address marker. For systems that have preassigned uses for the memory address space at the top of physical memory, this allows the OEM to reconfigure this temporary location.

The **CONFIG_PCI_MEM_AVAIL** parameter defines the first physical address that is allocated to devices that are requesting physical memory address space as a resource for runtime purposes. The default should work for nearly all systems, but it is possible that your chipset may impose restrictions on the area where this address space starts.

The **CONFIG_PCI_IO_PORT_BASE** parameter defines the first I/O address that the PCI subsystem may use to allocate to devices that are requesting physical I/O address space as a resource for runtime purposes. Unlike physical memory allocation, I/O addresses go down as they are allocated; **CONFIG_PCI_IO_ALLOC** is subtracted from **CONFIG_PCI_IO_PORT_BASE** to obtain the next available I/O address range for allocation.

The **CONFIG_PCI_ROM_SHADOW_START** parameter defines the first location in the ISA space below the 1MB address marker where PCI option ROMs may be shadowed; normally, this starts at C000h, but can be adjusted for special memory map considerations.

The **CONFIG_PCI_IRQ_BITMAP** parameter is a 16-bit mask that specifies which system IRQs are available for PCI use. Although any IRQ from 0 to 16 may be assignable to PCI INTx# lines, in practice chipsets only support a limited subset. Of that subset, certain IRQs may be further restricted because they are used for other purposes in the design, so they must be made unavailable to PCI. The OEM should specify the IRQs that are truly available to the PCI subsystem for assignments to PCI devices during POST, by setting the corresponding bits in this bitmask, so that the PCI subsystem does not allocate reserved IRQs. Up to four, but no more, of these IRQs are used by the PCI subsystem, because only INTA#, INTB#, INTC#, and INTD# are supported in the hardware.

Additional configurable parameters allow the OEM to adjust temporary workspace addresses used within the PCI subsystem; these parameters are brought out to eliminate hard-coded constants and allow for special memory map considerations during POST. These parameters are discussed in detail in Chapter 7.

One special project file table, **PCI_ROM**, is not actually a parameter. It allows the OEM to specify the physical addresses of PCI option ROMs to be associated with a specific bus, device, and function. This feature should (and in fact must) be used for all PCI option ROMs that need to run, but which are not detectable by the PCI subsystem when it enumerates the PCI devices. Thus, while a common PCI VGA or SCSI card inserted into a real PCI slot would have an option ROM detectable during enumeration because the PCI device header indicates this, an embedded VGA or SCSI controller might require an external ROM that needs to be specified with **PCI_ROM**. It should be noted that, unlike ISA ROM extensions, the copy of the PCI option ROM as specified using this parameter must not be visible in the ISA ROM scan range, or it will be accidently executed by the ISA ROM scan range as an ISA ROM, and because it is a PCI option ROM, it will not be executed in the proper context (shadow RAM enabled; bus, device, and function number passed in as parameters, etc.). For detailed information about the **PCI_ROM** table, consult Chapter 7.

11.5 Configuring PCI in the Board Personality Module

In general, the Board Personality Module contains code and data that implements board-level policy for all components of the BIOS, including 16-bit and 32-bit PCI. PCI policy includes how PCI interrupt lines are routed among the slots and devices, and the mechanism by which IRQs are assigned to INT#A, INT#B, INT#C, and INT#D by the chipset.

11.5.1 PCI Interrupt Routing Table

In order to support assignment and reassignment of PCI IRQs, operating systems such as Windows 98 need to know how the system board has wired each PCI slot's interrupt pins to the PCI Interrupt Router's interrupt pins. This information is obtained by the operating system from the PCI Interrupt Routing Table, implemented in the Board Personality Module of an EMBEDDED BIOS adaptation because it describes board-level policy.

Each PCI system board consists of one or more slots and a PCI Interrupt Router (a component of the PCI bus controller). Embedded devices that are not plugged into slots can be thought of as taking their own "slot" as well, for purposes of this discussion. Each slot has four interrupt pins, known as INTA#, INTB#, INTC#, and INTD#. The PCI Interrupt Router has several interrupt pins, known as PIRQ1#, PIRQ2#, PIRQ3#, … PIRQn#. There is no PIRQ0#. The INTn# pins for each slot may be wire OR'd with other INTn# pins from the same or other slots, and these groups of pins may also be connected to a PIRQn# pin on the Interrupt Router. The actual PIRQ value assigned to each interrupt pin on each Interrupt Router is assigned by the PCI resource allocation code in the EMBEDDED BIOS PCI Subsystem.

The PCI IRQ routing information is stored in a table, defined in the Board Personality Module's shared 16-bit/32-bit source file called BRD1632.ASM. The bit mask of system IRQs to use for PCI is also defined and typically used by this table.

Here is an example of the board PciIrqTbl and PCI_IRQ bit mask definitions for a hypothetical board. It shows a table that might be build which actually matches the standard barber-poling method for PCI INT line routing. In practice, this table would not be necessary since the PCI enumeration code defaults to this routing method. However, it provides a useful example for

reference. If you have a board with nonstandard PCI interrupt assignments, or one with embedded devices that have specific interrupt lines routed to them, then you should start with this as a first pass, and modify as appropriate for your own custom table.

```
        IF      OPTION_SUPPORT_PCI

;       Evaluation Board wiring (from schematic)
;
;       Device PCI INT line ->:         INTA     INTB     INTC      INTD
;       Device
;       Slot 3 (device 12h):          SYSINTA, SYSINTB, SYSINTC, SYSINTD
;       Slot 4 (device 13h):          SYSINTB, SYSINTC, SYSINTD, SYSINTA
;       Slot 5 (device 14h):          SYSINTC, SYSINTD, SYSINTA, SYSINTD
;       Onboard (device 11h):         SYSINTD,   N/A,     N/A,     N/A
;
;       Thus Slot3 (SLT3 in silkscreen on the board) has its
;       INTA* pin tied to the system board PCI INTA* signal, but SLT4 has its
;       INTA* pin tied to the system board PCI INTB* signal, and so on.

;       Bits that are set in CONFIG_PCI_IRQ_BITMAP indicate the associated
;       IRQ is available for use by PCI.
;
;       In most cases, IRQ14 and IRQ15 must be reserved for the IDE controller.
;       IRQ13 is reserved for the FPU, and IRQ's 0-2, 6, and 8 are reserved by
;       various timers, keyboard, etc.  IRQ12 is typically reserved for the mouse.

;       No more than 4 IRQs need be in this list, assuming no conflicts exist
;       with the IRQ bitmaps provided in the table.  If fewer are available, PCI INT
;       lines may share a system IRQ, which may result in higher interrupt latency.

;       PCI_IRQ contains the list (bitmap) of allowed PCI IRQs.

PCI_IRQ = CONFIG_PCI_IRQ_BITMAP                     ; allowed IRQs for PCI use.

;       The PciIrqTable is only defined once, in the 16-bit case.  The 32-bit
;       code is patched with the start and end addresses of the table by GSMERGE.

        IF      BIOS_16

;       Define the link values for the PCI INT routing on the system board.

SYSINTA = 1
SYSINTB = 2
SYSINTC = 3
SYSINTD = 4

PciIrqTbl label byte

;        Bus 0, Device 12h, Slot 3.

PCIIRQENT 0, 12h,\
          SYSINTA, PCI_IRQ,\          ; Slot 3 PCI INTA routes to system board PCI INTA.
          SYSINTB, PCI_IRQ,\
          SYSINTC, PCI_IRQ,\
          SYSINTD, PCI_IRQ,\
          3

;        Bus 0, Device 13h, Slot 4.

PCIIRQENT 0, 13h,\
          SYSINTB, PCI_IRQ,\          ; Slot 4 PCI INTA routes to system board PCI INTB.
          SYSINTC, PCI_IRQ,\
          SYSINTD, PCI_IRQ,\
          SYSINTA, PCI_IRQ,\
          4

;        Bus 0, Device 14h, Slot 5.

PCIIRQENT 0, 14h,\
          SYSINTC, PCI_IRQ,\          ; Slot 5 PCI INTA routes to system board PCI INTC.
          SYSINTD, PCI_IRQ,\
          SYSINTA, PCI_IRQ,\
          SYSINTB, PCI_IRQ,\
          5
```

```
;       Bus 0, Device 11h, Ethernet controller.

PCIIRQENT 0, 11h,\
          SYSINTD, PCI_IRQ,\
          SYSINTA, PCI_IRQ,\
          SYSINTB, PCI_IRQ,\
          SYSINTC, PCI_IRQ,\
          0FFh

PCI_TABLE_INFO_SIZE = ($-PciIrqTbl)  ; Size of table in bytes.
PciIrqTblEnd label byte              ; Public end of table.
        ENDIF   ; (BIOS_16)          ; (Define table only once.)
        ENDIF   ; (OPTION_SUPPORT_PCI)
```

11.5.2 Board Personality Module Routines

The interrupt routing policy is handled by an implementation of the PCI Interrupt Routing Table, and the **BoardGetPciInfo** BPM function, which uses the table. See Chapter 20 for the definition of this BPM function.

The **BoardAssignPciIrq** BPM function is called from the PCI subsystem in the core BIOS to map a system IRQ level to a PCI interrupt line by programming the hardware (typically, the Edge-Level Control register of the PIC as well as the chipset's PCI interrupt steering control registers). As part of its work, it calls **CsAssignPciIrq** to perform the chipset programming.

> *Hint: Normally, it should not be necessary for the OEM to override the **CsAssignPciIrq** function, but if it does become necessary, it is better to comment out the call to the CSPM function, and in the place of the call, insert the new code. This preserves the integrity of the CSPM and keeps all policy in the BPM.*

Two additional BPM functions, **BoardPciReadScratch** and **BoardPciWriteScratch**, are used by the PCI subsystem to save information about the configured system (such as the discovered number of busses in the system) for later use by API functions. This information must be saved in a place where it can be retrieved even by protected mode code that may not have direct access to RAM save areas under control of the BIOS such as the BDA and EBDA. The default versions of these routines use CMOS cells to save this information; however, if no CMOS is present in the system, the OEM must redefine these routines in the BPM to save the information elsewhere.

11.5.3 Chipset Personality Module Routines

As mentioned earlier, the PCI subsystem calls BPM functions to manage PCI policies, and BPM functions may call CSPM functions to perform chipset-specific work. OEMs implementing new CSPMs must implement routines **CsAssignPciIrq** and **CsGetPciInfo** to enable the PCI subsystem to correctly perform its work. For more information about these functions, consult Chapter 19.

Chapter 12

DISK FILE SYSTEM MANAGEMENT

EMBEDDED BIOS provides support for file system mass storage (disks and their emulators) through the File System Control Layer (FSCL) and File System Drivers (FSDs). This chapter presents the overall architecture of this software, documents how it interacts with the rest of the system through architected programming interfaces, and then discusses the practical aspects of configuring mass storage devices in the project file.

In addition to the standard PC floppy, IDE, ATA, and CD-ROM devices, EMBEDDED BIOS provides solid-state emulation of disk drives, both hard disk partitions and floppy diskettes, with file system drivers supporting various media organizations. The ROM disk file system driver provides read-only disk I/O services over media managed by the **Rom** MTD. The RAM disk file system driver provides read/write disk I/O services over media managed by the **Ram** MTD. And the Flash disk file system driver provides read/write disk I/O services over NOR Flash MTDs (several types are supported). This Chapter explains the uses and tradeoffs of using these disk emulators, and describes the procedures to define and troubleshoot them.

12.1 File System Control Layer

The File System Control Layer (FSCL) provides INT 13h disk device I/O services for client software, including operating systems, application software, and BIOS components such as Manufacturing Mode.

All FSCL clients request disk I/O services through the INT 13h BIOS software API, as described in Chapter 21. FSCL in turn routes I/O requests for specific disks to the appropriate File System Driver (FSD) associated with the disk file system.

FSCL hides the actual implementation of file systems, and presents them as floppy diskettes or hard disk drives. The underlying media may be real disk drives; or emulators using memory technologies such as RAM, ROM, or Flash; or network drives.

EMBEDDED BIOS provides standard FSDs for real floppy disk drives, real IDE/ATA drives, ATA “El Torito” bootable CD-ROM drives, ROM disk drive emulators, RAM disk drive emulators, and Flash disk drive emulators. The OEM can even extend the architecture with a user-defined file system without modifying the core source code.

Because all disk I/O requests are handled by the FSCL, one FSD (perhaps the User-defined FSD) may call other FSDs through the INT 13h interface. This makes it possible for an FSD to implement disk duplexing, mirroring, striping, or other logical tasks, transparently to the operating system or application software in the system.

12.1.1 FSCL Architecture

The FSCL architecture provides a way for file system drivers (FSDs), including those supporting floppy disks, IDE drives, ROM disks, RAM disks, Flash disks, and OEM-defined drivers, to participate in the system in a cooperative way. File systems can be mapped to specific BIOS unit numbers by the OEM using the SETUP screen system, transparently to the drivers themselves. FSCL initializes each participating file system during POST, and routes INT 13h I/O requests to the appropriate FSD, based on this BIOS unit mapping.

The architecture provides for each file system to provide access to multiple devices in the same class within the same system. This allows support for up to four real physical floppy drives, four real physical IDE drives, and a virtually unlimited number of ROM, RAM, and Flash disks.

The architecture also permits FSDs to support both soft-style (floppy format) and hard-style (hard disk partitioned) file system layouts. The purpose of this feature is to provide the OEM with a choice of floppy-format or partitioned ROM, RAM, and Flash disks, although the idea can be logically extended to treating real IDE drives as floppy units, and real floppy drives as partitioned media, all transparently to the operating system.

FSCL is responsible for receiving disk I/O requests from client software via INT 13h. FSCL interprets these requests and by comparing the drive number passed in the DL register with table entries in the **FILE_SYSTEM** table, routes the requests to the appropriate FSD. Some functions, such as 00h (reset) and 08h (get drive parameters) are handled in a special way by FSCL, since they involve controlling or returning information about the entire mass storage subsystem.

In order to perform its work, an FSD may require some specialized FS Helper services (FSHLP API) provided by FSCL. These services provide a unified way to manage conversions from cylinder/head/sector coordinates to 32-bit sector numbers, and to initialize table entries during POST.

12.1.2 File System Types

FSCL supports both floppy-like and hard disk-like devices. Floppy-like devices, with no partition table, are called Soft file systems, and are always presented to operating system and application software as drives numbered from 00h through 7fh. Soft devices have a Partition Boot Record (PBR) located in their first logical sector number (LSN) 0. The PBR contains the file system's logical geometry (such as information about FAT size and cluster size, etc.), and is written by the operating system's FORMAT utility (or its equivalent).

Hard disk-like devices, which are normally partitioned with an FDISK utility or its equivalent, are called Hard file systems, and are always presented to operating system and application software as drives numbered from 80h through ffh. Hard file systems have a Master Boot Record (MBR) located in their first logical sector number (LSN) 0. The MBR contains the partition table written by the operating system's FDISK utility (or its equivalent), and defines

where the disk's file system partitions are located. Each DOS-compatible file system partition contains a Partition Boot Record (PBR) in its first sector.

While FSCL can be configured to have IDE drives respond with Soft drive numbers or floppy drives respond with Hard drive numbers, this usage is obviously nonstandard and is discouraged. It is the operating system and its application software that implicitly assume that a disk device mapped to one of the above ranges has a certain file system layout.

In general, an FSD may support both Hard and Soft formatted devices, but this is not always the case. The Floppy FSD in the BIOS for example, only supports the Soft format, and the Ide FSD in the BIOS only supports the Hard format, in accordance with industry standards. The EMBEDDED BIOS CD-ROM file system driver, ROM Disk, RAM Disk, and Flash Disk FSDs support both Hard and Soft disk emulation.

12.1.3 FILE_SYSTEM Table

The functionality of FSCL is largely data-driven, based on a table created with the **FILE_SYSTEM** macro in the project file at BIOS build time.

The **FILE_SYSTEM** macro is used to define the specific file systems that will be supported in the system. As previously mentioned, a given FSD may support multiple file systems. These file systems, as defined by the **FILE_SYSTEM** macro, are then mapped to drives in the SETUP screen, according to the user's needs. Not all of the entries in the **FILE_SYSTEM** table need be selected by the user. Only those enabled will actually be initialized by FSCL. The **FILE_SYSTEM** table entries represent the possible file systems that the BIOS will support.

When FSCL receives INT 13h requests for a specific drive, they are routed to the FSD that is handling the file system for the drive. The dispatching mechanism indexes into the **FILE_SYSTEM** table to locate the FSD associated with the file system itself.

The file system table is specified in a tabular format with **FILE_SYSTEM** entries. Each line in the table specifies a new file system that is governed by a particular FSD. Each line contains exactly five (5) operands, as in the following *hypothetical example*:

```
;             Type  Device    Start Addr  Length      SETUP name (unique)
;             ----  ------    ----------  ----------  -------------------
FILE_SYSTEM   Soft, Floppy,           0h,         0h, "Floppy 0"
FILE_SYSTEM   Soft, Flash,    080000000h,    400000h, "4MB Flash Disk 0"
FILE_SYSTEM   Hard, Ide,              0h,         0h, "IDE Drive 0"
FILE_SYSTEM   Hard, Ide,              1h,         0h, "IDE Drive 1"
```

The first operand specifies the type of file system (soft or hard). Soft file systems are configured by the BIOS to respond as floppies to the operating system; that is, they are associated with unit numbers in the range 00h-7fh (bit 7 clear). Hard file systems are configured by the BIOS to respond as hard disks to the operating system; that is, they are associated with unit numbers in the range 80h-ffh (bit 7 set). Soft file systems are never partitioned, whereas hard file systems are always partitioned.

The second operand specifies the file system driver (FSD) to be associated with the file system. There is a set of standard FSDs provided in the core BIOS, and the OEM can add new FSDs if needed. The following is a list of built-in file systems supported by the core BIOS:

Floppy True floppy disk drives (360K, 1.2M, 720K, 1.44M, 2.88M)

Ide	IDE hard drives and relatives (ATA cards as well)
Cdrom	ATA CD-ROM drives with bootable “El Torito” CD-ROM media
Rom	ROM disk driver (read-only, sectors direct-mapped to memory)
Ram	RAM disk driver (read/write, sectors direct-mapped to memory)
Flash	Flash disk driver (read/write, sectors movable in memory)

OEM-defined file systems may be added in the system by assigning them a unique name (say, User), adding an entry in the **FILE_SYSTEM** table with that name, and then naming the entrypoint of the new file system according to the naming conventions described in the chapter on File System Drivers.

The third operand identifies the location of the underlying media for the file system, to the FSD. For FSDs that emulate drives with memory (ROM, RAM, or Flash disks), the starting media address of the memory array is specified here. This is illustrated in the example above with the entry for a Flash file system called "4MB Flash Disk 0", which starts at media address 80000000h.

For FSDs that need to identify physical equipment, this field may be divided into several bitfields. For IDE drives, bit 0 indicates whether the physical drive is a master or slave device, and bit 1 indicates whether the controller I/O base address is 1f0h (0) or 170h (1). For Floppy drives, this field is simply the floppy drive unit number, from 0 to 3.

The fourth operand provides additional information about the file system to the FSD, and this information is FSD-specific. For FSDs that emulate drives with memory (ROM, RAM, or Flash disks), the size of the memory array is specified here in bytes. In the example above, the 4MB Flash Disk is assigned a length field of 400000h, or 4MB.

For FSDs that identify physical equipment like floppy disks and IDE drives, this field is not used.

The fifth operand is the human-readable name assigned to the file system, for purposes of display in the SETUP screens and in operator prompts (such as when the user is prompted to verify an RFD, for example). This name should not exceed 16 characters, or the SETUP screen may not be displayed correctly.

Entries in the **FILE_SYSTEM** table are assigned a table index, which is then used when mapping drives with **CONFIG_CMOS_ASSIGN_*x***, where ***x*** ranges from A: through K:. The BIOS build automatically numbers the Soft file system entries first, from 1 through however many there are. Then, Hard file system entries follow that number. So in our above example, there are four entries, numbered 1 through 4. If the Hard entries had been declared first, or interspersed between the Soft entries, the two Soft entries would still be assigned indexes 1 and 2, and the Hard entries would be assigned indexes 3 and 4, due to this automatic sorting during the BIOS build process.

Each **FILE_SYSTEM** table entry is compiled into an instance of the **FS_BASE** structure. Those **FS_BASE** structures with type Soft are stored in a table labelled **FsSoftTbl**; those with type Hard are stored in a table labelled **FsHardTbl**. The number of entries in each table are automatically defined by the symbols **FsSoftCount** and **FsHardCount**, respectively. These structures and values are copied to the Extended BIOS Data Area during POST.

12.1.4 FSCL Data Structures

FSCL uses data structures to maintain the definitions for file systems and their associated FSDs. The user need not be concerned with these structures as they are hidden from the INT 13h interface. OEMs using existing FSDs in their designs need not be concerned with these structures since they are managed by FSDs already written. FSD implementors, however, need to understand these structures since the FSD is responsible for initializing them and receiving parameters from them to perform work.

12.1.4.1 FS_BASE Structure

The **FS_BASE** structure (defined in file INC\STRUC.INC) is compiled into the BIOS from the parameters specified in the **FILE_SYSTEM** macro. It contains the following members:

FsbCall - A 16-bit near pointer, with respect to **BIOS_GRP**, to the File System handler entrypoint associated with a File System Driver. The naming convention for File System entrypoints is ***_p*_FileSys*Xxx***, where "*Xxx*" is the "device" field in the **FILE_SYSTEM** macro.

FsbName - A 16-bit near pointer relative to **BIOS_GRP** containing the ASCIIZ-printable name of the device (from the "name" field in the **FILE_SYSTEM** macro).

FsbAddr - A 32-bit value derived from the "startaddr" field of the **FILE_SYSTEM** macro.

FsbLen - A 32-bit value derived from the "length" field of the **FILE_SYSTEM** macro.

12.1.4.2 FS_UNIT Structure

The **FS_UNIT** structure (defined in file INC\STRUC.INC) is built by FSCL in RAM, in the Extended BIOS Data Area, for each drive properly configured in the Setup system.

As can be seen, it is derived from the FS_BASE structure in ROM, and adds additional fields calculated by the FSD at initialization time. It contains the following members:

FsuCall - A 16-bit near pointer, with respect to **BIOS_GRP**, to the File System handler entrypoint associated with a File System Driver. The naming convention for File System entrypoints is ***_p*_FileSys*Xxx***, where "*Xxx*" is the "device" field in the **FILE_SYSTEM** macro. This field is copied by FSCL from the **FS_BASE** structure.

FsuName - A 16-bit near pointer relative to **BIOS_GRP** containing the ASCIIZ-printable name of the device (from the "name" field in the **FILE_SYSTEM** macro). This field is copied by FSCL from the **FS_BASE** structure.

FsuAddr - A 32-bit value derived from the "startaddr" field of the **FILE_SYSTEM** macro. This field is copied by FSCL from the **FS_BASE** structure.

FsuLen - A 32-bit value derived from the "length" field of the **FILE_SYSTEM** macro. This field is copied by FSCL from the **FS_BASE** structure.

FsuCyls - A 32-bit doubleword specifying the number of cylinders on the device, for purposes of reporting to the operating system via INT 13h. This field must be initialized by the FSD during file system initialization.

FsuHds - A 16-bit word specifying the number of heads on the device, for purposes of reporting to the operating system via INT 13h. This field must be initialized by the FSD during file system initialization.

FsuSpTk - A 16-bit word specifying the number of sectors per track on the device, for purposes of reporting to the operating system via INT 13h. This field must be initialized by the FSD during file system initialization.

FsuSpCl - A 16-bit word specifying the number of sectors per cylinder on the device, as calculated by multiplying **FsuSpTk** by **FsuHds**. This field must be initialized by the FSD during file system initialization.

FsuStat - An 8-bit storage location reserved for use by the FSD. Its use is unarchitected.

FsuCtrl - An 8-bit storage location reserved for use by the FSD. Its use is unarchitected.

FsuOpt - An 8-bit storage location reserved for use by the FSD. Its use is unarchitected.

FsuByte - An 8-bit storage location reserved for use by the FSD. Its use is unarchitected.

FsuWrd1 - A 16-bit storage location reserved for use by the FSD. Its use is unarchitected.

FsuWrd2 - A 16-bit storage location reserved for use by the FSD. Its use is unarchitected.

FsuWrd3 - A 16-bit storage location reserved for use by the FSD. Its use is unarchitected.

12.1.4.3 FS_PACKET Structure

The **FS_PACKET** structure (defined in file INC\STRUC.INC) is built by FSCL in RAM, as a method of passing parameters to the FSD when calling the File System handler entrypoint. All members, except for FspWork, are initialized by FSCL before calling the FSD.

This structure contains the following members:

FspType - An 8-bit value specifying the type of call. During POST, the File System handler is called for device initialization with this field set to **FSCALL_INIT**. For normal INT 13h I/O requests, this field contains **FSCALL_HARD** or **FSCALL_SOFT**, depending on the setting of bit 7 of the DL register when the INT 13h call was received by FSCL.

FspFunc - An 8-bit value specifying a subfunction code. For normal INT 13h I/O requests, this field contains the value in register AH at the time of the INT 13h call. During POST, this field contains **FSCALL_HARD** or **FSCALL_SOFT**, as derived from the file system type specified in the **FILE_SYSTEM** macro (either Hard or Soft, respectively).

FspUnit - A 16:16 segment offset pointer to the **FS_UNIT** structure maintained in RAM by FSCL for the file system governed by the FSD.

FspNdta - A 16:16 segment offset pointer to the data transfer area. This pointer has been normalized so that the offset value will not exceed 16 (000fh). For **FSCALL_INIT** request types, this data transfer area can be used as a work area for reads and writes to determine the media's geometry.

FspCnt - A 16-bit value containing the number of sectors to transfer. For **FSCALL_INIT** request types, this value is set to 1 by FSCL.

FspLba - A 32-bit value containing the media logical block address (relative to 0), of the data to transfer. This value is computed by FSCL from FspCyl, FspHead, and FspSect below, using values in **FS_UNIT**). This field set to 0 for **FSCALL_INIT** request types.

FspCyl - A 32-bit value containing the 0-based cylinder number of the data to transfer. This field is set to 0 for **FSCALL_INIT** request types.

FspHead - A 16-bit value containing the 0-based head number of the data to be transferred. This field is set to 0 for **FSCALL_INIT** request types.

FspSect - A 16-bit value specifying the 1-based sector number of the data to be transferred. This field is set to 1 for **FSCALL_INIT** request types.

FspRegs - A 16:16 segment offset pointer to a StackReg structure, containing the general registers of the CPU as saved from the INT 13h call by FSCL. This pointer may be used to modify user-level registers as necessary to implement certain non-standard OEM INT 13h functions. This field is undefined for **FSCALL_INIT** request types.

FspWork - An 8-byte unarchitected field that may be used by the FSD as a temporary work area; provided however, that the contents upon entry to the FSD are undefined, and that the contents will not be preserved after the FSD returns to FSCL. This field is not retained across calls to FSDs.

12.1.5 FSHLP API

The FSHLP API provides a set of standard tools for FSD writers to simplify FSD design. Use of the FSHLP API functions over ad hoc methods allows FSDs to leverage existing working code to perform translation of cylinder/head/sector units to 32-bit sector numbers, and perform file system initialization.

While it is possible to write an FSD without calling the FSHLP API functions it would be best to use them, since these functions may be abstracted in future versions of the architecture, allowing FSDs that use them to gain functionality through improved architecture.

The FSHLP API functions are not callable directly by application programs or operating systems; they are always called from FSDs.

These API functions are only callable in real mode, and must preserve registers unless they are used to return values in specific cases. The carry flag (CY) is used to indicate either a successful or failing outcome from a function call, unless otherwise specified in the API definition.

FSDs should keep stack depth to a minimum. A suggested maximum amount of stack usage by the FSD is 64 bytes. Do not assume that it is acceptable to allocate data buffers or other such data structures on the stack in an FSD.

If you need to allocate memory for run-time use in the FSD, this should be done when the file system driver receives the **FsInit** call during POST for each file system that it governs.

12.1.5.1 FsHlpInit Function

The **FsHlpInit** FSHLP function may be called by the FSD to retrieve the logical geometry (number of cylinders, heads, and tracks) from an existing partition table or BPB record. It is called from the File System handler, normally during initialization, but may be called at any time.

FsHlpInit assumes that **FspNdta**[BP] is a far pointer to memory containing the first sector of the logical device. If **FspFunc**[BP] (during POST) or **FspType**[BP] (after POST) contains **FSCALL_HARD**, the memory area is assumed to contain the partition table, and the entries are scanned to determine the logical geometry o the device. Otherwise, the memory area is assumed to contain a DOS BPB and boot record, and the geometry is determined from that BPB.

In either case, the data are verified by the **FsHlpInit** function, and **FsHlpInit** returns CY set if the data do not appear to be valid enough to make a sound determination about the file system's geometry.

If the geometry can be determined, then the CY flag is cleared upon return, and the **FsuCyls**, **FsuHds**, **FsuSpTk**, and **FsuSpCl** fields in the **FS_UNIT** structure are updated with the new geometry for the file system, as determined from the memory data.

Input Parameters:

> SS:BP - 16:16 segment offset pointer to **FS_PACKET** structure as originally passed to the FSD's entrypoint in the SS:BP register pair.

Output Parameters:

> FsuCyls - Number of cylinders supported by the file system.
>
> FsuHds - Number of heads supported by the file system.
>
> FsuSpTk - Number of sectors per track supported by the file system.
>
> FspSpCl - Number of sectors per track supported by the file system.
>
> CY - Set if failure, else clear if success.

Unpreserved Registers:

> Flags.

12.1.5.2 FsHlpFind Function

The **FsHlpFind** FSHLP function may be called by the FSD to find all devices of a certain type (i.e., those governed by a specified File System Driver).

This feature is used by the Flash FSD to enumerate all of the file systems governed by the FSD when it is called by the SETUP screen's FORMAT RFD menu item. It could also be used by an FSD performing a logical function, such as drive mirroring or striping, to enumerate all file systems beneath it.

Input Parameters:

AX - A 16-bit near pointer to the File System handler entrypoint for the type of device to be searched for (relative to the BIOS_GRP group).

DL - The starting drive number to be scanned, from 00h to ffh.

Output Parameters:

DL - Drive number of matching file system.

SI - File system type (either **FSCALL_SOFT** or **FSCALL_HARD**).

DS:SI - A 16:16 segment offset pointer to the FS_UNIT structure for the matching device.

CY - Set if failure (no more devices or device not found), else clear if success.

Unpreserved Registers:

Flags.

12.2 File System Drivers

The EMBEDDED BIOS FSCL architecture provides for the flexibility of supporting many types of mass storage simultaneously, without the need for OEM-level customization of the core BIOS itself. This architecture is client-server based; with the operating system and application software being the clients of FSCL, and the FSDs being the servers that handle file system requests.

FSDs provide an element of abstraction in the BIOS, so that the underlying implementation of mass storage is not directly visible to the operating system or application program. ROM, RAM, and Flash disks appear just as their IDE and Floppy counterparts to the application software.

FSDs can easily be added to the system without reworking control paths and data structures in the BIOS, as would be necessary with other systems. FSDs are self-contained, simple to implement, and may even use other BIOS services, or DOS services if it can be determined that DOS is loaded in the system.

12.2.1 FSD Architecture

The purpose of an FSD is to provide a specific set of services for a class of mass storage, hiding the programming details of the mass storage media or device from the FSCL. FSDs are typically small, minimalistic, and are procedural, rather than focussed on details such as register manipulation.

FSDs only process requests in real mode, unlike their MTD counterparts. FSDs may of course call upon the services of MCL to access media, but MCL requests are always made in real mode, and the details concerning MTDs switching to protected mode and back are hidden by the MCL interface.

FSDs may take part in the system's overall poower management, if they are included in the power device tree table created with the **POWER_DEVID** macro. When specified in this table, FSDs must accept and handle power management requests through their corresponding power control entrypoint (see the **POWER_DEVID** macro definition for details).

Some FSD functions may not be valid operations for certain classes of mass storage. For example, the INT 13h functions specific to floppies are not available for hard drives, and vice versa. Generally speaking, the INT 13h functions that provide for initializing, reading from, and writing to mass storage are supported by all FSDs.

The application or operating system software initially requests a mass storage I/O operation via an INT 13h software interrupt request. This is handled by FSCL by generating a request packet and calling the entrypoint of the FSD associated with the device specified in the original INT 13h request. The processing is handled by the FSD, and then the FSD returns control to FSCL, which returns the results of the operation to the application or operating system software in the general register set.

12.2.2 FSD Entrypoint

Each file system driver in the system has an entrypoint, properly named to match the file system device name as specified in the **FILE_SYSTEM** table entries in the project file.

The naming convention established for FSDs is as follows: **_p_FileSys*Xxx***, where **Xxx** is the device name field in the **FILE_SYSTEM** macro. Thus, the entrypoint for the **Floppy** FSD is **_p_FileSysFloppy**, and the entrypoint for the **Ide** FSD is **_p_FileSysIde**. Similarly, the OEM could define a special **User** type as **_p_FileSysUser**. The **DefProc** macro actually automatically generates the **_p_** portion of the identifier, so that the OEM only sees the **FileSysXxx** in the code.

The entrypoint for the FSD is called with a 16-bit near CALL instruction within FSCL, and must return with a near RET instruction back to FSCL. Parameters are passed to the FSD entrypoint in an **FS_PACKET** structure pointed to by the SS:BP register pair. FSCL is responsible for converting the INT 13h call's register arguments into the **FS_PACKET** structure before calling the FSD, and translating the **FS_PACKET** structure's contents into register contents as output before FSCL returns to the INT 13h requestor.

The FSD should process each call by interpreting the request type as follows.

Processing FSCALL_INIT Requests

If the **FspType** field contains the value **FSCALL_INIT**, then the FSD should perform initialization functions relating to device initialization, and must set fields **FsuCyl**, **FsuHds**, **FsuSpTk**, and **FsuSpcl** in the RAM-based **FS_UNIT** structure. This **FS_UNIT** structure is

pointed to by **FspUnit** and DS:DI on entry. In addition to setting these mandatory fields, the FSD may also at its option initialize **FsuStat**, **FsuCtrl**, **FsuOpt**, **FsuByte**, **FsuWrd1**, **FsuWrd2**, and **FsuWrd3** for its own use in processing future requests.

During processing of **FSCALL_INIT**, the FSD may also need to acquire additional RAM to service requests associated with this file system. This is the time for the FSD to determine the top of available RAM by making the INT 12h BIOS call, then update the low-memory size in BIOS segment 40h as necessary to reserve any memory it might need.

Processing FSCALL_SOFT or FSCALL_HARD Requests

If FspType contains the value FSCALL_SOFT or FSCALL_HARD, then the FSD is being called to perform a normal INT 13h I/O requests. In this case, FspFunc contains the INT 13h register AH value.

Certain INT 13h functions are preprocessed by FSCL in a special way, as follows:

00h - **Reset**. The File System handler for each FSD is called in succession, so that each FSD in the system gets called once, and possibly more than once, as it may handle multiple file systems.

01h - **Get Status**. The File System handler is not called; the result of the last INT 13h operation of this type (Soft or Hard) is returned instead, in accordance with the industry standard definitions for reporting the status of floppy and hard drives.

08h - **Get Drive Parameters**. The File System handler is called. If the File System handler returns no error (CY clear), FSCL obtains values from the **FS_UNIT** structure and returns them to the user. For Soft devices, the File System handler is expected to return the CMOS type in register BX, and a far pointer to the DPT in register pair DX:CX. These concepts are reviewed in Chapter 3.

15h - **Get DASD Type**. The File System handler is called. If the File System handler returns no error (CY clear), FSCL obtains values from the **FS_UNIT** structure to return to the user.

Input Parameters:

SS:BP - 16:16 segment offset pointer to **FS_PACKET** structure, defined earlier in this chapter.

AH - A copy of the **FspType** field of the **FS_PACKET** structure for convenience.

SI - A copy of the **FspFunc** field of the **FS_PACKET** structure for convenience.

DX:CX - A copy of the **FspLba** field of the **FS_PACKET** structure for convenience.

ES:DI - A copy of the **FspUnit** field of the **FS_PACKET** structure for convenience.

DS - A copy of the ES register, for convenience.

Output Parameters:

AX - Returned to the caller of INT 13h. The AH portion of this register is preserved by FSCL in the system and is returned on subsequent calls to function **01h (Get Status**) calls for the same device class.

CY - Set if failure, else clear if success.

Unpreserved Registers:

AX, BX, CX, DX, SI, DI, DS, ES, BP, and Flags.

12.3 Floppy Disk Drive Support

Floppy disk drives are mostly mechanical, with no real controller on them at all. Instead, a Floppy Disk Controller (FDC) is supplied on the motherboard either as a stand-alone 82077/82078 part or as a part of the chipset or Super I/O companion chip.

Remember that "floppy support" does not refer to emulators, such as ROM, RAM, or Flash disks. Those are discussed in Chapter 12. In this Chapter, we are discussing the configuration of real physical disk drives.

12.3.1 Enabling Floppy Disk Support in the Build

The EMBEDDED BIOS floppy disk support is provided by the Floppy file system driver, enabled with the **FILE_SYSTEM** macro in the project file. See Chapter 7 for a detailed description of this macro. The **FILE_SYSTEM** macro does not define the only configuration of floppy disks in the system; it defines all the possible configurations. Thus, in full-featured system with four floppy disk drives, we would have the following table entries covering all four floppy disk drives:

```
FILE_SYSTEM Soft, Floppy, 0, 0, "First Floppy"
FILE_SYSTEM Soft, Floppy, 1, 0, "Second Floppy"
FILE_SYSTEM Soft, Floppy, 2, 0, "Third Floppy"
FILE_SYSTEM Soft, Floppy, 3, 0, "Fourth Floppy"
```

In the above example, the first parameter (Soft) indicates that the file system to be defined is floppy-like, and not formatted in the style of a hard disk partition. The second parameter indicates that the **Floppy** file system driver will govern the file system. The third parameter (0-3) specifies the unit number (in this case, floppies 0-3 attached to the FDC’s cable). The fourth parameter is not used for floppy disks and must be zero. The fifth parameter is an ASCII string inside quotes that specifies the name that is to be displayed in the Setup screen.

If your system will not ever use more than two floppy disk drives, then the last two table entries could be removed. If no floppy disks are to be supported in the system, then no **FILE_SYSTEM** entries specifying the **Floppy** file system driver should be specified.

12.3.2 Configuring Floppy Disks in Setup

While the build process uses the **FILE_SYSTEM** macro to specify which floppy disk drives are to be supported in the target, the Setup screen is used on the target itself to map the various drives to actual drive letters.

Remember that we used ASCII names to represent the various floppy disks with the **FILE_SYSTEM** macro. By going into the Basic Setup screen, the drive mapping section allows the user to select assignments for drives A: through K:. Only drives A: through D: support real floppy disks. Simply scroll through the possible assignments (made possible at build time), until the right ones are selected.

12.3.3 Tuning the Floppy Disk Driver

Most embedded systems have special needs when supporting floppy disk drives. For example, some systems have no DMA controller attached to the FDC, or have a nonstandard one connected to it. These systems require polled I/O, for example, and that is selectable with a configuration option.

The following configuration options can be manipulated in the project file to control how the floppy disk driver works:

OPTION_FLOPPY_82077	FIFO support
OPTION_FLOPPY_SEEK	Seek floppy at boot
OPTION_FLOPPY_WATCHIO	Display registers in/out of service routine
OPTION_FLOPPY_DMA	Select between DMA and polling
OPTION_FLOPPY_144_ONLY	Only allow 1.44 floppies in state machine
OPTION_FLOPPY_FAST_POLL	Improve performance with fast polling

12.3.3.1 82077 FIFOs

The 82077 option, if set, specifies that the floppy disk driver should enable the FIFO on the FDC. This allows more relaxed timing in the software, making a more reliable system.

12.3.3.2 Seek During Boot

The SEEK option, if set, enables the seek during POST before boot, common to many desktop systems. This is not really necessary in most embedded applications and only consumes extra boot time.

12.3.3.3 Debugging Polled I/O

The WATCHIO option, if set, enables a register dump before and after each disk I/O operation, so that the system can be debugged. This is useful when getting polling to work on new hardware that does not support DMA.

12.3.3.4 DMA or Polled Data Transfers

The DMA option, which is normally set, specifies that the FDC will use pseudo-DMA in conjunction with an 8237A DMA controller to perform device I/O without the CPU performing manual IN or OUT instructions. Without this mechanism, the FDC must be polled *very rapidly* by the CPU, meaning interrupts have to be disabled, and error checking may be limited. Several options (that follow) augment the polled approach, which is selected when DMA is disabled.

The 144_ONLY option, if set, indicates that the floppy disk driver state machine should not attempt to determine what type of diskette was inserted in the drive; instead, a 1.44MB disk is assumed. This is important if the CPU is operating slowly enough with respect to the FDC in a

polled situation that it simply can't handle the error condition that comes up when retrying the I/O with various floppy form factors.

The FAST_POLL option, if set, indicates that the floppy disk driver read/write path should be optimized for performance in a polled I/O situation so that the minimum number of conditional jumps are taken, at some expense in code size. This affords the best possible CPU performance as compared with the FDC data rate, so that polled I/O has the best chance of working.

Systems employing special DMA controllers, such as those on the Intel 386-EX CPU, may need to implement the **BoardInitDma**, **BoardEnableDmaCtrl**, **BoardDisableDmaCtrl**, and **BoardFloppyDma** functions to support floppy disk DMA in the hardware.

12.4 Hard Disk (IDE/ATA) Support

In contrast to floppy drives, IDE drives, or more generally ATA drives, are more electronic than mechanical. PC Card ATA drives, for example, rarely contain a spindle and instead are usually based on a Flash memory array managed by a microcontroller that offers a register-based interface to the host CPU. Whether the drive has a spindle or is memory-based, the ATA interface is active; that is, the host motherboard does not require a special controller to send commands via a protocol to the drive. Instead, the CPU sends commands and engages in protocol with the controller that is located on the drive itself.

Remember that "IDE/ATA support" does not refer to emulators, such as ROM, RAM, or Flash disks. Likewise, IDE/ATA does not refer to the "El Torito" CD-ROM emulation of a drive, described later.

12.4.1 Enabling IDE/ATA Disk Support in the Build

The EMBEDDED BIOS IDE/ATA disk support is provided by the **Ide** file system driver, enabled with the **FILE_SYSTEM** macro in the project file. See Chapter 7 for a detailed description of this macro. The **FILE_SYSTEM** macro does not define the only configuration of IDE/ATA disks in the system; it defines all the possible configurations. Thus, in full-featured system with four IDE disk drives, we would have the following table entries covering all four IDE disk drives, two on each controller:

```
FILE_SYSTEM Hard, Ide, 0, 0, "Master on 1F0h"
FILE_SYSTEM Hard, Ide, 1, 0, "Slave on 1F0h"
FILE_SYSTEM Hard, Ide, 2, 0, "Master on 170h"
FILE_SYSTEM Hard, Ide, 3, 0, "Slave on 170h"
```

In the above example, the first parameter (Hard) indicates that the file system to be defined is hard disk-like (partitioned), and not formatted in the style of a floppy diskette. The second parameter indicates that the **Ide** file system driver will govern the file system. The third parameter is partitioned into two bitfields for this driver. Bit 0 indicates whether the drive is master (0) or slave (1). Bit 1 indicates which controller will be used; either 0 for 1f0h or 1 for 170h. The fourth parameter is unused by the **Ide** FSD. The fifth parameter is an ASCII string inside quotes that specifies the name that is to be displayed in the Setup screen.

If your system will not ever use more than two IDE disk drives, then the last two table entries could be removed. If no IDE/ATA disks are to be supported in the system, then no **FILE_SYSTEM** entries specifying the **Ide** file system driver should be specified.

12.4.2 Configuring IDE/ATA Disks in Setup

While the build process uses the **FILE_SYSTEM** macro to specify which IDE disk drives are to be supported in the target, the Setup screen is used on the target itself to map the various drives to actual drive letters.

Remember that we used ASCII names to represent the various IDE/ATA disks with the **FILE_SYSTEM** macro. By going into the Basic Setup screen, the drive mapping section allows the user to select assignments for drives A: through K:. Only drives C: through K: support real IDE/ATA disks. Simply scroll through the possible assignments (made possible at build time), until the right ones are selected.

12.4.3 Tuning the IDE/ATA Disk Driver

Most embedded systems have special needs when supporting IDE/ATA disk drives. For example, many IDE drives support autodetection protocols and LBA transfers, but some do not. These options, and others, are selectable as configuration options in the project file.

The following configuration options can be manipulated in the project file to control how the IDE disk driver works:

OPTION_IDE_AUTODETECT	Autodetect drive geometry during POST
OPTION_IDE_CHS	Support CHS drive geometry translation
OPTION_IDE_LBA	Support LBA drive geometry translation
OPTION_IDE_POLLED	Support polled .vs. interrupt driven I/O
OPTION_IDE_DISABLE_INTS	Disable interrupts during disk I/O
OPTION_IDE_SLOWDOWN	Delay after each word transfer for I/O
OPTION_IDE_RESET	Reset drive during POST
OPTION_IDE_SEEK	Seek drive during POST

12.4.3.1 Drive Autodetection

The AUTODETECT option, when set, enables code that can automatically query IDE/ATA drives and determine the number of heads, cylinders, and sectors per track for the drive. These raw values may be augmented by a translation mechanism, either CHS or LBA, so that drives larger than 512MB may be used in a design. The dominant standard is LBA.

12.4.3.2 Drive Geometry Translation (LBA and CHS)

The CHS option, when set, enables code that supports the Phoenix formula for translation of the raw drive parameters. This is becoming less and less of a standard but is provided for compatibility.

The LBA option, when set, enables code that supports what has emerged as the dominant standard for drive geometry.

12.4.3.3 Polled .vs. Interrupt-Driven I/O Completion

The POLLED option, if set, causes the driver to poll the status register on a drive to wait for it to complete its I/O, rather than use an interrupt to signal I/O completion. Some embedded targets may reassign the standard interrupt (IRQ 14) for other purposes, requiring polled I/O to be used.

12.4.3.4 Disabling Interrupts During Transfers

The DISABLE_INTS option, when set, causes the driver to disable CPU interrupts around data transfers, so that application software cannot interrupt the progress of a sector transfer.

12.4.3.5 Slowing Down I/O Transfers

The SLOWDOWN option, when set, causes the driver to not use REP INSW or REP OUTSW instructions for data transfer, but instead uses a LOOP construct that reads or writes a word at a time, delaying between successive I/O instructions. This is provided for systems where the device emulating a hard drive cannot keep up with back-to-back I/O requests.

12.4.3.6 Drive Reset During POST

The RESET option, when set, causes the drive to be reset during POST. This consumes extra boot time and use is discouraged. Some drives may need to be reset during POST or they won't operate properly. The OEM can determine if this is necessary for the target in question, and enable or disable the option appropriately.

12.4.3.7 Drive Seek During POST

The SEEK option, when set, causes the drive head to be seeked during POST. This consumes extra time as RESET does, and is not really necessary except for making the system sound like a desktop machine when it boots.

12.5 "El Torito" CD-ROM Support

EMBEDDED BIOS supports bootable CD-ROMs as described in the "El Torito" BIOS specification. This specification defines a method by which an image of a floppy or hard disk can be stored in a prescribed area on the CD-ROM. "El Torito"-aware BIOS software can then autodetect these images and create drives that map to them. The built-in CD-ROM file system drive provides this drive emulation directly in the BIOS core, making it possible to use CD-ROM devices as boot devices in embedded applications.

12.5.1 Enabling CD-ROM Support in the Build

The EMBEDDED BIOS CD-ROM support is provided by the **Cdrom** file system driver, enabled with the **FILE_SYSTEM** macro in the project file. See Chapter 7 for a detailed description of this macro. The **FILE_SYSTEM** macro does not define the only configuration of CD-ROM drives in the system; it defines all the possible configurations. Thus, in full-featured system with four CD-ROM drives, we would have the following table entries covering all eight CD-ROM file systems (both soft and hard images supported on each of two drives per controller):

```
FILE_SYSTEM   Soft, Cdrom,001f00000h, 00E0003f6h, "CD Fl/Pri Master"
FILE_SYSTEM   Soft, Cdrom,001f00001h, 00E0103f6h, "CD Fl/Pri Slave "
FILE_SYSTEM   Soft, Cdrom,001700002h, 00E000376h, "CD Fl/Sec Master"
FILE_SYSTEM   Soft, Cdrom,001700003h, 00E010376h, "CD Fl/Sec Slave "

FILE_SYSTEM   Hard, Cdrom,001f00000h, 00E0003f6h, "CD Hd/Pri Master"
FILE_SYSTEM   Hard, Cdrom,001f00001h, 00E0103f6h, "CD Hd/Pri Slave "
```

```
FILE_SYSTEM   Hard, Cdrom,001700002h, 00E000376h, "CD Hd/Sec Master"
FILE_SYSTEM   Hard, Cdrom,001700003h, 00E010376h, "CD Hd/Sec Slave "
```

In the above example, the first parameter (Soft/Hard) indicates the type of bootable image that the file system will support (either floppy-like [Soft] or hard disk-like [Hard]). The second parameter indicates that the **Cdrom** file system driver will govern the file system.

The third parameter is partitioned into two bitfields for this driver. Bit 0 indicates whether the drive is master (0) or slave (1). Bit 1 indicates which controller will be used; either 0 for 1f0h or 1 for 170h. Bits 16-31 specify the I/O base address of the controller itself (commonly, 1f0h for the primary controller and 170h for the secondary controller).

The fourth parameter is partitioned into two bitfields for this driver. Bits 0-15 specify the secondary I/O base address of the controller itself (commonly, 3f6h for the primary controller and 376h for the secondary controller). Bits 16-31 should be set to E00h for the master drive and E01h for the slave drive, on either controller.

The fifth parameter is an ASCII string inside quotes that specifies the name that is to be displayed in the Setup screen.

Note that CD-ROM drives use similar controller addresses as IDE drives do; however, care must be taken to not enable a CD-ROM file system at the same address as an IDE file system (this is sorted out in Basic Setup). Also note that the third and fourth parameter fields (with hex numbers) are used differently by the CDROM file system driver and the IDE file system driver.

12.5.2 Configuring CD-ROM Drives in Setup

While the build process uses the **FILE_SYSTEM** macro to specify which CD-ROM drives are to be supported in the target, the Setup screen is used on the target itself to map the various drives to actual drive letters.

Remember that we used ASCII names to represent the various CD-ROM drives with the **FILE_SYSTEM** macro. By going into the Basic Setup screen, the drive mapping section allows the user to select assignments for drives A: through K:. Simply scroll through the possible assignments (made possible at build time), until the right ones are selected.

12.6 Emulating Disks With ROM

The EMBEDDED BIOS ROM disk provides emulation of a floppy disk drive by performing memory copies from a ROM image of a floppy disk or a hard disk partition. The ROM disk provides read-only operation; it does not allow emulation of writes or formatting.

The ROM disk is an ideal long-term storage solution for software that is not intended to be updated in the field. If updating is required, then the Resident Flash Disk (RFD) should be used with Flash components instead.

In some systems, it may make sense to use both the RFD and the ROM disk. The RFD can be used for application program and data storage that may require updating in the field, and the ROM disk can contain a backup set of software in case the Flash becomes destroyed during field reprogramming. With a hybrid system like this, it is possible to make a field-programmable device fail-safe.

Both the ROM disk and the RFD offer superb read performance, because reads consist almost entirely of data copies from ROM into RAM. Both disks are also extremely reliable, because they have no moving parts.

The ROM disk works on a copy of a file system that is obtained with a special program (`DISKIMAG.EXE`) from a floppy disk or hard disk partition that you fill with your own contents. Thus, if you can make a bootable diskette or hard drive, you can create a ROM disk version of that exact disk or partition.

Don't forget that although the diskette you're making a copy of probably supports 1.44MB of space, you may not have 1.44MB of ROMs that you're copying the image into. You need to review the special procedures outlined later in this chapter to ensure that you copy all of your data into the ROMs that the ROM disk software uses. The same logic applies to hard drive partitions; the ROM is probably much smaller than the hard drive partition you're using.

12.6.1 Enabling the ROM Disk Support Options

Before using the ROM disk feature, you'll need to have configured EMBEDDED BIOS properly. This is done by enabling the ROM disk's file system driver for each ROM disk you wish to configure in the system with the **FILE_SYSTEM** macro in the project file.

The **FILE_SYSTEM** macro specifies whether the ROM disk will emulate a floppy or hard disk, and the media address of the ROM image itself (see Chapter 13 for more about media addresses).

```
FILE_SYSTEM Soft, Rom, 80000000h, 200000h, "First ROM Disk"
FILE_SYSTEM Hard, Rom, 40000000h, 180000h, "Second ROM Disk"
```

In the above examples, the first declares a soft-formatted (Floppy) ROM disk named "First ROM Disk" that is mapped to the ROM image at media address 80000000h and is 200000h bytes long (2MB). The second example shows a hard-formatted (Partitioned) ROM disk named "Second ROM Disk" that is mapped to the ROM image at media address 40000000h and is 180000h bytes long (1.5MB).

Note: The names specified in quotes must have more than four characters in them, or the assembler will not interpret the strings correctly (instead, it will wrongly assume that you're specifying a 16-bit or 32-bit binary value). Use longer, more descriptive names, as shown above.

<u>The range of 32-bit addresses in the ROM array must correspond to a range of addresses as specified with the **MEDIA_REGION** macro in the project file</u>. The **MEDIA_REGION** macro generates table entries that tell the core BIOS which MTD is associated with each 32-bit address region. For details about how to set-up that table, consult the **MEDIA_REGION** macro description in Chapter 6. The name of the MTD associated with ROM addresses is **Rom**. If you don't have any **MEDIA_REGION** macro entries in your project file, then one will automatically be assigned to the entire address range (00000000h-ffffffffh) for you.

12.6.2 Enabling the ROM Disk in SETUP

From the user's standpoint, ROM disks are mapped to drive letters by entering the Basic Setup screen and cycling through the options on each drive letter. Hard-formatted (partitioned) ROM disk drives are not available for drives A: and B:, but are available for all other drives. Soft-formatted (floppy) ROM disk drives are available for drives A:, B:, C:, and D: only.

To hard-code the factory default for a drive (say, drive A:) to map to the ROM disk, set **CONFIG_CMOS_ASSIGN_A** (or B for drive B:) to the index corresponding to the file system defined in the file system table. In our above example, the "First ROM Disk" has an index of 1 (the zero index means no drive assignment), and the "Second ROM Disk" has an index of 2. Thus, we might set **CONFIG_CMOS_ASSIGN_A** to 1, and **CONFIG_CMOS_ASSIGN_C** to 2, so that the "First ROM Disk" was assigned to drive A: and the "Second ROM Disk was assigned to drive C:.

12.6.3 Building a ROM Disk Image

Enabling the ROM disk feature in EMBEDDED BIOS is easy, as the above procedures show. Creating the ROM disk image to be stored at the ROM location you select is even easier.

To build a ROM disk image that is stored in a file, you need a floppy that you have made bootable, and copied your files to. Then, run the DISKIMAG utility (found in the TOOLS directory) on the disk, specifying the number of kilobytes to copy. For hard disk partitions, use the drive partition letter (say, C:), and then use the /P switch to indicate to DISKIMAG that it should create a partition table as a part of the disk image itself. This creates the master boot record that DOS and other operating systems need to properly recognize the image as a hard drive.

Beware of the "optimization" that DOS uses when storing files on your disk (either soft or hard). If you create a file and then delete it, the space is reclaimed by the operating system, but DOS maintains a roving pointer that points to the next area beyond the one just allocated by the file you deleted as a "sure bet" for allocating more data. This technique eliminates DOS's long scan through the FAT when creating new files on a reasonably full disk, but it also causes files to be stored toward the end of the disk, even when adequate space exists at the beginning.

This optimization can cause problems when you are creating a disk that is to be copied by DISKIMAG. If you are not transferring the entire diskette or partition to ROM, but instead are transferring only a portion of it, you must make certain that the file system on the disk, including the files you intend to copy, are located within the sectors to be transferred. If not, then the file data won't be copied by DISKIMAG.

You can be sure you've packed files towards the beginning of the disk by starting with a freshly-formatted floppy or disk partition, and then issue COPY commands without any intervening DELETE commands. Or, if you do delete files, remove the diskette from the drive for at least two seconds (this doesn't apply to hard drives of course), and return it to the drive. DOS will detect that the media has changed, and restore its roving "optimization" pointer.

The following example shows how to build a bootable DOS floppy whose contents are then transferred to a file by DISKIMAGE, so that it can be burned into ROM or Flash:

```
C:\EBIOS43\UTIL> FORMAT  A: /S

C:\EBIOS43\UTIL> COPY  MYPROG.EXE  A:

C:\EBIOS43\UTIL> DISKIMAG  A:  THEFILE  128
```

[the 128KB binary result is in THEFILE]

The following example shows how to create an image of the operating system on drive D: (a hard disk partition) whose contents are then transferred to a file by DISKIMAGE, so that it can

be burned into ROM or Flash. The DEFRAG command packs the system files toward the front of the disk:

```
C:\EBIOS43\UTIL> DEFRAG  D:

C:\EBIOS43\UTIL> DISKIMAG  D:  THEFILE  1024  /P
```

[the 1MB binary result is in THEFILE, complete with a partition table]

12.6.4 Troubleshooting the ROM Disk

The best way to debug the operation of the ROM disk is to go through a checklist process. Here is a recommended procedure for bringing up a ROM disk assigned to drive A:

1. Verify that the **FILE_SYSTEM** macro definition is correct. See Chapter 13.

2. Verify that **OPTION_SUPPORT_DISKIO** is enabled (the default is enabled).

3. Verify that **CONFIG_CMOS_ASSIGN_x** is set to the index associated with the right file system as defined in your **FILE_SYSTEM** table. The 0 index means *no assignment*.

At this point, you're ready to boot the target and enter the debugger. We aren't going to try to boot the system, because it will only distract us from the methodical step-by-step procedure necessary to verify that each part of the ROM disk initialization is working properly. Even if you think the ROM disk is working because you get the A: prompt and can see files, that doesn't mean the BIOS ROM disk is actually working. Consider that Embedded DOS-ROM has a built-in ROM disk scan, which may be detecting your ROM disk and you may be seeing that drive instead of the BIOS ROM disk. We don't want two drives mapped to the same thing in different ways, because it will cause problems later.

4. Boot the target and enter the debugger.

5. Using the RFL (read Flash memory) command, display the area of the address space where the ROM image should be. The RFL command is a bit like the D[ump] command, but it displays data at *media addresses* instead of physical addresses. Let's suppose the **FILE_SYSTEM** macro defines the ROM disk starting media address to be 03ff0000h. The sample RFL command would look like this (note the colon between the two sets of four hex digits is a peculiarity of RFL, and does not mean we have a segment:offset notation):

```
EBDEBUG: RFL 03ff:0000

03ff0000  EB 52 90 4D 53 44 4F 53-33 2E 33 00 02 01 01 00   .R.MSDOS3.3.....
03ff0010  02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00   ...@............
03ff0020  00 00 00 00 00 00 00 00-21 00 24 00 00 00 00 FF   ........!.$.....
03ff0030  FF 00 00 80 00 72 7C 00-11 44 4F 53 20 20 20 20   .....r|..DOS
03ff0040  20 53 59 53 4E 6F 20 73-79 73 00 44 69 73 6B 20    SYSNo sys.Disk
03ff0050  62 61 64 00 B8 00 11 8E-D0 BC 00 7A 8E C0 2B C0   bad........z..+.
03ff0060  8E D8 BF 00 7C 8B F7 FC-B9 00 02 F3 A4 2E FF 2E   ....|...........
03ff0070  35 7C 0E 1F A1 18 7C 8B-16 1A 7C F6 E2 A3 2A 7C   5|....|...|...*|

EBDEBUG: RFL

03ff0080  2A E4 A0 10 7C F7 26 16-7C 03 06 0E 7C 8B 0E 11   *...|.&.|...|...
```

```
03ff0090  7C 83 C1 0F D1 E9 D1 E9-D1 E9 D1 E9 03 C8 89 0E   |...............
03ff00A0  28 7C BB 00 7A E8 04 01-BA 10 00 26 38 37 74 16   (|..z......&87t.
03ff00B0  BE 39 7C 8B FB B9 0B 00-FC F3 A6 74 36 83 C3 20   .9|........t6..
03ff00C0  4A 75 E8 40 EB DC BE 44-7C B4 0E AC 0A C0 74 08   Ju.@...D|.....t.
03ff00D0  B3 07 56 CD 10 5E EB F1-2B C0 CD 16 CD 19 C4 1E   ..V..^..+.......
03ff00E0  31 7C 26 8B 57 05 03 16-2D 7C 52 8A 16 2C 7C FF   1|&.W...-|R..,|.
03ff00F0  2E 31 7C 8B 36 24 7C 26-8B 47 1C A3 2D 7C 26 8B   .1|.6$|&.G..-|&.

EBDEBUG: RFL

03ff0100  47 1A A3 26 7C B8 20 11-8E C0 A1 0E 7C 8B 0E 16   G..&|. .....|...
03ff0110  7C 2B DB E8 96 00 40 03-1E 0B 7C E2 F6 B8 80 00   |+....@...|.....
03ff0120  8E C0 2B DB A1 26 7C 0B-C0 74 9B B9 F8 FF 0B F6   ..+..&|..t......
03ff0130  75 02 B5 0F 23 C1 3B C1-74 A4 A1 26 7C 83 E8 02   u...#.;.t..&|...
03ff0140  53 8A 1E 0D 7C 2A FF F7-E3 5B 03 06 28 7C 83 D2   S...|*...[..(|..
03ff0150  00 2A ED 8A 0E 0D 7C E8-52 00 40 83 D2 00 03 1E   .*....|.R.@.....
03ff0160  0B 7C E2 F3 06 B8 20 11-8E C0 8B 3E 26 7C 03 FF   .|.... ....>&|..
03ff0170  0B F6 74 0A 26 8B 15 89-16 26 7C 07 EB A6 03 3E   ..t.&....&|....>

EBDEBUG: RFL

03ff0180  26 7C D1 EF 26 8B 15 73-04 B1 04 D3 EA 81 E2 FF   &|..&..s........
03ff0190  0F 89 16 26 7C 07 EB 8C-51 50 2B C0 8A 16 2C 7C   ...&|...QP+...,|
03ff01A0  CD 13 58 59 E2 0A BE 4B-7C E9 1D FF 51 B9 05 00   ..XY...K|...Q...
03ff01B0  E8 04 00 72 E3 59 C3 50-53 51 52 56 57 1E 06 8B   ...r.Y.PSQRVW...
03ff01C0  16 1E 7C 03 06 1C 7C 83-D2 00 F7 36 2A 7C 8A E8   ..|...|....6*|..
03ff01D0  8A CC 80 E1 03 D0 C9 D0-C9 8B C2 8B 16 18 7C F6   ..............|.
03ff01E0  F2 8A F0 FE C4 0A CC 8A-16 2C 7C B8 01 02 CD 13   .........,|.....
03ff01F0  07 1F 5F 5E 5A 59 5B 58-C3 00 00 00 00 00 55 AA   .._^ZY[X......U.
```

6. From this display, you can recognize the key ROM disk image components in the floppy's boot record, because it is the first sector in the ROM disk image itself. First, at the end of this 512-byte display, note that the last two bytes in the sector are 55h and aah. These must be present in order for the ROM disk to recognize the image properly. If the signature is reversed, then you need to use a more standard tool for formatting the floppy disk you used with DISKIMAG. If the signature isn't there, then the ROM disk certainly won't find this image in the area you're looking. If it is there, skip to step 8.

7. If the whole display, in fact, seems to contain repeating values, such as ffh or 00h, then you've found the problem, but there are several possibilities:

   a) The chipset or high-integration CPU has not been properly programmed to cause the ROM's contents to be accessable at the address you've specified as the ROM disk address. Use the CSR and CSW debugger commands to verify that the chip select registers, PGP pins, or ROMCS0 pins are set-up properly, and then repeat the procedure.

   b) The address you've specified may be incorrect. Take a look at the syntax of the RFL command in Chapter 9. RFL takes a single 32-bit media address with a colon in the middle of it. It does not accept physical addresses, nor does it accept segment:offset notation. You can try to view the BIOS ROM image by using "RFL 000F:0000". If this displays the Embedded BIOS image, you're on the right track, but that doesn't mean that you have the right media technology driver selected for the region containing your ROM disk image. If you don't see it, try using "D F000:0000". If this shows the beginning of the Embedded BIOS image, then there is probably a problem with MMU mapping.

Check your **CONFIG_PAGED_MEM_SEG** parameter to see if it matches what your board or chipset MMU hardware can support.

c) If the BIOS image can be displayed, then try the location where this same ROM is mapped in the extended memory address space. On 386-EX and SC300/SC400-class designs, you can try this: "DB %03FF0000". If you see the BIOS image, then you know that protect mode memory accesses are working, and that the A20 gate is working. If this shows the vector table or some low memory area instead, then you most likely need to focus on A20 gating problems.

d) If the BIOS image can be found at %03FF0000 or the corresponding address for your target, then double-check the ROM disk address. If your ROM disk is actually mapped to the physical address space, use a DB command to display it at the physical address space. If it appears there, but not through the MMU, then there is an MMU programming problem which may be related to the window starting address specified by **CONFIG_PAGED_MEM_SEG**.

If you still believe you are using DB to dump the correct address, and the chipset registers are programmed properly to allow you to gain access to the ROM, and you've seen DB work on other extended memory addresses as in (c) above, then it is time to suspect that your ROM disk image is in fact not burned into the ROM, or that the ROM isn't in the socket, or is not wired to the CPU properly, etc. For this stage, use a logic probe (very inexpensive, if you don't already have one), and verify that the chip select pin on the ROM device actually strobes when you use the same DB command for the ROM's starting address, but that when you choose another address in the system where the ROM shouldn't respond, that the chip select does not strobe. If the strobe is not working properly, then you have a wiring problem or a chipset register configuration problem.

e) If it appears that the chip select is strobing properly, then you should start to suspect that the image you created on your floppy disk or hard disk partition did not make it into your file, or that your file didn't make it into the correct place in the ROM. Use your PROM programmer's display feature to verify that what is actually burned into the ROM at the location you expect to be mapped to the ROM disk address is actually the data. If you see the data here, but can't see it with the debugger command, then you should start to suspect that the image starts in the wrong place in the ROM. To determine if this is so, replicate your ROM disk image with a DOS COPY command (be sure to use /B to make BINARY copies), so that you have enough data to fill the entire ROM. For example, if you have a 256KB ROM disk, and a 1MB ROM part, you could use the following DOS COPY command to create a file called 1MB.BIN that can be burned into the ROM:

```
COPY  /B  256KB.ROM+256KB.ROM+256KB.ROM+256KB.ROM  1MB.BIN
```

Now try using the debugger with this ROM in place to see if this caused data to become available at the address where the ROM disk should be located. If it did, then you have bracketed the problem to programming the image in the wrong place in the part. At this point, it would be good to not trust your PROM programming procedure.

f) If this test still doesn't produce any image at the address you're expecting it to appear, then rethink again the possibility that this is a chipset configuration problem, but not something involving just basic addressing. Consider wait states, command timing, and things of that nature. Fortunately, the chipset registers can be programmed and inspected with the CSW and CSR debugger commands, so you can try experiments and then use DB to display the ROM's contents.

8. Assuming that you've been able to display what seems to be actual data at the ROM disk address, you should verify that the contents of the first 512 bytes (an example is shown earlier in this section) have the format that the ROM disk needs to operate correctly. The first byte should be an EBh. If it isn't, then this boot record was created using some FORMAT program that is unusual, as all standard MS-DOS compatible FORMAT programs create this in the first byte.

9. Check the media descriptor byte on the second line; it is the sixth hexadecimal byte displayed on the line after the address at the beginning. In the example above, it is F0h. If your media descriptor doesn't start with an F, then it isn't a valid media descriptor. It should be F0h if you used a 1.44MB floppy during the DISKIMAG process. If everything looks good here, and keeping in mind that your signature as checked in step 8 looks correct, then you're ready to try to read the sector using the RD debugger command:

```
EBDEBUG: RD 0 1 0 0 4000:0
Sector 1, head 0, track 0, read from unit 0 into 4000:0000.
EBDEBUG: DB 4000:0

4000:0000  EB 52 90 4D 53 44 4F 53-33 2E 33 00 02 01 01 00   .R.MSDOS3.3.....
4000:0010  02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00   ...@............
4000:0020  00 00 00 00 00 00 00 00-21 00 24 00 00 00 00 FF   ........!.$.....
4000:0030  FF 00 00 80 00 72 7C 00-11 44 4F 53 20 20 20 20   .....r|..DOS
4000:0040  20 53 59 53 4E 6F 20 73-79 73 00 44 69 73 6B 20    SYSNo sys.Disk
4000:0050  62 61 64 00 B8 00 11 8E-D0 BC 00 7A 8E C0 2B C0   bad........z..+.
4000:0060  8E D8 BF 00 7C 8B F7 FC-B9 00 02 F3 A4 2E FF 2E   ....|...........
4000:0070  35 7C 0E 1F A1 18 7C 8B-16 1A 7C F6 E2 A3 2A 7C   5|....|...|...*|
```

If this display looks reasonable; i.e., the same as the display we obtained with the RFL statement that was used to examine the contents of the ROM itself, then the INT 13h ROM disk is working. You now have a confirmation that INT 13h services are being provided correctly to DOS, so if there are ROM disk problems at the command-prompt level, then it is not the BIOS that is at issue, but the DOS configuration.

If this command doesn't display the same data, then make sure you specified the correct drive number in the RD command above (use 0 for A: or 1 for B:), and that it reported successful data transfer. If there are still problems, consult General Software.

12.6.5 Using Paged or Windowed ROM Disks

Most targets have hardware (either a chipset or external windowing logic) that, under program control, can map portions of ROM arrays into a narrow region of memory (16K, 32K, or 64K) below 1MB. The ROM disk software calls the MCL to handle access to the ROM itself, and because the MCL calls the Board Personality Module's (BPM) **BoardMapAddress** routine to determine whether windowing or protected mode access should be used, the OEM can modify **BoardMapAddress** as necessary to use any windowing scheme.

The default **BoardMapAddress** routine (if none is supplied by the OEM in the BPM) calls the Chipset Personality Module's (CSPM) **CsMapAddress** routine. When using mainstream processors with chipset-like qualities such as the AMD SC300, SC310, SC400, and SC410, the CSPM as supplied by General Software provides the control software necessary to program the memory management units in the silicon. The OEM can choose to use the underlying **CsMapAddress** routine, or if a different method is desired, new code can be placed in the BPM or a new CSPM.

Still further, it may be desirable for a target to use real-mode addressing for address ranges below 1MB, windowed addressing for certain addresses above 1MB, and protected mode addressing for selected regions above 1MB. This flexibility can easily be built-into the **BoardMapAddress** routine in the OEM's BPM.

12.7 Emulating Disks With RAM

The EMBEDDED BIOS RAM disk provides emulation of a floppy disk or hard disk partition by performing memory copies from a RAM image of a floppy disk or hard disk partition. The RAM disk provides read and write operations, and can be formatted either with a DOS FORMAT command, or can be formatted from the SETUP screen system (see Chapter 16 for details).

The RAM disk is an ideal solution for short-term storage that is needed when the target is powered-on but can lose its state afterwards. It is also an excellent driver for PCMCIA battery-backed SRAM cards that hold their data even when power is removed. The EMBEDDED BIOS formatting procedure is compatible with 1MB, 2MB, 4MB, 8MB, 16MB, and 32MB PC cards.

CAUTION: Do not attempt to specify RAM disk sizes other than the powers of two above. Doing so will not allow the RAM disk software to create the proper BPB for DOS.

As with the ROM disk, the RAM disk offers exceptional read performance, and its write performance is essentially equal to read performance. It is quite reliable (however subject to the effects of power failures in non-battery-backed targets).

12.7.1 Enabling the RAM Disk Support Options

Before using the RAM disk feature, you'll need to have configured EMBEDDED BIOS properly. This is done by enabling the RAM disk's file system driver for each RAM disk you wish to configure in the system with the **FILE_SYSTEM** macro in the project file.

The **FILE_SYSTEM** macro specifies whether the RAM disk will emulate a floppy or hard disk, and the media address of the RAM image itself (see Chapter 13 for more about media addresses).

```
FILE_SYSTEM Soft, Ram, 80000000h, 200000h, "First RAM Disk"
FILE_SYSTEM Hard, Ram, 40000000h, 180000h, "Second RAM Disk"
```

In the above examples, the first declares a soft-formatted (Floppy) RAM disk named "First RAM Disk" that is mapped to the RAM area at media address 80000000h and is 200000h bytes long (2MB). The second example shows a hard-formatted (Partitioned) RAM disk named "Second RAM Disk" that is mapped to the RAM area at media address 40000000h and is 180000h bytes long (1.5MB).

> *Note: The names specified in quotes must have more than four characters in them, or the assembler will not interpret the strings correctly (instead, it will wrongly assume that you're specifying a 16-bit or 32-bit binary value). Use longer, more descriptive names, as shown above.*

The range of 32-bit addresses in the RAM array must correspond to a range of addresses as specified with the **MEDIA_REGION** macro in the project file. The **MEDIA_REGION** macro generates table entries that tell the core BIOS which MTD is associated with each 32-bit address region. For details about how to set-up that table, consult the **MEDIA_REGION** macro description in Chapter 6. The name of the MTD associated with RAM addresses is **Ram**. If you

don't have any **MEDIA_REGION** macro entries in your project file, then one will automatically be assigned to the entire address range (00000000h-ffffffffh) for you.

Be certain that if you decide to use main system memory for the RAM disk, that you tell EMBEDDED BIOS and application software to stay clear of it. We suggest using the *top* of either the lower or extended memory spaces for RAM disks that use main system memory, and then setting false upper limits for either low or extended memory with the **CONFIG_MAX_LOW_MEMORY** and **CONFIG_MAX_EXT_MEMORY** parameters in your project file. For example, if you have decided to map the RAM disk into low memory at the 512KB boundary (segment address 8000h), then you'll want to set **CONFIG_MAX_LOW_MEMORY** to 512 instead of 640.

12.7.2 Enabling the RAM Disk in SETUP

From the user's standpoint, RAM disks are mapped to drive letters by entering the Basic Setup screen and cycling through the options on each drive letter. Hard-formatted (partitioned) RAM disk drives are not available for drives A: and B:, but are available for all other drives. Soft-formatted (floppy) RAM disk drives are available for drives A:, B:, C:, and D: only.

To hard-code the factory default for a drive (say, drive A:) to map to the RAM disk, set **CONFIG_CMOS_ASSIGN_A** (or B for drive B:) to the index corresponding to the file system defined in the file system table. In our above example, the "First RAM Disk" has an index of 1 (the zero index means no drive assignment), and the "Second RAM Disk" has an index of 2. Thus, we might set **CONFIG_CMOS_ASSIGN_A** to 1, and **CONFIG_CMOS_ASSIGN_C** to 2, so that the "First RAM Disk" was assigned to drive A: and the "Second RAM Disk was assigned to drive C:.

12.7.3 Initializing the RAM Disk

The RAM disk memory can already be loaded with a file system and user application files if it is a battery-backed SRAM card. In this case, no additional formatting is required for the RAM disk to begin functioning. The RAM disk is even bootable as drive A: or C: if desired.

If the SRAM card is not formatted, or if you will be using uninitialized memory, then you will need to format the RAM disk with either the DOS format utility, or the built-in formatting utility in the SETUP screen system. The built-in formatting utility in the SETUP screen is enabled in the BIOS build with the **OPTION_SETUP_RAMDISK** option. If available, selecting this option will cause the RAM disk to be automatically reformatted with a boot record, two FATs, and an empty root directory. This is actually the high-level file system format, not a low-level one.

Note that the BPBs supplied by the RAM disk may not correspond to the ones supplied by the FORMAT program for your brand of DOS, since these are within limits at the discretion of the FORMAT program's designer. General Software recommends reformatting the RAM disk at the higher layer with the FORMAT program supplied by the DOS vendor.

12.7.4 Troubleshooting the RAM Disk

Here is a recommended procedure for bringing up a RAM disk assigned to drive A:

1. Verify that the **FILE_SYSTEM** macro definition is correct. See Chapter 13.

2. Verify that **OPTION_SUPPORT_DISKIO** is enabled (the default is enabled).

3. Verify that **CONFIG_CMOS_ASSIGN_x** is set to the index associated with the right file system as defined in your **FILE_SYSTEM** table. The 0 index means *no assignment*.

4. Verify that you've got **OPTION_SETUP_RAMDISK** and **OPTION_SUPPORT_SETUP** enabled, so that you can format the RAM disk from the SETUP main menu.4. Now you're ready to boot the target and do a BIOS format of the RAM disk. Press <Del> during the memory count-up (use ^C with redirected consoles) and you'll enter the main SETUP menu. Select FORMAT RAM DISK, and it should complete immediately.

5. Now, without powering-off the system, from the main SETUP menu, enter the debugger to see that the formatting has taken effect.

6. Using the RFL (read Flash memory) command, display the area of the address space where the ROM image should be. The RFL command is a bit like the D[ump] command, but it displays data at *media addresses* instead of physical addresses. Let's suppose the **FILE_SYSTEM** macro defines the RAM disk starting media address to be 00100000h (normally, this is actually equivalent to physical address %00100000h, but not in all systems). The sample RFL command would look like this (note the colon between the two sets of four hex digits is a peculiarity of RFL, and does not mean we have a segment:offset notation):

```
EBDEBUG: RFL 0010:0000

00100000  EB 52 90 4D 53 44 4F 53-33 2E 33 00 02 01 01 00   .R.MSDOS3.3.....
00100010  02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00   ...@............
00100020  00 00 00 00 00 00 00 00-21 00 24 00 00 00 00 FF   ........!.$.....
00100030  FF 00 00 80 00 72 7C 00-11 44 4F 53 20 20 20 20   .....r|..DOS
00100040  20 53 59 53 4E 6F 20 73-79 73 00 44 69 73 6B 20    SYSNo sys.Disk
00100050  62 61 64 00 B8 00 11 8E-D0 BC 00 7A 8E C0 2B C0   bad........z..+.
00100060  8E D8 BF 00 7C 8B F7 FC-B9 00 02 F3 A4 2E FF 2E   ....|...........
00100070  35 7C 0E 1F A1 18 7C 8B-16 1A 7C F6 E2 A3 2A 7C   5|....|...|...*|

EBDEBUG: RFL

00100080  2A E4 A0 10 7C F7 26 16-7C 03 06 0E 7C 8B 0E 11   *...|.&.|...|...
00100090  7C 83 C1 0F D1 E9 D1 E9-D1 E9 D1 E9 03 C8 89 0E   |...............
001000A0  28 7C BB 00 7A E8 04 01-BA 10 00 26 38 37 74 16   (|..z......&87t.
001000B0  BE 39 7C 8B FB B9 0B 00-FC F3 A6 74 36 83 C3 20   .9|........t6..
001000C0  4A 75 E8 40 EB DC BE 44-7C B4 0E AC 0A C0 74 08   Ju.@...D|.....t.
001000D0  B3 07 56 CD 10 5E EB F1-2B C0 CD 16 CD 19 C4 1E   ..V..^..+.......
001000E0  31 7C 26 8B 57 05 03 16-2D 7C 52 8A 16 2C 7C FF   1|&.W...-|R..,|.
001000F0  2E 31 7C 8B 36 24 7C 26-8B 47 1C A3 2D 7C 26 8B   .1|.6$|&.G..-|&.

EBDEBUG: RFL

00100100  47 1A A3 26 7C B8 20 11-8E C0 A1 0E 7C 8B 0E 16   G..&|. .....|...
00100110  7C 2B DB E8 96 00 40 03-1E 0B 7C E2 F6 B8 80 00   |+....@...|.....
00100120  8E C0 2B DB A1 26 7C 0B-C0 74 9B B9 F8 FF 0B F6   ..+..&|..t......
00100130  75 02 B5 0F 23 C1 3B C1-74 A4 A1 26 7C 83 E8 02   u...#.;.t..&|...
00100140  53 8A 1E 0D 7C 2A FF F7-E3 5B 03 06 28 7C 83 D2   S...|*...[..(|..
00100150  00 2A ED 8A 0E 0D 7C E8-52 00 40 83 D2 00 03 1E   .*....|.R.@.....
00100160  0B 7C E2 F3 06 B8 20 11-8E C0 8B 3E 26 7C 03 FF   .|.... ....>&|..
00100170  0B F6 74 0A 26 8B 15 89-16 26 7C 07 EB A6 03 3E   ..t.&....&|....>
```

```
EBDEBUG: RFL

00100180  26 7C D1 EF 26 8B 15 73-04 B1 04 D3 EA 81 E2 FF   &|..&..s........
00100190  0F 89 16 26 7C 07 EB 8C-51 50 2B C0 8A 16 2C 7C   ...&|...QP+...,|
001001A0  CD 13 58 59 E2 0A BE 4B-7C E9 1D FF 51 B9 05 00   ..XY...K|...Q...
001001B0  E8 04 00 72 E3 59 C3 50-53 51 52 56 57 1E 06 8B   ...r.Y.PSQRVW...
001001C0  16 1E 7C 03 06 1C 7C 83-D2 00 F7 36 2A 7C 8A E8   ..|...|....6*|..
001001D0  8A CC 80 E1 03 D0 C9 D0-C9 8B C2 8B 16 18 7C F6   ..............|.
001001E0  F2 8A F0 FE C4 0A CC 8A-16 2C 7C B8 01 02 CD 13   .........,|.....
001001F0  07 1F 5F 5E 5A 59 5B 58-C3 00 00 00 00 00 55 AA   .._^ZY[X......U.
```

7. From this display, you can recognize the key RAM disk image components in the floppy-compatible boot record, because it is the first sector in the RAM disk image itself. First, at the end of this 512-byte display, note that the last two bytes in the sector are 55h and aah. These must be present in order for the RAM disk to recognize the image properly. If the signature isn’t there, then the RAM disk didn’t initialize the same area of RAM you’re displaying.

8. If the whole display, in fact, seems to contain repeating values, such as ffh or 00h, then you’ve found the problem, but there are several possibilities:

   a) The memory is just uninitialized and the displayed memory is not where the RAM disk begins in memory. Try using the WFL command to enter values at the specified address, and then dump it again:

   ```
   EBDEBUG:  WFL  0010:0000  1 2 3 4
   EBDEBUG:  RFL  0010:0000
   ```

   b) The chipset or high-integration CPU has not been properly programmed to cause the RAM to be accessable at the address you’ve specified as the RAM disk address. Use the CSR and CSW debugger commands to verify that the chip select registers and DRAM configuration registers are set-up properly, and then repeat the procedure.

   c) The address you’ve specified may be incorrect. Take a look at the syntax of the RFL command in Chapter 9. RFL takes a single 32-bit media address with a colon in the middle of it. It does not accept physical addresses, nor does it accept segment:offset notation. You can try to view the BIOS ROM image by using "RFL 000F:0000". If this displays the Embedded BIOS image, you're on the right track, but that doesn't mean that you have the right media technology driver selected for the region containing your RAM disk data. If you don't see it, try using "D F000:0000". If this shows the beginning of the Embedded BIOS image, then there is probably a problem with MMU mapping. Check your **CONFIG_PAGED_MEM_SEG** parameter to see if it matches what your board or chipset MMU hardware can support.

   d) If the BIOS image can be displayed, then try the location where this same ROM is mapped in the extended memory address space. On 386-EX and SC300/SC400-class designs, you can try this: "DB %03FF0000". If you see the BIOS image, then you know that protect mode memory accesses are working, and that the A20 gate is working. If this shows the vector table or some low memory area instead, then you most likely need to focus on A20 gating problems.

   e) If the BIOS image can be found at %03FF0000 or the corresponding address for your target, then double-check the ROM disk address. If your ROM disk is actually mapped to

the physical address space, use a DB command to display it at the physical address space. If it appears there, but not through the MMU, then there is an MMU programming problem which may be related to the window starting address specified by **CONFIG_PAGED_MEM_SEG**.

9. Assuming that you've been able to display what seems to be actual data at the RAM disk address, you should verify that the contents of the first 512 bytes (an example is shown earlier in this section) have the format that the RAM disk needs to operate correctly. The first byte should be an EBh. If it isn't, then the RAM disk initialization code won't recognize the RAM disk image on the next boot.

10. Check the media descriptor byte on the second line; it is the sixth hexadecimal byte displayed on the line after the address at the beginning. In the example above, it is F0h. If your media descriptor doesn't start with an F, then it isn't a valid media descriptor. If everything looks good here, and keeping in mind that your signature as checked in step 7 looks correct, then you're ready to try to read the sector using the RD debugger command:

```
EBDEBUG: RD 0 1 0 0 4000:0
Sector 1, head 0, track 0, read from unit 0 into 4000:0000.
EBDEBUG: DB 4000:0

4000:0000  EB 52 90 4D 53 44 4F 53-33 2E 33 00 02 01 01 00   .R.MSDOS3.3.....
4000:0010  02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00   ...@............
4000:0020  00 00 00 00 00 00 00 00-21 00 24 00 00 00 00 FF   ........!.$.....
4000:0030  FF 00 00 80 00 72 7C 00-11 44 4F 53 20 20 20 20   .....r|..DOS
4000:0040  20 53 59 53 4E 6F 20 73-79 73 00 44 69 73 6B 20    SYSNo sys.Disk
4000:0050  62 61 64 00 B8 00 11 8E-D0 BC 00 7A 8E C0 2B C0   bad........z..+.
4000:0060  8E D8 BF 00 7C 8B F7 FC-B9 00 02 F3 A4 2E FF 2E   ....|...........
4000:0070  35 7C 0E 1F A1 18 7C 8B-16 1A 7C F6 E2 A3 2A 7C   5|....|...|...*|
```

If this display looks reasonable; i.e., the same as the display we obtained with the RFL statement that was used to examine the contents of the RAM itself, then the INT 13h RAM disk is working. You now have a confirmation that INT 13h services are being provided correctly to DOS, so if there are RAM disk problems at the command-prompt level, then it is not the BIOS that is at issue, but the DOS configuration.

If this command doesn't display the same data, then make sure you specified the correct drive number in the RD command above (use 0 for A: or 1 for B:, or 80 for C: or 81 for D:), and that it reported successful data transfer. If there are still problems, consult General Software.

12.8 Emulating Disks With Flash

The EMBEDDED BIOS Resident Flash Disk (RFD) provides emulation of a floppy disk or hard disk partition by reading and writing an array of Flash memory as though it were a real drive containing 512-byte sectors. Unlike physical floppy disks, the RFD can present a floppy up to 32 megabytes in size and have it recognized by virtually all modern operating systems.

The RFD supports both NOR and NAND Flash technologies. Only one technology is supported within a given BIOS build. The actual Flash technology is specified with the **MEDIA_REGION** statement in the project file.

The special properties of NOR Flash memory make reading quick, writing fairly slow, and erasing slow. NOR Flash is directly-accessible in the memory address space of the CPU, although it may be windowed into a small region below the 1MB address space or accessed as a linear array in the physical address space. NOR Flash devices are subject to wear, which can lead to slower operation and even device failure. NOR Flash blocks are usually 32KB-256KB in size, making an erase operation quite expensive in terms of the erasure time itself, and the amount of data copying when using a copy/erase/rewrite sequence to free-up an obsolete-but-previously-written area of the Flash.

The special properties of NAND Flash make reading, writing, and erasing all moderately fast.

The RFD solves these problems by providing a logical-to-physical mapping that efficiently uses the Flash in large blocks, in a rotating fashion called wear-leveling. This translation eliminates the need for the application to be aware of the WORM/Erase/Wear properties of the underlying Flash media.

The RFD requires "sectored" Flash devices to work properly. It relies on at least two blocks of Flash (usually, blocks are somewhere between 16KB and 128KB in size) that can be erased and programmed independently of one another. This array of blocks is called a "Flash array". There is no real maximum limit to the number of blocks that can comprise a Flash array. When more memory is required than one Flash device provides, multiple Flash devices in the same family can be used to form one large contiguous Flash array.

Bulk-type Flash parts don't work with RFD, and RFD is not intended for PCMCIA PC Cards based on Flash technology.

Only certain Flash parts can be supported by the RFD; a Media Technology Driver (MTD) must be present in the EMBEDDED BIOS adaptation that supports the Flash parts. See Chapter 13 for details about these MTDs.

RFD is an ideal solution for long-term storage of system and application software and data that must be updatable in the field and/or in the manufacturing facility.

In some systems, it may make sense to use both the RFD and the ROM disk. The RFD can be used for application program and data storage that may require updating in the field, and the ROM disk can contain a backup set of software in case the Flash contents become destroyed during field reprogramming. With a hybrid system like this, it is possible to make a field-programmable device fail-safe.

Both the ROM disk and the RFD offer superb read performance, because reads consist almost entirely of data copies from ROM into RAM. Write performance of the RFD is comparable to a floppy disk, whereas the ROM disk does not offer the ability to write to the media. Both disks are also extremely reliable, because they have no moving parts, and the wear-leveling algorithm in the RFD reduces the wear associated with writing and erasing Flash parts.

Some MTDs may support advanced features, including background erase and page-mode writes. These features are handled transparently by the MTD, optimizing RFD performance when the better Flash parts are used.

12.8.1 Enabling the RFD Support Options

Before using the RFD disk feature, you'll need to have configured EMBEDDED BIOS properly. This is done by enabling the RFD disk's file system driver for each Flash disk you wish to configure in the system with the **FILE_SYSTEM** macro in the project file.

The **FILE_SYSTEM** macro specifies whether the Flash disk will emulate a floppy or hard disk, and the media address of the Flash array itself (see Chapter 13 for more about media addresses).

```
FILE_SYSTEM Soft, Flash, 80000000h, 200000h, "First Flash Disk"
FILE_SYSTEM Hard, Flash, 40000000h, 180000h, "Second Flash Disk"
```

In the above examples, the first declares a soft-formatted (Floppy) Flash disk named "First Flash Disk" that is mapped to the Flash array at media address 80000000h and is 200000h bytes long (2MB). The second example shows a hard-formatted (Partitioned) Flash disk named "Second Flash Disk" that is mapped to the Flash array at media address 40000000h and is 180000h bytes long (1.5MB).

> *Note: The names specified in quotes must have more than four characters in them, or the assembler will not interpret the strings correctly (instead, it will wrongly assume that you're specifying a 16-bit or 32-bit binary value). Use longer, more descriptive names, as shown above.*

The range of 32-bit addresses in the Flash array must correspond to a range of addresses as specified with the **MEDIA_REGION** macro in the project file. The **MEDIA_REGION** macro generates table entries that tell the core BIOS which MTD is associated with each 32-bit address region. For details about how to set-up that table, consult the **MEDIA_REGION** macro description in Chapter 6. The name of the MTD associated with Flash addresses is dependent on the technology. For example, **Amd8_1**, **Amd8_2**, **Amd16_1**, **Int8_1**, **Int8_2**, **Int16_1**, etc. The MTD must match the media type, or the file system will not operate properly. If you don't have any **MEDIA_REGION** macro entries in your project file, then one will automatically be assigned to the entire address range (00000000h-ffffffffh) for you. This default MTD coverage is provided by the **Ram** MTD, which is suitable for use with the RFD using RAM instead of Flash for purposes of testing.

The **CONFIG_RFDDISK_KBBLKSIZE** parameter must be set to the physical size of a Flash block in the type of parts you are using. For example, 28F016 Flash devices have 64KB blocks, so this parameter would be set to 64. If Flash parts are being interleaved (as even/odd byte pairs), then you must double the normal device block size (thus, an array of two-way interleaved Flash parts with 16KB block size becomes an array with 32KB blocks).

The **OPTION_SUPPORT_SETUP** and **OPTION_SETUP_RFDDISK** parameters should be enabled. This allows the OEM to enter the SETUP screen's main menu and perform a low-level format of the RFD before high-level formatting it. POST will do this automatically the first time to initialize the Flash, but it is good to have a way to initialize the Flash later if need be.

It is a good idea to enable the Flash debugging commands in the debugger. Do this by enabling the **OPTION_SUPPORT_SETUP**, **OPTION_SETUP_DEBUGGER**, and **OPTION_DEBUG_FLASH** options. This will be useful in the test of the Flash array, before the Flash disk is actually used.

Also for testing purposes, we'll want to enable **OPTION_QUERY_VERIFYRFD** and **OPTION_QUERY_FORMATRFD**. Later these will be removed, but in the beginning they will be useful, as these options instruct the core BIOS to ask the user before automatically formatting or verifying/fixing the RFD contents. During the Flash array checkout, it is a good idea to not allow these automatic procedures to happen during POST, but go straight into the debugger and test the Flash with the RFL, WFL, and EFL commands.

12.8.2 Protected Mode .vs. Windowing Access to Flash

The physical accesses to the Flash array are handled by the MTD for the corresponding Flash technology. MTDs are designed to handle accesses to the media in either protected mode (with true 32-bit physical addresses) or in real mode (with access to the Flash handled through a fixed memory window located below 1MB). The MTDs call the Media Control Layer (MCL) to determine how to handle the 32-bit addresses such as those specified in the **MEDIA_REGION** table and the **FILE_SYSTEM** macro's starting address parameter.

In order to determine this, the MCL calls the Board Personality Module's (BPM) **BoardMapAddress** routine, which by default calls the Chipset Personality Module's (CSPM) **CsMapAddress** routine. For targets employing CSPMs or BPMs that support the chipset's memory management units, the default action uses the windowing approach, so as to maximize performance and provide compatibility with protected mode software such as Windows. Examples of CSPMs that provide hardware windowing are those from General Software for the AMD SC300, SC310, SC400, and SC410.

Should the OEM wish to force protected mode operation in certain situations where the MMU of the chipset must be reserved for use by the application program, a **BoardMapAddress** routine can be provided in the BPM which indicates protected mode access (see Chapter 20 for the specification of this routine).

The OEM can also instruct the BIOS to use a windowed approach in situations where no chipset support is available from General Software by providing a **BoardMapAddress** routine which performs the hardware mapping and indicates that the real-mode access is to be used (again, see Chapter 20).

AMD SC3x0 and SC4x0 users may need to set the **CONFIG_PAGED_MEM_SEG** configuration parameter to specify the memory window used by the MMU of the CPU. Consult the AMD documentation for details about which segment addresses are used by each MMU.

For more details about Flash drivers and address mapping, consult Chapter 13.

12.8.3 Enabling the RFD in SETUP

From the user's standpoint, Flash disks are mapped to drive letters by entering the Basic Setup screen and cycling through the options on each drive letter. Hard-formatted (partitioned) Flash disk drives are not available for drives A: and B:, but are available for all other drives. Soft-formatted (floppy) Flash disk drives are available for drives A:, B:, C:, and D: only.

To hard-code the factory default for a drive (say, drive A:) to map to the Flash disk, set **CONFIG_CMOS_ASSIGN_A** (or B for drive B:) to the index corresponding to the file system defined in the file system table. In our above example, the "First Flash Disk" has an index of 1 (the zero index means no drive assignment), and the "Second Flash Disk" has an index of 2. Thus, we might set **CONFIG_CMOS_ASSIGN_A** to 1, and **CONFIG_CMOS_ASSIGN_C** to 2, so that the "First Flash Disk" was assigned to drive A: and the "Second Flash Disk was assigned to drive C:.

12.8.4 Testing the Flash Array

Before beginning to use the RFD, it is a good idea to check-out the operation of the Flash array and the MTD assigned to that array. This will ensure that the **MEDIA_REGION** table is correct and that the correct MTD has been chosen for the system.

1. Boot the target, pressing <Del> during the memory count-up so that POST will enter the SETUP system's main menu.

2. You may receive a query about whether to format or verify the RFD. Respond negatively to these questions. We want to bypass RFD operations right now that could hang the machine if the Flash driver isn't set-up properly.

3. From the main menu of the SETUP screen system, enter the debugger.

4. From the debugger's prompt, use the RFL command to display the start of the Flash array's contents in the address space. For our example, let's suppose that the **FILE_SYSTEM** starting address value is set to 00800000h.

```
EBDEBUG:  RFL  80:0
0080:0000  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0010  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0020  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0030  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0040  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0050  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0060  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0070  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
```

This looks like erased Flash memory, because when Flash is erased, its contents are set to 1's (the FFFFh pattern). Note that the Flash is displayed in 16-bit words, because the MTDs operate on words, not individual bytes, even if you have an 8-bit Flash part. If you see the erased pattern, skip step 5 and go to step 6.

5. If you see other values, then either the Flash contains valid data that just hasn't been erased, or it is not yet accessible in the system. If this is the case, try erasing the Flash with the EFL command as follows:

```
EBDEBUG:  EFL  80:0
Flash block erased.
```

If an error occurs at this stage, or if the EFL command hangs, then you don't have a correct MTD assigned to the region with the **MEDIA_REGION** macro, or the Flash is not being presented correctly in the address space, or Vpp or write-enable may not be signaled to the Flash device. Note: It may be necessary to power-off the target to reset the state of the Flash devices in the array if they get into an invalid state.

If the EFL command completes its erasure successfully, then use the RFL command again to see that the contents are all FFFFh values. If not, check the **MEDIA_REGION** macro, the device addressing, or Vpp/write-enable.

a) The most common problem is that the **MEDIA_REGION** macro is not set-up properly. Each entry in the **MEDIA_REGION** table brings together the starting and ending physical addresses of the Flash array with the MTD responsible for handling that kind of Flash memory. Because there are different MTDs for 8-bit and 16-bit devices, and those MTDs are further subdivided into those with 1-way and 2-way interleave support, it is important to get the right MTD.

b) Another possibility is that the Flash requires a special signal, such as a programmable package pin (PGP on Elan CPUs) or Vpp voltage to become active before writing can

take place. If your target has such special requirements, then these hardware signals can be manipulated in the BPM routines, **BoardEnableWrites** and **BoardDisableWrites**. It is up to the OEM to define these routines for hardware not specifically supported by General Software.

c) Some chipsets or high-integration CPUs require chipset registers to be programmed so as to enable Flash parts to respond to certain physical addresses. This initialization is typically done in the BPM's **BoardInit1** routine, because address maps are largely board-dependent. It is up to the OEM to define this routine for hardware not specifically supported by General Software.

6. Let's assume you have an erased Flash part with FFFFh in each word of the array. Now use WFL to write two words to the Flash part in succession:

```
EBDEBUG:  WFL  80:0  1234  2345
1234 written to 0080:0000.
2345 written to 0080:0002.
```

If this shows success, display the Flash memory again with RFL to see that the media was actually modified. If this WFL command hangs or reports failure, then any of the problems cited in step 5 above may be the cause. Note: It may be necessary to power-off the target to reset the state of the Flash devices in the array if they get into an invalid state.

```
EBDEBUG:  RFL  80:0
0080:0000  1234 2345 FFFF FFFF FFFF FFFF FFFF FFFF
0080:0010  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0020  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0030  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0040  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0050  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0060  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
0080:0070  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
```

If the media was actually modified as shown in the above display, then the MTD is working with your Flash array, and you're ready to verify the block size of the Flash array. If you don't see something similar to the above display, then there are problems. See some possible solutions in step 5.

7. With the MTD basically working, verify the block size as follows. Let's assume in our example that we have a 64KB block. That means that if the first block's starting address is 00800000h, then the second block's address is 00810000h. Just for clarity as an additional example, for 16KB blocks, the first block would be 00800000h and the second, 00804000h. However, we'll continue on here with the idea of 64KB blocks.

a) Use EFL to erase the first block and the second block:

```
EBDEBUG:  EFL  80:0
Flash block erased.
EBDEBUG:  EFL  81:0
Flash block erased.
```

b) Use WFL to write a different word at the beginning of each block:

```
EBDEBUG:  WFL 80:0  1234
1234 written to 0080:0000.
EBDEBUG:  WFL 81:0  2345
2345 written to 0081:0000.
```

c) Use RFL to verify that these values stick in the right places in the Flash array.

d) Use EFL to erase the first block. If you've got the block size right (64KB in our example), this won't erase the second block, so you should see FFFFh everywhere in the first block, but 2345h in the second block.

e) If the second block gets erased when you erase the first block, then the array has blocks of different sizes than you thought it did, or is somehow interleaved differently than you thought it was, or you are using the wrong MTD. Continue to make experiments such as this, until you determine the minimum eraseable block size on the target.

f) Once you've determined the real logical block size on the target, build a new BIOS with **CONFIG_RFDDISK_KBBLKSIZE** set to the size of a block in kilobytes. It is critical to the RFD's operation that this parameter be correct, or the RFD may appear to work for a while but will lose file data.

12.8.5 Initializing the RFD (Low-Level Formatting)

Once the Flash array has been tested, you'll need to low-level format the RFD so that it can begin emulating a floppy disk. Keep in mind that this type of formatting is not the same as the formatting that the DOS FORMAT command provides; here, this low-level formatting erases the Flash blocks and establishes the logical-to-physical sector mapping system in the media.

If you have completed the Flash testing in the above section, you will be used to responding negatively to the queries to verify or format the RFD. In this section and from now on however, you'll want to respond positively to them, so that the RFD will initialize properly.

The low-level formatting process can be done in any of the following ways:

During POST. POST will automatically prepare the RFD by initializing any blocks that do not contain a valid RFD header. Additionally, each block is scanned to ensure that sector slots marked "erased" truly have no data stored in them.

From the SETUP system (see Chapter 16), you can select the FORMAT FLASH DISK option. This will automatically begin sectoring the Flash devices.

From the Manufacturing Mode (see Chapter 18), you can program the Flash devices remotely over a high-speed RS232 serial link. By simply issuing requests to erase the blocks used by the Flash array, you will prepare them for the RFD's use.

From an application program, you can use the EMBEDDED BIOS Flash programming extensions to the INT 15h System BIOS functions. This program can then run on the target. After it formats the RFD, it needs to reboot the target (soft reset is fine) in order for the RFD software to notice that the RFD Flash array has been updated by foreign software.

We suggest you use the special SETUP screen already designed to format your Flash array. Once formatted, the Flash array is still not capable of being used to store files. While the low-level formatting of the device has been performed, the high-level DOS FORMAT command

needs to be used to actually store the boot record, FATs, and root directory in the blank sectors so that DOS or another operating system can recognize it as a valid drive.

Formatting should take about 0.5 seconds to 2 seconds per block, depending on the type of devices you are using in the Flash array, the Vpp voltage being used, and numerous other factors.

After the RFD has been low-level formatted, it is ready to be high-level formatted if it is a floppy emulator, and FDISKed and high-level formatted if it is a hard drive emulator. This can be accomplished in either of two ways: Manufacturing Mode, or the DOS FORMAT command.

12.8.6 Using DOS to FDISK a Hard-Formatted RFD

When the RFD is used to emulate a hard disk (**FILE_SYSTEM** type Hard), then you must create a partition table on the disk after it is low-level formatted, but before it is high-level formatted. This is accomplished in exactly the same way that FDISKing a real hard drive is done.

From DOS, run FDISK and select the right hard drive to partition. Make sure you don't accidently partition another hard drive unintentionally; there is an option in FDISK to switch hard drives for targets with multiple hard drives. Once you partition the drive and create a DOS partition, you'll need to make it the active partition, and then save the partition table. DOS will want you to reboot the machine so that when it does, the new drive partition can be recognized as ready to format at the high level.

Caution: When using DOS to FDISK any drive, including the RFD or even an IDE drive, make certain that it is FDISKed while it is the first drive in the system at the time FDISK is run, if you wish to boot an operating system from the drive. Otherwise, the FDISK utility will install the wrong BIOS unit number in the Master Boot Record on the drive, and the drive will not be bootable. This is a limitation/defect of some DOS operating systems, not the BIOS.

12.8.7 Using DOS to FORMAT the RFD

If another drive is available on your target to hold the FORMAT program, then you can use the DOS FORMAT program to create a file system on the RFD. The drive letter to format will be either A: or B:, or another drive letter for a hard drive, depending on which drive was mapped to the RFD with **CONFIG_CMOS_ASSIGN_A** and **CONFIG_CMOS_ASSIGN_B** or their counterparts in the BASIC SETUP screen, Drive A: and Drive B: device assignments.

Let's assume that you've assigned drive A: to a Flash drive in the SETUP system. Then, from another drive (say, a hard disk), type:

```
C> FORMAT  A:
Press any key to begin formatting...
Formatting 100% complete.
C>
```

Once the FORMAT has completed, you can start copying files to the RFD just as though it were a big floppy disk. To make the RFD bootable, be sure to specify the /S option on the FORMAT command, or use the SYS command after formatting the RFD.

Do not attempt to specify tracks, sectors per track, or single sided options on the DOS FORMAT command. Either use no parameters (other than the drive letter), or supply the "/C" option, if you are using Embedded DOS-ROM's FORMAT command. The /C option allows you to create a compact file system that has only one FAT and a small, 64-entry root directory.

12.8.8 Using Manufacturing Mode to Format the RFD

If another drive is not available on your target to hold the FORMAT program, and you have an RS-232 serial connection to the target, you can use Manufacturing Mode to high-level format the RFD and also copy files to it, all from a host PC or laptop. To do this, you'll need to enable Manufacturing Mode in your BIOS build (see Chapter 14), and then follow this procedure:

1. To start, boot the target and enter the SETUP main menu. If you have a remote console over an RS-232 serial connection, press ^C during the memory count-up to enter SETUP. From the SETUP screen's main menu, select ENTER MANUFACTURING MODE. This will cause a couple garbled characters to be displayed on the terminal screen, and the target will stop displaying text.

2. On the host, exit the terminal emulation software and reboot, installing `DEVICE=MFGDRV.SYS /PORT=n /BAUD=m /UNIT=0` in its CONFIG.SYS file. This device driver creates a drive letter on the host the next time it is booted, that maps to the target's RFD.

3. On the host, switch to the first drive letter beyond the last normal drive in the system. For example, if you have drives A:, B:, and C: on your host machine, then D: will be the drive assigned to the Manufacturing Mode drive associated with the RFD on the target.

4. Use the FORMAT program that comes with your DOS on your HOST, not the target, and ask to format the remote drive:

```
C>  FORMAT D:
FORMAT will destroy all data on nonremovable drive D:
Proceed with Format? (Y/N): Y
Formatting 100% complete.
C>
```

   The formatting process may take a few seconds, or more depending on how big the RFD is. If you encounter a General Failure or Sector not Found error, then the Flash is not responding to the protocol fast enough. In that case, select a slower baud rate on both the target and host. The default baud rate is 19K, but it can go as low as 9600, and up to 115K.

5. Now the remote drive can be treated as any DOS drive. You may copy files to the drive using utilities and DOS commands, edit files on the remote drive, and use DIR, COPY, RENAME, and other techniques you would ordinarily use on the host to manipulate these files.

Chapter 13

DRIVERS FOR FLASH AND OTHER MEDIA

EMBEDDED BIOS provides support for Flash and other storage media through the Media Control Layer (MCL) and Media Technology Drivers (MTDs). This chapter presents the overall architecture of this software, and documents how it interacts with the rest of the system through architected programming interfaces.

13.1 Media Control Layer

The Media Control Layer (MCL) provides abstracted I/O services for client software in EMBEDDED BIOS, including Flash disk software, Flash programming interfaces, the ROM disk, the debugger's Flash programming commands, and Manufacturing Mode.

All MCL clients request MCL services through the MCL Application Programming Interface (MCL API). MCL uses this interface to hide details about the underlying media so that clients can remain small.

At the same time, the logical division of work marked by the API boundary makes it possible to implement support for new media types without requiring extensive knowledge of how all the client file systems operate; instead, when a new driver is implemented, all MCL clients can automatically access the new media.

MCL also abstracts addressing of media, and can support media I/O in both real and protected modes of the CPU. Further, MCL allows the Board Personality Module (BPM) to enter into the address translation loop, so that the OEM can establish an addressing architecture that is used system-wide and conforms with chipset programming.

Efficient and flexible Vpp control is essential for high-performance embedded applications. Flash components may require a programming voltage to be applied during writing, locking, or erasing procedures. MCL provides scheduling of Vpp controls so that Vpp is only enabled when necessary. Further, MCL routes Vpp requests through the BPM so that the OEM can define any proprietary scheme for controlling Vpp without affecting core BIOS files.

Interrupt latency is a problem wherever interrupts are disabled during extensive movement of data. The MTDs may receive requests to transfer extensive amounts of data, while in 0:32 protected mode without an IDT available. This necessitates disabling interrupts at the time the transfer is made. MCL provides MTDHLP functions to reduce interrupt latency by allowing the

MTD to periodically switch back to real mode to allow interrupts to be serviced by application software. All MTDs using these features limit exposure to high interrupt latency when performing Flash I/O.

13.1.1 MCL Architecture

MCL is responsible for receiving I/O requests from the file systems implemented in the BIOS (RFD), the debugger Flash programming commands, and Manufacturing Mode. MCL interprets these requests and by inspection of the address mapping table's entries created with the **MEDIA_REGION** macro, routes them to the appropriate Media Technology Driver (MTD).

In order to perform its work, an MTD may require some specialized MTD Helper services (MTDHLP API) provided by MCL. These services provide a unified way to access memory mapping hardware, switch between real and protected modes of the CPU, enable or disable Vpp, provide micro-delays, and perform optimized data copies, among other things. The MTDHLP API is documented later in this chapter and is available only to writers of MTDs.

As part of MCL's processing of an MTDHLP API function, it may be necessary for MCL to request services of the Board Personality Module (BPM). These services include memory mapping requests (which commonly are routed by the BPM to the Chipset Personality Module or CSPM), and enable or disable Vpp to Flash arrays.

MCL provides a high-level API (the MCL API) for its clients, to hide the complexity of the MTD operations from its clients (the RFD, ROM disk, or debugger, for example). This reduces the complexity of the client software and makes it possible to run the client software on many different media types, and new media types as MTDs become available. The MCL API provides clients with the ability to read and write data to media, and then lock and erase logical blocks of media.

13.1.1.1 Media Types

MCL hides the details of how media are programmed for its clients, providing only a few basic operations that can be performed at the highest level. The small number of request types, and the simplicity of the request types, mean that it is possible to implement support for virtually any kind of storage media with an MTD, and have it plug into an adaptation of EMBEDDED BIOS so as to provide useful work.

General Software provides MTDs in the core BIOS for NOR Flash parts available from Intel and AMD, since they represent the majority of the types used in embedded x86 designs. Bulk erase, boot block, and sectored parts are all supported.

General Software also provides MTDs in the core BIOS for NAND parts available from Toshiba and AMD. The same Toshiba NAND driver also supports AMD UltraNand. These MTDs do not window or physically-map their media; rather, they use an I/O port pair to manage the Flash array. The standard MCL API abstracts this addressing issue, allowing the RFD and other software to work without modification on these types of media.

Additional Flash types are easily supported with additional MTDs. Contact General Software for information about other supported devices.

Other memory types, such as ROM and RAM, are also supported by MTDs in the core BIOS. For example, the ROM disk uses the ROM MTD, and the RAM MTD can be used to try-out the RFD on a target where Flash support is not yet available.

The MTD architecture can extend far beyond simply random-access memories. Consider that storage need not take place on the target itself. Using a data connection to a host PC, it might make sense for an MTD to simulate ROM, RAM, or Flash media with an RS232 asynchronous serial communication line, or with an Ethernet packet driver. Once an MTD is written for a given data communications layer, then the MTD can provide virtual I/O capabilities.

MTDs can also be written that logically transform one media type into another. For example, an MTD might be written that presents boot block devices as sectored Flash devices, by combining the smaller parameter blocks into another equal-sized larger block. During block erasure operations, the MTD could hide the details of erasing the brother parameter blocks for the special block.

MTDs can also be employed to change the performance of accesses to the media, by local caching of commonly-used data in RAM. Consider that an MTD might be implemented that responds to a certain logical address space known only to certain clients (say, the RFD). The caching MTD would cache requests and reorder them as necessary to gain performance. When this MTD needed to actually perform media accesses, it would issue MCL API function requests by proxy to the MTD handling the actual media, in a different range of physical addresses.

Additional intermediate functions for MTDs include security wrappers, disk mirrors, and diagnostic tools.

13.1.1.2 Media Addressing

All media addresses in the MCL system are 32 bits wide, and specify byte locations in the logical address space. In many systems, the logical address space corresponds to the 4.2 gigabyte physical memory address space addressible by the CPU itself. In other systems, media addresses may not correspond to directly-addressible memory locations. For this reason, we must distinguish logical (media) addresses which are specified by MCL clients, from physical (CPU) addresses which are available to the CPU in protected mode.

The simplest method for addressing media such as RAM or ROM in a system is to directly map it into the 32-bit physical memory address space of the CPU. The ROM and RAM MTDs handle these cases by switching to protected mode and accessing the memory locations with a 4K granular selector mapped to physical address 00000000h, and using 32-bit offsets with respect to this selector that correspond to logical addresses. In this case, physical addresses are associated with logical addresses because the media responds to those physical addresses on the bus.

Another method for addressing RAM and ROM in a system is to construct a window, or small region of memory address space below the 1MB address marker, and then page selected portions of the larger RAM or ROM array into the window. The windowed approach is also supported by the ROM and RAM MTDs, because some targets cannot directly map entire RAM or ROM arrays into extended memory, or may not support protected mode. In the windowed case, logical addresses are mapped to physical addresses within the window through hardware assistance. Because the hardware is usually configurable, the logical address range for the ROM or RAM is selected based on convenience; for example, it is just as easy to think of logical addresses 80000000h-8fffffffh to map the device as it is for 90000000h-9fffffffh to be used. In other words, the logical address assignments are somewhat arbitrary.

Some storage devices are not direct-addressible; that is, they require more work to read or write a byte of storage than to simply strobe an address to the address pins, and together with manipulation of control signals, read or write data to the data I/O pins. Instead, they may require that a protocol be used to feed addresses and data through I/O ports mapped to the device. In this case, the address space in the device must be mapped to a range of logical addresses as with the windowed example. Instead of performing direct-memory I/O however, the MTD must translate these logical addresses into corresponding device-specific parameters which can be passed-through to the device itself. As with the windowed approach, the assignment of the specific logical address space for such devices is somewhat arbitrary.

Some embedded CPUs and chipsets such as the AMD SC300 or AMD SC400 require initialization in order to map chip selects to a range of physical addresses. This initialization, normally done in routines **CsInit1** or **BoardInit1**, must be coordinated with the logical addresses specified by client software. For example, if the RFD is directed to use a Flash array at logical address 00800000h on an AMD SC300-based target, then the DOSCS chip selects must be programmed to respond to the addresses starting at physical address 00800000h and ending at the end of the Flash array.

These initialization values must also be coordinated with the **CsMapAddress** and **BoardMapAddress** routines, when using more advanced processors such as the AMD SC400, since the SC400 does not necessarily map the Flash array into the physical address space, but instead makes it available only through a programmable window. In such cases, an architected 32-bit logical address space is a necessity, and General Software has provided a straightforward segmentation of the 32-bit logical address space to accommodate ROMCS0 and ROMCS1 devices.

MCL uses a logical address table built with the **MEDIA_REGION** macros, to describe the logical address space in the system. Logical address ranges are associated with MTDs responsible for handling the devices in those ranges. Once they receive control, MTDs request that MCL translate logical addresses to windowed or physical addresses. MCL performs this work by calling **BoardMapAddress**, which by default calls **CsMapAddress**. The default implementation of **CsMapAddress** performs an identity mapping of the 32-bit logical address to the same address in the physical address space. This simple mapping allows most designs to use this architecture to access devices mapped into extended memory without the aid of special board or chipset module routines.

By implementing the **CsMapAddress** function in the CSPM, the MCL's request to translate a logical address is handled at the chipset level, typically mapping the specified memory into a memory window. This mapping usually takes place by programming the chipset registers in this routine with values derived from the logical address. In some circumstances, it is necessary for the chipset to respond to a certain range of logical addresses (as noted earlier with the SC400 example) and associate them with one chip select, and another range of logical addresses with another chip select. In this case, these conventions are implemented in the **CsMapAddress** routine.

The OEM can override the conventions and policy decisions of the underlying **CsMapAddress** function by supplying a **BoardMapAddress** function which either performs proprietary mapping without calling **CsMapAddress** at all, or translates the logical address passed to it before passing on the modified logical address to **CsMapAddress**.

In order to allow restoration of the registers associated with the memory addressing function of **BoardMapAddress** and **CsMapAddress**, two additional functions, **BoardUnMapAddress** and **CsUnMapAddress**, are called by MCL on exit from any media function. Normally these routines do nothing, but they can be supplied by the OEM to restore the state of the mapping

hardware to what it was before the original mapping. Note that an MTD function may call the MapAddress functions zero, one, or many times in order to handle a request, but the UnMapAddress function is always called exactly once for each request. Therefore, if a save/restore state mechanism is to be implemented in the OEM's adaptation, a board-level or chipset-level flag must be implemented that indicates that the associated UnMapAddress function must or must not perform work.

13.1.1.3 Vpp Control

MCL implements Vpp control by providing two routines, **MtdHlpEnableVpp** and **MtdHlpDisableVpp**, for MTDs to call to indicate that Vpp needs to be turned on before writes occur, and off after they are completed. This abstraction of the mechanics of enabling and disabling Vpp in the system offers several important advantages.

The most obvious advantage is that the proprietary details of how Vpp is controlled are hidden from all MTDs, so that MTDs do not need to be modified when being used in a system with a new Vpp control mechanism. It also means that newly-implemented MTDs can be used in systems that already have Vpp controls defined, without considering the Vpp control implementation in the new MTDs.

Another advantage to the separation of the mechanics of Vpp control from the control requests is that MCL can act as a central clearinghouse for Vpp requests, and possibly optimize them in order to improve overall system performance. MCL can do this by realizing that the actual enabling of Vpp is usually expensive, since a short (but significant, say 100us) delay is required after enabling Vpp before it has stabilized. This delay, if incurred for each write operation, would have serious performance impacts on the system.

To address this performance challenge, MCL attempts to defer disabling Vpp once it is enabled by a call to **BoardEnableWrites**, so that subsequent writes that issue Vpp enable requests do not incur the stabilization delay. After a configurable period of real time in which no Vpp enable requests are received, MCL automatically initiates a call to the BPM's **BoardDisableWrites** routine to physically turn-off Vpp.

As has been alluded to above, MCL calls the BPM routines to enable and disable Vpp for the system. The BPM's **BoardEnableWrites** routine performs the actual enabling of Vpp in the circuit, and also performs whatever delay is necessary so that when it returns to MCL, the Vpp has stabilized. The BPM's **BoardDisableWrites** routine is called by MCL only when Vpp is truly no longer used; therefore, it simply disables Vpp, and lets MCL handle the deferred Vpp disable.

13.1.1.4 Interrupt Latency

MCL provides MTDs with simple functions to switch to protected mode from real mode, and to switch back again into real mode. These tools can be used by MTDs to break-up large data transfers in protected mode into a series of smaller transfers that have lower interrupt latency.

When an MTD operates in protected mode, it calls the **MtdHlpToProt** function, which disables interrupts necessarily. It cannot establish an IDT because it may not be prepared to handle user application interrupts in protected mode. Because disabling interrupts can adversely affect overall system performance, the amount of time that interrupts are disabled must be minimized.

The actual amount of time that interrupts may be left disabled is dependent on the application. The amount of work that can be done in that amount of time is, of course, a function of how fast the target is. The application's interrupt latency requirement is usually a function of how it needs to interface with the outside world. Consider that an application using interrupt-driven asynchronous serial I/O may need to perform transfers at baud rates of 19K baud or higher; at this rate, an interrupt arrives approximately every 500us for each character. This would mean that we would need to transfer any-sized chunk of data in an MTD in under 500us.

Of course, without segmenting the I/O, it would be possible to exceed this time limit by the sheer performance limits of the memory system and CPU. For example, a request to move 64KB in this amount of time would need to transfer data faster than (65536 kb / .000500 sec) = 128MB/sec, not counting the time it takes to switch to protected mode and switch back again. Clearly, 128MB/sec is a high data rate that is unsustainable on many targets.

Direct-access storage MTDs, such as Flash, ROM, and RAM MTDs, can be coded switch back into real mode at regular work intervals when transferring large data blocks, so that the maximum interrupt latency hit is no more than that necessary to transfer 512 bytes. This reduces the above example's data rate requirements from 128MB/sec to 1MB/sec, which is sustainable for most embedded targets.

13.1.2 MCL Entrypoints

MCL receives control in two ways. The first request type is submitted from within the core BIOS power manager to indicate that a change in power management state is taking place. This is handled with the **MediaPwrLevel** entrypoint.

The second request type is submitted by an MTD client, which may be the debugger, Manufacturing Mode, the ROM disk, the RFD, or possibly other file systems or specialized subsystems. These clients all request the basic services of MCL in order to interact with certain media so as to provide higher-level functionality to their clients. The second request type is formed by all the other requests. There are several entrypoints, including **MediaLockBlock**, **MediaReadBlock**, **MediaWriteBlock**, **MediaStartErase**, and **MediaEraseComplete**.

These entrypoints are not directly callable from application programs; they are always called from within the core BIOS. Applications can cause these entrypoints to be invoked by calling APM functions to control the system's power level, or by calling the Flash programming API of the INT 15h software interrupt (see Chapter 21 for details).

MCL is always called from a real-mode context with interrupts enabled. MCL may switch modes or cause interrupts-disabled windows to occur, since it must pass control to the underlying MTDs which may switch modes as necessary to perform their functions.

Registers are generally preserved unless they are used to return values in specific cases. The carry flag (CY) is used to indicate either a successful or failing outcome from a function call. In one case (**MediaEraseComplete**), the CY flag is used to indicate the status of an ongoing operation.

Stack depth is kept to a minimum in MCL, in anticipation of passing on this stack availability to the underlying MTD. MTDs should keep stack depth to a minimum as well. A suggested maximum amount of stack usage by the MTD is 64 bytes.

13.1.2.1 MediaPwrLevel Entrypoint

The **MediaPwrLevel** function is called with procedure linkage by the EMBEDDED BIOS Power Management System to coordinate the MCL's power with the rest of the system's state.

MCL's power management function's purpose is largely a placeholder. It needs to be present so that it can be specified as a node in the power management tree, with its subordinate **MtdPwrLevel** power management functions specified as children. This allows MTDs to be notified when the system's power is about to change states, so that they can perform cleanup when power goes down, and resume activities when power returns.

MTDs need not support their power management function, unless they are to be included in the power management device tree.

Input Parameters:

DS - Points to the extended BIOS data area (EBDA).
BX - Device index.
CL - New power level.
CH - Old power level.

Output Parameters:

None.

Unpreserved Registers:

Flags.

13.1.2.2 MediaLockBlock Procedure

The **MediaLockBlock** function is called with procedure linkage to lock a block of storage in a sectored or boot block Flash device. Once a block has been locked, it remains write-protected until unlocked by a subsequent erase operation.

Some MTDs may or may not support the lock function, and some may not even support the erase functions. For example, the **Rom** MTD does not support this function because it cannot change the underlying media. The **Ram** MTD supports erase by emulating it, assuming the same block size as specified for the RFD, but has no way to "lock" a block of RAM.

Input Parameters:

DX:AX - 32-bit address of 1st byte within block to be locked.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

13.1.2.3 MediaStartErase Procedure

The **MediaStartErase** function is called with procedure linkage to start a block erase operation on a block of storage in a sectored, boot block, or bulk erase Flash device. Once the block is erased, its contents are reset by the device such that each byte contains the hexadecimal value, ffh (all ones).

This function's purpose is to begin erase processing, and optimally return before the erasure has completed, so that the erase processing can occur in the background while the system continues with other operations. This can be effective in increasing RFD performance. Some MTDs may not implement an asynchronous erasure operation, and instead implement the entire erase functionality in the **MediaStartErase**, so that the **MediaEraseComplete** routine always returns to the caller with a "completed" status. For devices without background erase capabilities, this routine should complete synchronously as described.

Certain MTDs provided with the core EMBEDDED BIOS software illustrate how to break-up the processing of starting the erase process and determining if the erase process has completed. One example of such an MTD is `MTDINTA.ASM`. Note that this MTD (necessarily) handles situations where, once an erase process has started and control has returned to the client, another request is received to perform a read or write before the erase operation has completed. In these cases, the erase operation must either be suspended during the secondary operation's performance, or alternatively, the erase operation may be concluded before processing the secondary operation. All MTD operations must be coordinated to handle such background processing gracefully without loss of data.

Some MTDs may or may not support the erase functions (either **MediaStartErase** or **MediaEraseComplete**). For example, the **Rom** MTD does not support this function because it cannot change the underlying media. The **Ram** MTD does support the function by emulating it, assuming the same block size as specified for the RFD.

Input Parameters:

DX:AX - 32-bit address of 1st byte within block to be erased.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

13.1.2.4 MediaEraseComplete Procedure

The **MediaEraseComplete** function is called with procedure linkage to determine if the erasure operation started by a previous call to the **MediaStartErase** function has completed or is still in progress.

Some MTDs may not implement an asynchronous erasure operation, and instead implement the entire erase functionality in the **MediaStartErase**, so that the **MediaEraseComplete** routine always returns to the caller with a "completed" status.

Some MTDs may or may not support the erase functions (either **MediaStartErase** or **MediaEraseComplete**). For example, the **Rom** MTD does not support this function because it cannot change the underlying media. The **Ram** MTD does support the function by emulating it, assuming the same block size as specified for the RFD.

Input Parameters:

DX:AX - 32-bit address of the block as specified in the call to **MediaStartErase**.

Output Parameters:

CY - clear if no erase operation in progress, else set.

Unpreserved Registers:

Flags.

13.1.2.5 MediaReadBlock Procedure

The **MediaReadBlock** function is called with procedure linkage to read data from the media.

Commonly, MTDs call the **MtdHlpRead** function to perform the transfer, when the media is accessible as direct-access storage within the memory address space. Making use of this function in new MTDs saves space by reusing existing code, and also takes advantage of its method of breaking the read into pieces when in protected mode to reduce interrupt latency.

Sometimes, MTDs cannot use this helper function, because device programming is required. For example, some Flash devices may be controlled through a range of I/O ports, and may not actually present any directly-addressible memory regions to the CPU.

Input Parameters:

DX:AX - 32-bit address of 1st byte of media to be read.

ES:BX - 16:16 segment offset pointer to 1st byte of user buffer where data will be written in RAM.

CX - Number of bytes to be transferred. A zero value indicates 65,536 bytes. The value does not have to be even.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

13.1.2.6 MediaWriteBlock Procedure

The **MediaWriteBlock** function is called with procedure linkage to write data from a user-specified buffer to the media.

The caller must take care that the writing is possible; i.e., that the area of the media to be written does not contain other data that has not been erased, unless the underlying media supports writing without erasing. NOR Flash devices require erasure of large blocks, whereas NAND Flash devices can be supported by MTDs that build-in an automatic pre-erase operation. Similarly, the RAM MTD can write to an area that has not previously been erased.

Some MTDs may or may not support the write function (or erase functions). For example, the **Rom** MTD does not support this function because it cannot change the underlying media.

Input Parameters:

DX:AX - 32-bit address of 1st byte of media to be written.

ES:BX - 16:16 segment offset pointer to 1st byte of user buffer containing data to be written.

CX - Number of bytes to be transferred. A zero value indicates 65,536 bytes. The value must be even.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

13.1.2.7 MediaQuery Procedure

The **MediaQuery** function is called with procedure linkage to request information about the MTD itself, or to probe the actual media to verify that it is operational, or to determine the block size or data path width. Additional information classes may be made available in future releases.

Some MTDs may or may not support the Query function; its implementation is optional.

Input Parameters:

DX:AX - 32-bit address of media associated with entity to be queried.

CL - Information class, as follows:

MEDIA_QUERY_DRIVER - Return information about MTD.
MEDIA_QUERY_PROBE - Probe media to verify correct identity.
MEDIA_QUERY_BLOCKSIZE - Return erasable unit size as log2(bytes).
MEDIA_QUERY_DATAPATH - Return data path width in bytes.

Output Parameters:

CY - set if failure, else clear if success.

For **MEDIA_QUERY_DRIVER** requests:

AX - Bitmask of capabilities, as follows:

MEDIA_CAPABILITY_PROBE - MTD supports probe media subfunction.
MEDIA_CAPABILITY_BLOCKSIZE - MTD can return the media's blocksize.
MEDIA_CAPABILITY_DATAPATH - MTD can return the data path width.
MEDIA_CAPABILITY_MODIFY - Writes can modify sectors (NOR Flash).
MEDIA_CAPABILITY_AUTOERASE - Writes automatically erase blocks.

For **MEDIA_QUERY_PROBE** requests:

CY - Set if media not present or incompatible, else clear.

For **MEDIA_QUERY_DATAPATH** requests:

AH - Bits per part (8, 16, 32, etc.)
AL - Interleave factor (1, 2, 4, etc.)

For **MEDIA_QUERY_BLOCKSIZE** requests:

AX - erasable unit size in log2(bytes). (6 means 64, 7 is 128, 8 is 256, etc.)

Unpreserved Registers:

Flags.

13.1.3 MTDHLP API

The MTDHLP API provides a set of standard tools for MTD writers to simplify MTD design. Use of the MTDHLP API functions over ad hoc methods allows MTDs to leverage existing working code to perform mode switches, translate addresses, execute microdelays, enable and disable Vpp, and perform other necessary tasks.

While it is possible to write an MTD without calling the MTDHLP API functions, it would be foolish to do so. Consider that MTDHLP functions that perform address translation leverage existing code that manages windowing in the chipset or high-integration CPU. The mode switching routines eliminate the need to write such mechanical code and allow the OEM instead to focus on the media programming task at hand. Finally, Vpp controlling functions provide the delayed Vpp disable mechanism, boosting your MTD's performance without having to complicate the MTD's code paths.

The MTDHLP API functions are not directly callable from application programs; they are always called from within the MCL itself, in response to its clients' requests.

The MTDHLP API has functions that operate in real mode, protected mode, or both modes. Not all functions are designed to operate in both modes. Consult the individual functional description for details.

Registers must be preserved unless they are used to return values in specific cases. The carry flag (CY) is used to indicate either a successful or failing outcome from a function call. In one case (**EraseComplete**), the CY flag is used to indicate the status of an ongoing operation.

MTDs should keep stack depth to a minimum. A suggested maximum amount of stack usage by the MTD is 64 bytes. Do not assume that it is acceptable to allocate data buffers or other such data structures on the stack in an MTD.

13.1.3.1 MtdHlpToProt API Function

The **MtdHlpToProt** function is called with procedure linkage to switch the mode of the processor from real mode to protected mode. At the time the mode switch occurs, the DS and ES registers are loaded with a value that can be used to address the entire 4.2GB memory address space in the CPU.

This function must be called from real mode only; it may not be called from V86 mode.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.
DS - 4K-granular (32-bit) selector that maps to physical address 00000000h.
ES - 4K-granular (32-bit) selector that maps to physical address 00000000h.

Unpreserved Registers:

Flags.

13.1.3.2 MtdHlpToReal API Function

The **MtdHlpToReal** function is called with procedure linkage to switch the mode of the processor from protected mode to real mode. At the time the mode switch occurs, the DS and ES registers are destroyed, since in protected mode they contained a selector that is unusable in real mode.

This function must be called from protected mode only. It may not be called from V86 mode.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags, DS, and ES.

13.1.3.3 MtdHlpMapAddress API Function

The **MtdHlpMapAddress** function is called with procedure linkage to translate a 32-bit media address into either a 16:16 real-mode address of a windowed area managed by board-specific hardware, or a 32-bit physical address that can be used as a 32-bit offset with respect to a 4K granular selector mapping physical address 000000h of the target.

This function may be called from protected mode or real mode.

Input Parameters:

DX:AX - 32-bit media address.

Output Parameters:

CY - set if real mode mapping, else clear if physical mapping.

If real mode mapping:

ES:DI - 16:16 real-mode translated address, and
CX - number of mapped bytes visible at that address

If protected mode mapping:

EDI - 32-bit offset relative to start of physical memory.

Unpreserved Registers:

Flags.

13.1.3.4 MtdHlpMapReal API Function

The **MtdHlpMapReal** function is called with procedure linkage to translate a real-mode 16:16 address to a 32-bit physical address so that the real mode address can be accessed from protected mode.

This function may be called from protected mode or real mode.

Input Parameters:

BP:BX - 16:16 real-mode address.

Output Parameters:

ESI - 32-bit offset relative to start of physical memory.

Unpreserved Registers:

Flags.

13.1.3.5 MtdHlpQueryRegion API Function

The **MtdHlpQueryRegion** function is called with procedure linkage to determine where a region as defined with the **MEDIA_REGION** macro starts and how big it is. This is needed by some MTDs such as the bulk Flash MTD, which must perform programming on an entire part, not just a portion thereof.

This function may be called from protected mode or real mode.

Input Parameters:

DX:AX - 32-bit address within region to query.

Output Parameters:

DX:AX - 32-bit address of first byte within region queried.
DI:SI - count of bytes within region.

Unpreserved Registers:

Flags.

13.1.3.6 MtdHlpDelay API Function

The **MtdHlpDelay** function is called with procedure linkage to perform a delay approximating a specified number of microseconds. This routine exists so that MTDs do not invent their own versions of this routine, so that only one piece of code in the system is used to perform fine timing.

This function may be called from protected mode or real mode.

Input Parameters:

CX - number of microseconds to delay.

Output Parameters:

None.

Unpreserved Registers:

Flags.

13.1.3.7 MtdHlpEnableVpp API Function

The **MtdHlpEnableVpp** function is called with procedure linkage to enable Vpp and/or write enable lines to make sure that the media is writable. In some cases this may involve waiting a

significant amount of time (100s of microseconds); therefore, it is recommended that this routine be called from real mode, not protected mode, to avoid high interrupt latency.

Enabling of external voltages is totally board-specific, and therefore this routine calls the board module's routine to perform the work. However, the board module's routine always performs the necessary delay to wait for voltage to ramp-up before returning. Since this delay on every I/O could severely impact back-to-back writes in most systems, MCL uses a timeout to keep Vp high even after the MTD commands it to be disabled with a call to **MtdHlpDisableVpp**. This allows background erases to continue, and then after some time (a few seconds), Vpp is lowered automatically by MCL.

This function may be called from protected mode or real mode, but protected mode calls should be avoided to reduce interrupt latency.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

13.1.3.8 MtdHlpDisableVpp API Function

The **MtdHlpDisableVpp** function is called with procedure linkage to start a timer which, once expired, disables Vpp and/or write enable lines to make sure that the life of the battery supplying Vpp is conserved.

Enabling of external voltages is totally board-specific, and therefore this routine calls the board module's routine to perform the work. However, the board module's routine always immediately disables power, and to prevent ongoing erases from being aborted, this function starts a timer which, once expired, calls the board module's **BoardDisableWrites** routine.

This function may be called from protected mode or real mode, but protected mode calls should be avoided to reduce interrupt latency.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

13.1.3.9 MtdHlpRead API Function

The **MtdHlpRead** function is called with procedure linkage to read from most memory-mapped media that do not require special protocols to read data. RAM, ROM, and NOR Flash are examples of types of media which fall into this category. Parts like NAND Flash have a different access method and are not accessible with this function.

This function's purpose is to establish a pre-written, tested read routine that can be used by the majority of MTDs to speed MTD development, and allow reuse of the same code at the same time. Calling this function is a matter of convenience, and is not a requirement of any MTD's design.

This function may be called from real mode only, not protected or V86 mode.

Input Parameters:

> DX:AX - 32-bit media address of first byte of media to read.
> BP:BX - 16:16 real-mode address of first byte of user buffer.
> CX - number of bytes to read.

Output Parameters:

> CY - set if failure, else clear if success.

Unpreserved Registers:

> All but SS and SP.

13.2 Media Technology Drivers (MTDs)

The EMBEDDED BIOS MTD architecture offers complete flexibility for support of virtually any media type. An MTD can present a logical media that might appear to be read/write storage, but in fact might actually be implemented as an RS-232 or Ethernet protocol with a host, or accesses to a shared memory subsystem. MTDs simply implement their APIs, using available MTDHLP API functions to ease their jobs wherever possible.

When required to address random-access storage such as ROM, RAM, or Flash, MTDs must be prepared to perform their work in real mode or in protected mode, although it is up to the MTD to request mode switching at appropriate times. Consider that an MTD might be optimized for bulk data transfer at the expense of interrupt latency, by switching to protected mode during its entire processing of a data transfer. Another MTD might be optimized to minimize interrupt latency at the expense of bulk data transfer performance, by performing the transfer in a series of short steps, each of which might involve a switch to protected mode and back again. This flexibility allows for tradeoffs to be made without compromising the EMBEDDED BIOS architecture.

13.2.1 MTD Architecture

The purpose of an MTD is to provide a specific set of services for a class of media, hiding the programming details of the media from the MCL. MTDs are typically small, minimalistic, and are procedural, rather than focussed on the minutia of register manipulation.

MTDs must be capable of processing requests in real or protected mode, depending on the results of calls to **MtdHlpMapAddress**. This is the case because the BPM may determine that the media is best mapped in a window, or in the upper physical addresses in memory.

MTDs must minimize their processing in protected mode so that interrupt latency is kept as low as possible. When a new MTD is written, block copies of data normally performed in protected mode should be broken up into minimal-sized pieces (say, 1KB), so that the MTD gives the system a chance to accept and process interrupts between handling of data movement in protected mode.

MTDs may take part in the system's overall power management, if they are included in the power device tree table created with the **POWER_DEVID** macro. When specified in this table, MTDs must accept and handle power management requests through their corresponding power control entrypoint, documented later in this section.

Some MCL functions may not be valid operations for certain media, in which case the MTD either simulates the function and indicates to MCL that the operation was performed successfully, or simply returns with failure to the MCL, indicating that the operation cannot be performed. For example, the RAM MTD simulates block erasure by filling a quantity of RAM with a special data value (0xffff). However, the ROM MTD cannot simulate writes or erases, since ROM by definition is not changable. In this case, the ROM MTD must return failure for **WriteBlock**, **StartErase**, and **LockBlock** requests.

13.2.2 MTD Entrypoints

There are two basic classes of requests that an MTD may process; power management requests from the power management subsystem, and media I/O requests from MCL. There are five function types provided for MCL. MTDs should be coded in such a way that additional request types received from MCL are treated as failing by setting the CY flag and returning without further processing. This allows the MCL architecture to grow while continuing to support down-level drivers.

13.2.2.1 MTD Request Entrypoint

The **MtdSvc*Xxx*** function (where *Xxx* is the name of the MTD) is called with procedure linkage by the MCL to submit an I/O request to the MTD.

All registers required by the MTD function are passed in the appropriate registers documented later in this section. Additionally, a function code is passed in the (SI) register to identify the request type.

Input Parameters:

SI - Request type, as follows:

0 - MEDIA_CMD_READ (read request).
1 - MEDIA_CMD_WRITE (write request).
2 - MEDIA_CMD_START_ERASE (start erase request).

3 - MEDIA_CMD_ERASE_COMPLETE (erase complete query).
4 - MEDIA_CMD_LOCK_BLOCK (lock block request).

All other registers - As documented for each function.

Output Parameters:

None.

Unpreserved Registers:

Flags.

13.2.2.2 MTD Power Management Entrypoint

The **Mtd*Xxx*PwrLevel** function (where *Xxx* is the name of the MTD) is called with procedure linkage by the EMBEDDED BIOS Power Management System to coordinate the MTD's power with the rest of the system's state.

When implemented for an MTD, this routine is responsible for control over the power management state of the devices managed by the MTD. For example, some Flash devices can be placed in a low-power mode where they consume little or no power, but cannot respond to requests.

MTDs need not support their power management function, unless they are to be included in the power management device tree.

Input Parameters:

DS - Points to the extended BIOS data area (EBDA).
BX - Device index.
CL - New power level.
CH - Old power level.

Output Parameters:

None.

Unpreserved Registers:

Flags.

13.2.3 Dispatching to Function Handlers

It is recommended that MTDs use a dispatch table indexed by request type to dispatch to individual function handlers within the MTD. This design facilitates expansion without causing the per-request path to get longer. Although MCL does not require this design, it may be the best performer and at the same time keep maintenance costs low. Below is an example dispatch routine (**MtdSvcXxx**) that dispatches through a command table. For sample code on-line, see the SYSTEM\MTDRAM.ASM file.

```
BIOS_  SEGMENT

;      The following table is used to dispatch to function
;      handlers by command ordinal.  MEDIA.INC contains the
;      ordinal assignments.

FUNC   MACRO  cmd, rtn
       ORG    CmdTbl+(2*cmd)
       dw     OFFSET BIOS_GRP:_p_&rtn
       ENDM

CmdTbl dw  (MAX_MEDIA_REQUEST+1) dup (OFFSET BIOS_GRP:_p_BadCommand)
       FUNC   MEDIA_CMD_READ, ReadBlock
       FUNC   MEDIA_CMD_WRITE, WriteBlock
       FUNC   MEDIA_CMD_START_ERASE, StartErase
       FUNC   MEDIA_CMD_ERASE_COMPLETE, EraseComplete
       FUNC   MEDIA_CMD_LOCK_BLOCK, LockBlock
       ORG    CmdTbl+(2*(MAX_MEDIA_REQUEST+1))
CMDTBL_LENGTH = ($-CmdTbl)

BIOS_  ENDS

DefProc MtdSvcXxx, PUBLIC
;      PRINTF <MtdSvcXxx: entered, SI=$x.\n>, <si>
       add    si, si              ; (SI) = word index into table.
       cmp    si, CMDTBL_LENGTH   ; is the index within range?
       jae    MtdSvcXxx_Fail             ; if not, fail this call.

;      Transfer control to individual routine, which when
;      it returns, will return to OUR caller, avoiding an
;      extra RET in the path.

       jmp    word ptr CmdTbl [si]       ; calls routine.

;      Come here if we received an invalid request.

MtdSvcXxx_Fail:
       stc
EndProc MtdSvcXxx
```

13.2.4 Protected-Mode and Real-Mode Control Paths

MTDs must be prepared to perform accesses to mapped addresses in either real mode or protected mode, at MCL's discretion. Recall that 32-bit media addresses as provided by MCL clients are not specifically tied to 32-bit physical addresses, but may be assigned to the extended memory physical address space accessible in protected mode, or windowed into a low memory segment below the 1MB address marker. If the MTD will support media that can be windowed with a hardware MMU such as that found in AMD SC300, SC310, SC400, or SC410 CPUs, then the MTD must be able to handle either protected mode or real mode addressing.

If the MTD does not support memory-mapped media, then only one control path need be provided by the MTD. For example, if an MTD were developed to route requests over an RS-

232 line to a host PC, then the I/O instructions necessary to program the UART could be performed just as easily in real mode as protected mode. Since all MCL requests start in real mode, there is no need in such an MTD for a protected mode path.

When both real mode and protected mode paths are required, the MTD must perform the mode switching at the times it deems necessary. A good simple example of this dual mode processing is illustrated by the following code fragment, taken from the RAM MTD's **WriteBlock** routine:

```
DefProc WriteBlock
       mov    si, cx              ; (SI) = # bytes to transfer.

;      Copy (SI) bytes from 16:16 address (BP:BX) to logical address (DX:AX).

WriteBlock_Loop:
       or     si, si              ; are there any bytes left to copy?
       clc                        ; assume success.
       jz     WriteBlock_Exit     ; return with success if none left.

       Pcall  MtdHlpMapAddress    ; does the media address mapping.
       jc     WriteBlock_Real     ; if real mode, (CX) = # bytes mapped.

;      We have a protected mode translation with media=(0:EDI), (SI)=count.

       IF     OPTION_SUPPORT_PROTECT_MODE
       mov    cx, si              ; (CX) = # bytes to transfer.

       Pcall  MtdHlpMapReal       ; converts (BP:BX) to (ESI).

;      Copy (CX) bytes of memory in protected mode from (0:ESI) to (0:EDI).

       Pcall  MtdHlpToProt        ; (DS)=(ES)=phys0.

       push   cx
       movzx  ecx, cx             ; (ECX) = # bytes to copy.
       shr    ecx, 1              ; (ECX) = # words to copy.
       rep    movs word ptr [edi], word ptr [esi] ; copy words 1st.
       rcl    ecx, 1              ; (ECX) = # bytes to copy.
       rep    movs byte ptr [edi], byte ptr [esi] ; remaining byte.
       pop    cx

       Pcall  MtdHlpToReal        ; back to real mode.
       clc                        ; we were successful.
       ELSE
       stc
       ENDIF  ; (OPTION_SUPPORT_PROTECT_MODE)
       jmp    WriteBlock_Exit     ; do the next block.

;      We have a media window at (ES:DI) that is (CX) bytes in length,
;      and the total amount of data left to copy to the user's buffer
;      is (SI) bytes. Compute MIN(SI,CX) and copy from (BP:BX) to (ES:DI).

WriteBlock_Real:
       cmp    si, cx              ; is (SI) > (CX)?
       ja     @f                  ; if so.
```

```
        mov     cx, si              ; (CX) = smallest.
@@:
        sub     si, cx              ; (SI) = remaining data to copy.
        add     ax, cx
        adc     dx, 0               ; (DX:AX) = next media address.

;       Prepare registers for copying.

        push    si                  ; save # bytes left to copy.

        mov     ds, bp
        mov     si, bx              ; (DS:SI) = ptr, user buffer.
        add     bx, cx              ; (BP:BX) = ptr, next user buffer.

;       Now do the copy operation itself.

        shr     cx, 1
        rep     movsw               ; copy by words first.
        rcl     cx, 1
        rep     movsb               ; copy last byte, if any.
        pop     si                  ; (SI) = restored # bytes to copy.
        jmp     WriteBlock_Loop     ; do the next block.

WriteBlock_Exit:
EndProc WriteBlock
```

In this code fragment, a loop is used to define the state where some data must be copied from the user buffer to the media. This contemplates a situation where either the protected mode or real mode paths may not complete the entire I/O operation, but rather may only complete a partial I/O, leaving the rest of the entire transaction to another cycle around the loop.

The SI CPU register is used to keep track of the number of bytes left to transfer, and the physical address of the next media address to be affected is kept in the DX:AX register pair, conveniently so that **MtdHlpMapAddress** can use the address as input to map the next chunk of media. If the CY flag is set by **MtdHlpMapAddress**, then the MTD knows that real-mode processing must occur. Otherwise, control continues through the protected mode path, which translates the real-mode 16:16 user buffer address to a physical address, and then switches to protected mode, performs the data movement, and switches back to real mode again.

The real mode code path is called to process a portion of the I/O, since the requested I/O size may be bigger than the memory window provided by the mapping hardware. The code performs an I/O that has a size equal to the minimum of (a) the window size and (b) the remaining size of the I/O. Control then passes back to the top of the main loop, which continues to process the rest of the I/O.

Note that MTDs must take care not to second-guess the mapping method that **MtdHlpMapAddress** may return on subsequent calls. It may be easy for that function to dictate that a portion of an I/O occur in protected mode, whereas another portion of the same I/O occur in real mode.

13.2.5 Adding a Custom MTD to the Board Personality Module

The MTDs provided with the core BIOS are implemented in assembly modules in the **SYSTEM** subdirectory; however, new MTDs need not occupy their own .ASM files in this directory. When the OEM needs to implement a new MTD for a specific design, it is best to avoid modifying the core BIOS files, to minimize the effort required to upgrade the core BIOS at any time.

The preferred approach is to create the MTD a routine at a time directly in the BPM itself. The fact that the MTD code is interspersed among BPM routines will not affect the operation of the core BIOS. Some OEMs may choose to split-off the MTD code into a separate file that is included by the BPM file; this is also acceptable because it doesn't require changes to the MAKEFILE, linker response file, and **SYSTEM** directory.

When adding your MTD to the BPM, proper segmentation should be employed. MTD code should be assembled in the **BIOS** segment, as illustrated by the code in the **MTDRAM.ASM** file.

13.2.6 Adding Windowing to the Board Personality Module

Although the **BoardMapAddress** function is specified more formally in Chapter 20, a few words about how this routine interplays with the MTD, are worthwhile here. This routine, found in the BPM, is called by MCL on behalf of an MTD when it issues an **MtdHlpMapAddress** function call. It is up to **BoardMapAddress** to decide whether the MTD will use protected mode or real mode to access the media, and then where the real mode window will be, etc.

Unless overridden with an OEM-specified **BoardMapAddress** function, the default version of this routine just calls **CsMapAddress** in the chipset module, so as to take advantage of any MMU in the chipset (Note to AMD users: The SC300, SC310, SC400, and SC410 are chipset-like in this regard and have **CsMapAddress** routines that program the MMU hardware).

When the OEM contemplates an override routine to perform some action that is different from the default, there are many issues to consider. Of course, the address space architecture is an important issue. For example, the SC400 and SC410 CSPMs segment the 32-bit address space into four parts, each of which is used to logically map different ROM Chip Select lines. The OEM might decide that a certain range of media addresses should be handled with one MMU, and another range handled with a different one. Or, under certain conditions (i.e., if the MMU(s) are busy with application data), then protected mode might be returned. While many possible issues come up and are dealt with in this routine, ultimately it must return its decision about how the MTD processes requests in a simple response: protected mode or real mode, and when real mode is returned, where the window is and how much is visible within the window.

13.2.7 MTD I/O Request Interface

I/O requests are submitted to MTDs through a single interface procedure. The name of this procedure is significant, and is formed by concatenating the string **MtdSvc** with the name of the MTD as specified in the **MEDIA_REGION** table entries to be associated with this MTD.

This routine is called in real mode with a near call from within MCL. Interrupts are enabled at the time of the call, and must be enabled upon return to MCL. Upon entry, the SI register contains the media function code, used to dispatch to the appropriate request handler within the MTD. This register need not be preserved by the dispatch routine.

13.2.7.1 LockBlock MTD Procedure

The MTD's private **LockBlock** function is called by the MTD's dispatch function to lock a block of storage in a sectored or boot block Flash device. Once a block has been locked, it remains write-protected until unlocked by a subsequent erase operation.

Some MTDs may or may not support the lock function, and some may not even support the erase functions. For example, the **Rom** MTD does not support this function because it cannot change the underlying media. The **Ram** MTD supports erase by emulating it, assuming the same block size as specified for the RFD, but has no way to "lock" a block of RAM.

Input Parameters:

DX:AX - 32-bit address of 1st byte within block to be locked.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

13.2.7.2 StartErase MTD Procedure

The MTD's private **StartErase** function is called by the MTD's dispatch function to start a block erase operation on a block of storage in a sectored, boot block, or bulk erase Flash device. Once the block is erased, its contents are reset by the device such that each byte contains the hexadecimal value, ffh (all ones).

This function's purpose is to begin erase processing, and optimally return before the erasure has completed, so that the erase processing can occur in the background while the system continues with other operations. This can be effective in increasing RFD performance. Some MTDs may not implement an asynchronous erasure operation, and instead implement the entire erase functionality in the **StartErase**, so that the **EraseComplete** routine always returns to the caller with a "completed" status. For devices without background erase capabilities, this routine should complete synchronously as described.

Certain MTDs provided with the core EMBEDDED BIOS software illustrate how to break-up the processing of starting the erase process and determining if the erase process has completed. One example of such an MTD is `MTDINTA.ASM`. Note that this MTD (necessarily) handles situations where, once an erase process has started and control has returned to the client, another request is received to perform a read or write before the erase operation has completed. In these cases, the erase operation must either be suspended during the secondary operation's performance, or alternatively, the erase operation may be concluded before processing the secondary operation. All MTD operations must be coordinated to handle such background processing gracefully without loss of data.

Some MTDs may or may not support the erase functions (either **StartErase** or **EraseComplete**). For example, the **Rom** MTD does not support this function because it cannot change the underlying media. The **Ram** MTD does support the function by emulating it, assuming the same block size as specified for the RFD.

Input Parameters:

DX:AX - 32-bit address of 1st byte within block to be erased.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

13.2.7.3 EraseComplete MTD Procedure

The MTD's private **EraseComplete** function is called by the MTD's dispatch function to determine if the erasure operation started by a previous call to the **StartErase** function has completed or is still in progress.

Some MTDs may not implement an asynchronous erasure operation, and instead implement the entire erase functionality in the **StartErase**, so that the **EraseComplete** routine always returns to the caller with a "completed" status.

Some MTDs may or may not support the erase functions (either **StartErase** or **EraseComplete**). For example, the **Rom** MTD does not support this function because it cannot change the underlying media. The **Ram** MTD does support the function by emulating it, assuming the same block size as specified for the RFD.

Input Parameters:

DX:AX - 32-bit address of the block as specified in the call to **StartErase**.

Output Parameters:

CY - clear if no erase operation in progress, else set.

Unpreserved Registers:

Flags.

13.2.7.4 ReadBlock MTD Procedure

The MTD's private **ReadBlock** function is called by the MTD's dispatch function to read data from the media.

Commonly, MTDs call the **MtdHlpRead** function to perform the transfer, when the media is accessible as direct-access storage within the memory address space. Making use of this function in new MTDs saves space by reusing existing code, and also takes advantage of its method of breaking the read into pieces when in protected mode to reduce interrupt latency.

Sometimes, MTDs cannot use this helper function, because device programming is required. For example, some Flash devices may be controlled through a range of I/O ports, and may not actually present any directly-addressible memory regions to the CPU.

Input Parameters:

DX:AX - 32-bit address of 1st byte of media to be read.

BP:BX - 16:16 segment offset pointer to 1st byte of user buffer where data will be written in RAM.

CX - Number of bytes to be transferred. A zero value indicates 65,536 bytes. The value does not have to be even.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

13.2.7.5 WriteBlock MTD Procedure

The MTD's private **WriteBlock** function is called by the MTD's dispatch function to write data from a user-specified buffer to the media.

The caller must take care that the writing is possible; i.e., that the area of the media to be written does not contain other data that has not been erased, unless the underlying media supports writing without erasing. NOR Flash devices require erasure of large blocks, whereas NAND Flash devices can be supported by MTDs that build-in an automatic pre-erase operation. Similarly, the RAM MTD can write to an area that has not previously been erased.

Some MTDs may or may not support the write function (or erase functions). For example, the **Rom** MTD does not support this function because it cannot change the underlying media.

Input Parameters:

DX:AX - 32-bit address of 1st byte of media to be written.

BP:BX - 16:16 segment offset pointer to 1st byte of user buffer containing data to be written.

CX - Number of bytes to be transferred. A zero value indicates 65,536 bytes. The value must be even.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

13.2.7.6 Query MTD Procedure

The MTD's private **Query** function is called by the MTD's dispatch function to return information to the caller about the media, the driver, or the data path to the media. Additional information classes may be made available in future releases.

Some MTDs may or may not support the Query function; its implementation is optional.

Input Parameters:

DX:AX - 32-bit address of media associated with entity to be queried.

CL - Information class, as follows:

MEDIA_QUERY_DRIVER - Return information about MTD.
MEDIA_QUERY_PROBE - Probe media to verify correct identity.
MEDIA_QUERY_BLOCKSIZE - Return erasable unit size as log2(bytes).
MEDIA_QUERY_DATAPATH - Return data path width in bytes.

Output Parameters:

CY - set if failure, else clear if success.

For **MEDIA_QUERY_DRIVER** requests:

AX - Bitmask of capabilities, as follows:

MEDIA_CAPABILITY_PROBE - MTD supports probe media subfunction.
MEDIA_CAPABILITY_BLOCKSIZE - MTD can return the media's blocksize.
MEDIA_CAPABILITY_DATAPATH - MTD can return the data path width.
MEDIA_CAPABILITY_MODIFY - Writes can modify sectors (NOR Flash).
MEDIA_CAPABILITY_AUTOERASE - Writes automatically erase blocks.

For **MEDIA_QUERY_PROBE** requests:

CY - Set if media not present or incompatible, else clear.

For **MEDIA_QUERY_DATAPATH** requests:

AH - Bits per part (8, 16, 32, etc.)
AL - Interleave factor (1, 2, 4, etc.)

For **MEDIA_QUERY_BLOCKSIZE** requests:

AX - erasable unit size in log2(bytes). (6 means 64, 7 is 128, 8 is 256, etc.)

Unpreserved Registers:

Flags.

13.2.7.7 Init MTD Procedure

The MTD's private **Init** function is called by the MTD's dispatch function during the POST system initialization to initialize the MTD as necessary to perform its other functions. No other requests will be made before the MTD receives an Init request.

This call allows MTDs that need to acquire resources such as system memory to obtain the memory before the operating system loads. At this time, MTDs using hardware that must be initialized, such as PC Card controllers or network interface units, may perform this work.

Some MTDs may or may not support the Init function; its implementation is optional.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

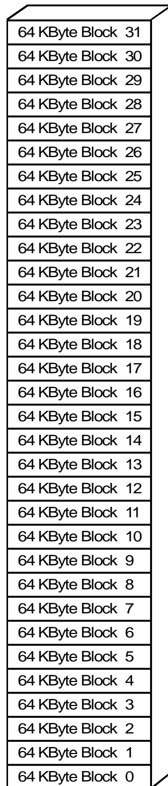
13.3 MEDIA_REGION Addressing Table

As described in Chapter 7, the **MEDIA_REGION** macro is used by the OEM in the Project file to build a table of media address regions that specify to MCL which MTDs handle which media addresses. Be certain to read the discussion of the **MEDIA_REGION** macro in that chapter to learn how to construct the table.

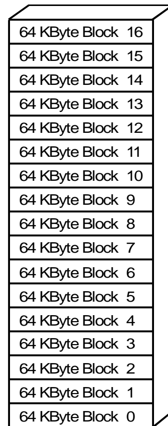
MTDs do not have direct access to the contents of the **MEDIA_REGION** table. Instead, this table belongs to MCL. MTDs can call the **MtdHlpQueryRegion** function for certain situations, to identify the starting media address of a region. This is useful in at least one common case, the erasure of bulk Flash memory, since the entire region of Flash must be written with the value 0 before it can be erased.

When routing requests, MCL searches the table linearly, looking for the first region that contains a specified media address. As a fall-back, the MCL maintains a special entry at the end of the table that maps the entire 32-bit media address space to the RAM MTD, leaving no request unsatisfied. This has the added advantage that no **MEDIA_REGION** table need be specified by the OEM to just test the RFD or RAM disk with RAM media, and also that no special case code that provides for a "no such region" error need be executed on every I/O.

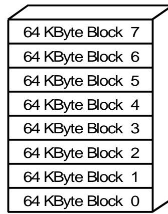
13.4 Common Flash Device Layout



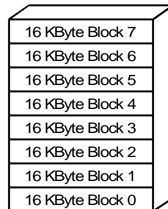
Intel 28F016SA
Memory Map



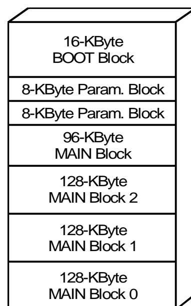
Intel 28F008SA
Memory Map



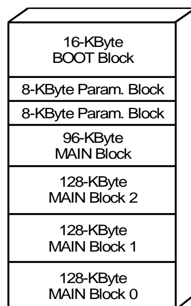
AMD Am29F040
Memory Map



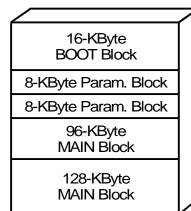
AMD Am29F010
Memory Map



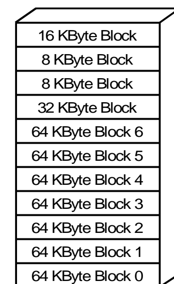
Intel 28F400BX-T
Memory Map



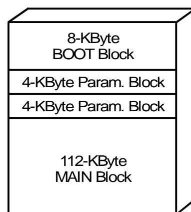
Intel 28F004BX-T
Memory Map



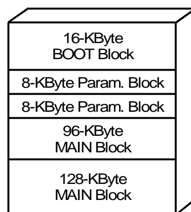
Intel 28F200BX-T
Memory Map
No RFD



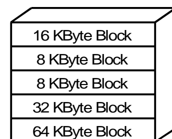
AMD Am29F400T
Memory Map



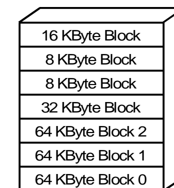
Intel 28F001BX-T
Memory Map
No RFD



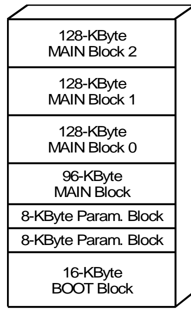
Intel 28F002BX-T
Memory Map
No RFD



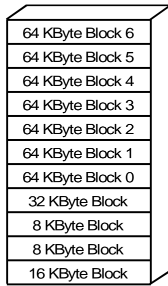
AMD Am29F100T
Memory Map
No RFD



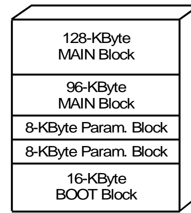
AMD Am29F200T
Memory Map



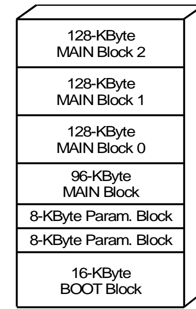
Intel 28F004BX-B Memory Map



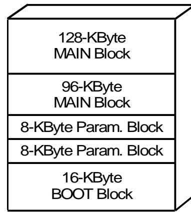
AMD Am29F400B Memory Map



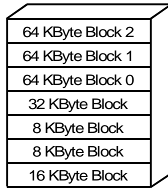
Intel 28F002BX-B Memory Map No RFD



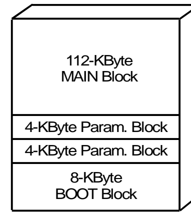
Intel 28F400BX-B Memory Map



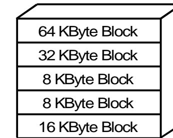
Intel 28F200BX-B Memory Map No RFD



AMD Am29F200B Memory Map



Intel 28F002BX-B Memory Map No RFD



AMD Am29F100B Memory Map No RFD

Chapter 14

MANUFACTURING MODE

Most volume consumer electronics products such as phones or PDAs need an electronic interface to the manufacturing equipment or test equipment that is used to manufacture, test, and repair the devices in the field. The EMBEDDED BIOS Manufacturing Mode enables a host computer to establish a high-speed communications link with the target device running EMBEDDED BIOS, so that the device can be tested and its Flash memory can be updated without running an operating system on the target.

An additional feature of Manufacturing Mode is the support of INT 13h drive redirection, so that host software (such as the supplied MFGDRV.SYS DOS device driver) can allow the field support technician to access files on a target's disk as though it were a local INT 13h device, for the purposes of updating software on the unit. This feature works with all drive types on the target, including ROM, RAM, RFD, Floppy, and IDE types (of course, the ROM disk only permits read access).

14.1 Entering Manufacturing Mode

In order for a target to support Manufacturing Mode, the **OPTION_SUPPORT_MFGMODE** option must be enabled. Then, one of three events must occur on the target to cause Manufacturing Mode to be activated. Once Manufacturing Mode is entered, the console keyboard and screen remain unresponsive until the target is rebooted again, or Manufacturing Mode receives a "boot target" command from the host PC.

The simplest way to enter Manufacturing Mode is to boot the target, enter the Setup main menu, and select **ENTER MANUFACTURING MODE**. This can even be selected when the console I/O is redirected over the same serial port that Manufacturing Mode uses. The BIOS build option, **OPTION_SETUP_MFGMODE**, must be enabled for this main menu item to appear.

The user can cause Manufacturing Mode to be entered as a boot action by selecting it in the BASIC Setup screen. EMBEDDED BIOS provides for four different boot activities, occurring one after another, so that it can attempt to boot from different drives, then perhaps DOS in ROM, and as a last resort, Manufacturing Mode or the debugger. This allows an embedded device to make its best efforts to boot properly, and if no booting of the operating system is possible, Manufacturing Mode can take over and allow field service personnel to diagnose the device.

Another way for Manufacturing Mode to be invoked is by inspection of a hardware signal in the target during POST. This method relies on the OEM writing some simple code that is inserted into the **BoardTestMode** routine in the BPM. The routine is called during POST right before the first boot action is performed, to determine if a special condition exists that should cause Manufacturing Mode to be activated. If the routine returns with the CY flag set, then POST enters Manufacturing Mode. If CY is clear, then POST continues with the first boot action. Usually, the OEM's proprietary code performs a test of a bit in an I/O port to make the decision about whether to set or clear the carry flag. Often, the bit and I/O port are associated with a UART's Line Status Register (LSR) or Modem Status Register (MSR), so that the BIOS can use the presence of a communications cable as an indication that a link is desired.

Another way for EMBEDDED BIOS to enter Manufacturing Mode is when a critical POST error occurs, and **OPTION_CRITICAL_MFGMODE** is enabled. Critical errors are those that normally cause the speaker to beep on a desktop PC; for example, interrupt controller or DMA controller test failures, or low memory failures during POST.

Manufacturing Mode can continue indefinitely (until power is removed or a message requesting the target to reboot is received), or it can time-out whenever the test mode hardware has stopped responding. If **OPTION_MFGMODE_TIMEOUT** is enabled, then Manufacturing Mode continually tests the I/O port used for the determination of whether Manufacturing Mode should be entered (see above). If the target determines that it has been removed from the "test jig", then it times-out the Manufacturing Mode and continues to boot the system.

14.2 Host PC Operation

Once the target has entered Manufacturing Mode, the host PC may cause the target to perform functions by issuing commands in protocol over the RS-232 connection. There are two ways to access the target from the host PC.

The first way is to run a program that accesses the host-side Manufacturing Mode functions. An example of such a program is `HOST.EXE`, found in the `UTIL` subdirectory. This program runs under DOS and, using a full screen windowing interface, illustrates the basic functionality of the Manufacturing Mode protocol. It should be noted that this program is a working example program, and is not intended to be a production-quality control tool.

The second way to access the target through Manufacturing Mode is to install the `MFGDRV.SYS` device driver, also found in the `UTIL` subdirectory. This device driver loads under MS-DOS and Embedded DOS-ROM, and maps a new drive letter on the host to a drive on the target.

14.2.1 Sample Manufacturing Mode HOST Program

The example program source that calls the Manufacturing Mode API functions can be found in `UTIL\HOST.C`. This program is a full-screen DOS program that uses COM1 (3f8h) on the host PC to communicate with the target.

The full-screen interface is provided by the Character Oriented Windowing (COW) interface libraries in the `COW` subdirectory.

To watch this program work, prepare an EMBEDDED BIOS adaptation that has **OPTION_SUPPORT_MFGMODE** enabled, **OPTION_SUPPORT_SETUP** enabled, and

OPTION_SETUP_MFGMODE enabled. Also, disable **OPTION_MFGMODE_TIMEOUT**, so that you will have ample time to place the target in test mode before using the host utility to access Manufacturing Mode functions on the target. Finally, select the serial port and baud rate to be used on the target by specifying **CONFIG_MFG_BAUD** for the baud rate, **MFG_COM_BASE** for the UART base address, **MFG_INT_VECT** for the interrupt vector associated with the UART's interupt driven receives. Choose 56K baud for the baud rate to start, as the sample `HOST.EXE` program uses that baud rate by default.

Then, run the HOST program, so that its main menu is displayed.

Next, boot the target, enter the SETUP screen system, select **ENTER MANUFACTURING MODE**. On the host, select **GET TARGET ATTENTION**, within a couple seconds of selecting the manufacturing mode on the target. You should see the host program immediately display a yellow status box that shows that the connection has been established.

If the connection hasn't been established, then try the connection again on the host side, or reboot the target and try again, this time having the host program get the target's attention within two seconds or so of the target's entering of manufacturing mode. If this still fails, check the baud rate for the manufacturing mode on the target to be sure it is using the right baud rate (56K baud or 115K baud, depending on your version of HOST), and that its serial port is working properly.

14.2.2 Manufacturing Mode Drive Redirection

The INT 13h redirection support in the Manufacturing Mode protocol can be exposed by loading the `MFGDRV.SYS` device driver on the host by using the following `CONFIG.SYS` line:

```
DEVICE=MFGDRV.SYS /BAUD=rate /PORT=COMn /UNIT=u /AUTO
```

This device driver runs under any DOS-compatible operating system, and creates a drive letter on your host PC (usually D: if your last hard drive is C:) that can be used to interact with the specified INT 13h unit.

The *u* parameter specifies the BIOS unit number of the floppy disk, RAM disk, RFD drive, or ROM disk to be redirected, where 0 corresponds to drive A: and 1 for drive B. By default, this value is 80 (a hex number without a "0x" in front or 'h' appended to it), which corresponds with the unit for the first hard drive or emulator.

The `/BAUD=rate` parameter can be used to match the baud rate used by the target's BIOS. Legal values are 19K, 28K, 38K, 56K, and 115K. If this parameter is not specified, then the baud rate is autodetected.

The `/AUTO` parameter, if specified, tells `MFGDRV.SYS` to automatically format the remote drive if it determines that it is unformatted. By default, `MFGDRV.SYS` will not automatically format the remote drive, and will instead examine the media for a pre-existing format. If not found, then `MFGDRV.SYS` asks the host PC operator if the remote drive should be formatted.

This device driver also redirects raw INT 13h requests to the redirected drive, by assigning the next local host unit to that target drive. Thus, if you have one floppy drive on your host (local host unit 0), then the next unit (1) will be assigned to the redirected drive. This is the value that is placed in the (DL) register when making INT 13h requests to the remote drive.

Because of the above mentioned INT 13h request routing, if Embedded DOS users need to install a boot record on the remote device without formatting the entire drive, `INSTBOOT.EXE` can be used, but the program requires that B: be used as the drive name, since A: refers to the host's drive A:.

IMPORTANT: `MFGDRV.SYS` assumes that other software does not reprogram the COM port being used on the host for its purposes, and that it has exclusive access to it. If you run other software, such as terminal emulation programs, they may disable the COM port UART, causing `MFGDRV.SYS` to appear to stop working. It is best to avoid running such software on the host when `MFGDRV.SYS` is loaded. Note: `HOST.EXE` is an example of such a program, since it takes over the UART for its own purposes. If you run `HOST.EXE` when `MFGDRV.SYS` is loaded, you must reboot the host PC for the `MFGDRV.SYS` driver to reestablish its control over the UART.

14.3 Host-Side Manufacturing Mode Functions

The following functions are supported by linking to the `MESSAGE.OBJ` module in the `UTIL\OBJ` directory. This .OBJ file is created by running GSMAKE in the UTIL directory. By compiling `MESSAGE.C` and linking to its object file, your application program can access the Manufacturing Mode host-side functions under program control.

All of the functions return a `BOOLEAN` value, which is actually just an unsigned short. If the value returned is 0 (`FALSE`), then the function failed. If the value returned is nonzero (`TRUE`), then the function succeeded.

14.3.1 MsgInitialize Function

The *MsgInitialize* function is called by the application to initialize the Manufacturing Mode message system to a known state.

The application uses this function to establish the I/O address of the 8250-compatible serial port to be used for communications, and the baud rate to be used. Upon successful return to the caller, the host is programmed for interrupt-driven receives of data from the target.

Request Format:

BOOLEAN MsgInitialize(
    IN USHORT *PortId***,**
    IN USHORT *BaudRate*
    );

Parameters:

PortId - A 16-bit unsigned value specifying the base I/O address of the UART to be used on the host. Examples are 0x3f8 for COM1, or 0x2f8 for COM2.

BaudRate - A 16-bit unsigned value specifying the baud rate to be used by the system. This baud rate is encoded as positive numbers as follows:

0 - Use 115K baud (not available on all UARTs).
1 - Use 56K baud.
2 - Use 38K baud.
3 - Use 28K baud.

4 - Use 19.2K baud.
5 - Use 9600 baud.

14.3.2 MsgDeinitialize Function

The *MsgDeinitialize* function is called by the application to undo the initialization established with *MsgInitialize*. Most importantly, it disables interrupt processing that was previously established for the communications port.

Request Format:

BOOLEAN MsgDeinitialize(
IN USHORT *PortId*
);

Parameters:

PortId - A 16-bit unsigned value specifying the base I/O address of the UART to be used on the host. Examples are 0x3f8 for COM1, or 0x2f8 for COM2.

14.3.3 MsgPingTarget Function

The *MsgPingTarget* function is called by the application to initiate a quick message exchange that verifies the communications link with the target. If this function returns successfully, then the link can be deemed established.

When the target's POST process is initiated and its TEST MODE hardware is enabled, it enters the Manufacturing Mode query loop, where it waits for this ping request before determining that there is no host to connect to. Thus, a ping must be issued shortly (within a few seconds) of the target's being reset.

Request Format:

BOOLEAN MsgPingTarget(
);

Parameters:

none.

14.3.4 MsgReceive Function

The *MsgReceive* function is called by the application to wait for a packet of information from the target. This function is a lower-level function used by the higher-level API functions to transport commands and responses. If the OEM needs to extend this message protocol (as defined in `INC\SERMSG.H` and `INC\SERMSG.INC`), this function will be needed.

Request Format:

BOOLEAN MsgReceive(
IN VOID ***_Buffer_,**
IN USHORT _BufferLength_**,**
**OUT USHORT ***_BytesRead_
);

Parameters:

Buffer - A 16:16 (far) pointer to a storage location where the incoming message will be stored by the message system when it arrives.

BufferLength - A 16-bit unsigned value specifying the size of the buffer in bytes.

BytesRead - A 16:16 (far) pointer to an unsigned short variable where the message system will return the actual size of the message received in bytes. If the message is smaller than the declared size of the buffer, then this value will be less than _BufferLength_.

14.3.5 MsgSend Function

The _MsgSend_ function is called by the application to send a message to the target. This function is a lower-level function used by the higher-level API functions to transport commands and responses. If the OEM needs to extend this message protocol (as defined in `INC\SERMSG.H` and `INC\SERMSG.INC`), this function will be needed.

Request Format:

BOOLEAN MsgSend(
IN VOID ***_Buffer_,**
IN USHORT _BufferLength_
);

Parameters:

Buffer - A 16:16 (far) pointer to a storage location containing the message to be sent.

BufferLength - A 16-bit unsigned value specifying the size of the buffer in bytes.

14.3.6 MsgBootTarget Function

The _MsgBootTarget_ function is called by the application to exit the Manufacturing Mode and continue on with POST to boot he operating system.

Request Format:

BOOLEAN MsgBootTarget(
);

Parameters:

none.

14.3.7 MsgGetLastPostCode Function

The *MsgGetLastPostCode* function is called by the application to return the last value written to the POST code I/O port, if that port is readable. In an ISA architecture, I/O port 80h is write-only and cannot be written. By changing this value to 2ffh or 3ffh (scratch registers on 16550 UARTs), this value can be inspected by this API function.

Request Format:

BOOLEAN MsgGetLastPostCode(
IN UCHAR ***Buffer***
);

Parameters:

Buffer - A 16:16 (far) pointer to an 8-bit storage location where the 8-bit POST code will be returned.

14.3.8 MsgChecksum Function

The *MsgChecksum* function is called by the application to execute a checksum function on the target. This function computes a 32-bit value associated with target memory starting at a specified physical address, for a specified number of bytes from 0 to 65,535.

Request Format:

BOOLEAN MsgChecksum(
IN ULONG *RegionStartAddress***,**
IN USHORT *RegionLength***,**
OUT ULONG ***Checksum***
);

Parameters:

RegionStartAddress - A 32-bit physical address on the target (not the machine making the *MsgChecksum* function call) of the first byte in the region to be checksummed.

RegionLength - A 16-bit unsigned value specifying the size of the contiguous region in bytes.

Checksum - A 16:16 (far) pointer to an unsigned long variable in the host program's address space where the system will return the checksum value.

14.3.9 MsgTestMemory Function

The *MsgTestMemory* function is called by the application to perform an exhaustive memory test on a specified block of target memory. This function causes the target to perform the test and return whether the memory was operational or not.

Request Format:

BOOLEAN MsgTestMemory(
  IN USHORT *TestType***,**
  IN ULONG *StartingPhysicalAddress***,**
  IN ULONG *BlockSize*
  );

Parameters:

TestType - A 16-bit word containing bits that, when set, indicate that a specific type of test must be performed in order for the test not to fail. If multiple bits are specified, then the test will only pass if all of the specific types of tests associated with each of those bits pass.

0x0001 - A bit is rotated through all successive bit positions within each word in the block to verify that each bit can toggle on and off.

StartingPhysicalAddress - A 32-bit physical address on the target (not the machine making the *MsgTestMemory* function call) of the first byte in the region to be tested.

BlockSize - A 32-bit unsigned long value specifying the number of contiguous bytes to be tested.

14.3.10 MsgReadFlash Function

The *MsgReadFlash* function is called by the application to read Flash memory starting at a specified physical address on the target. The data at the address are transferred to a staging buffer, which is then available for reading via the *MsgReadBuffer* function.

Request Format:

BOOLEAN MsgReadFlash(
  IN ULONG *DevicePhysicalAddress***,**
  IN ULONG *RelativePhysicalAddress***,**
  IN USHORT *Length*
  );

Parameters:

DevicePhysicalAddress - A 32-bit physical address in the target's address space that specifies the address of the physical Flash device. The relative address is then used as an offset with respect to this device address.

RelativePhysicalAddress - A 32-bit offset relative to the *DevicePhysicalAddress* of the first byte in the contiguous region to be transferred to the target's RAM staging buffer.

Length - A 16-bit value specifying the number of bytes to be transferred. The value 0 indicates 65,536 bytes should be transferred.

14.3.11 MsgWriteFlash Function

The *MsgWriteFlash* function is called by the application to write Flash memory starting at a specified physical address on the target. The data from the target's staging buffer are transferred to the Flash.

Request Format:

BOOLEAN MsgWriteFlash(
IN ULONG *DevicePhysicalAddress***,**
IN ULONG *RelativePhysicalAddress***,**
IN USHORT *Length*
);

Parameters:

DevicePhysicalAddress - A 32-bit physical address in the target's address space that specifies the address of the physical Flash device. The relative address is then used as an offset with respect to this device address.

RelativePhysicalAddress - A 32-bit offset relative to the *DevicePhysicalAddress* of the first byte in the contiguous region to be written from the target's RAM staging buffer.

Length - A 16-bit value specifying the number of bytes to be transferred. The value 0 indicates 65,536 bytes should be transferred.

14.3.12 MsgReadBuffer Function

The *MsgReadBuffer* function is called by the application to read data from the target's RAM staging buffer over the Manufacturing Link to a local host buffer.

This function is typically used in combination with other functions. For example, the contents of the Flash array on the target could be copied to the staging buffer with the *MsgReadFlash* function, and then *MsgReadBuffer* could be used to transfer that data to the host.

Updating sections of Flash is possible by using *MsgReadFlash* to read-in an entire block, transferring the data to the host with *MsgReadBuffer*, then transferring the changed portion back to the target with *MsgWriteBuffer*, followed with a Flash update using *MsgEraseFlash* followed by *MsgWriteFlash*.

Request Format:

BOOLEAN MsgReadBuffer(
IN USHORT *BufferOffset***,**
IN USHORT *BytesToRead***,**
OUT VOID *****LocalBuffer*
);

Parameters:

BufferOffset - A 16-bit unsigned value specifying the offset, relative to the start of the target's RAM staging buffer, of the block of data to be transferred to the host. The actual physical or segment address of the staging buffer is not exposed to the target.

BytesToRead - A 16-bit unsigned value specifying the number of bytes to transfer. If 0 is specified, then 65,536 bytes will be transferred.

LocalBuffer - A 16:16 (far) pointer to a host application buffer where the data will be copied to. This buffer must be large enough to accommodate *BytesToRead* bytes of incoming data.

14.3.13 MsgWriteBuffer Function

The *MsgWriteBuffer* function is called by the application to write data from a local host buffer to the target's RAM staging buffer over the Manufacturing Link.

This function is typically used in combination with other functions. For example, the contents of the Flash array on the target could be written from the host by using the *MsgWriteBuffer* to transfer the data from the host to the target, and then the *MsgWriteFlash* function could be used to write the contents of the staging buffer to the Flash.

Updating sections of Flash is possible by using *MsgReadFlash* to read-in an entire block, transferring the data to the host with *MsgReadBuffer*, then transferring the changed portion back to the target with *MsgWriteBuffer*, followed with a Flash update using *MsgEraseFlash* followed by *MsgWriteFlash*.

Request Format:

BOOLEAN MsgWriteBuffer(
IN USHORT *BufferOffset***,**
IN USHORT *BytesToWrite***,**
OUT VOID *****LocalBuffer*
);

Parameters:

BufferOffset - A 16-bit unsigned value specifying the offset, relative to the start of the target's RAM staging buffer, where the host data will be transfered to. The actual physical or segment address of the staging buffer is not exposed to the target.

BytesToWrite - A 16-bit unsigned value specifying the number of bytes to transfer. If 0 is specified, then 65,536 bytes will be transferred.

LocalBuffer - A 16:16 (far) pointer to a host application buffer containing the data to transfer to the target.

14.3.14 MsgLockFlash Function

The *MsgLockFlash* function is called by the application to write-protect a block in a Flash memory device. The physical address of the device itself is specified, as well as the relative offset of the block within the device. Once the block is locked, it cannot be written to.

Request Format:

BOOLEAN MsgLockFlash(
IN ULONG *DevicePhysicalAddress*,
IN ULONG *RelativePhysicalAddress*
);

Parameters:

DevicePhysicalAddress - A 32-bit physical address in the target's address space that specifies the address of the physical Flash device. The relative address is then used as an offset with respect to this device address.

RelativePhysicalAddress - A 32-bit offset relative to the *DevicePhysicalAddress* of the first byte in the block to be locked.

14.3.15 MsgEraseFlash Function

The *MsgEraseFlash* function is called by the application to unlock and erase a block in a Flash memory device. The physical address of the device itself is specified, as well as the relative offset of the block within the device. Once the block is erased, it becomes unlocked, and every byte within the block contains the value ffh (all bits set to 1).

Request Format:

BOOLEAN MsgEraseFlash(
IN ULONG *DevicePhysicalAddress*,
IN ULONG *RelativePhysicalAddress*
);

Parameters:

DevicePhysicalAddress - A 32-bit physical address in the target's address space that specifies the address of the physical Flash device. The relative address is then used as an offset with respect to this device address.

RelativePhysicalAddress - A 32-bit offset relative to the *DevicePhysicalAddress* of the first byte in the block to be erased.

14.3.16 MsgInt13 Function

The *MsgInt13* function is called by the application to issue an INT 13h disk BIOS function on the target. This allows the implementation of a variety of remote-access disk utilities, such as formatters, file system checkers, and remote disks, that can all operate in the context of Manufacturing Mode.

This function passes the AX, CX, and DX registers to the target for execution in its environment; however, the ES and BX registers, which are normally used to specify a buffer address for read, write, format, and verify functions, are not passed in the request. Instead, the RAM staging buffer is assumed to be the buffer to be used in all operations. The target automatically sets ES and BX to the segment and offset components of the address of the RAM staging buffer for each I/O.

The automatic use of the RAM staging buffer makes it possible to use the *MsgReadBuffer* and *MsgWriteBuffer* to read and write this buffer before or after using the *MsgInt13* function. Thus, a typical disk read involves a call to *MsgInt13*, followed by a call to *MsgReadBuffer*. Similarly, a write to disk involves writing the data to the target's staging buffer with a call to *MsgWriteBuffer*, followed by a call to *MsgInt13* to perform the disk write itself.

This use of the RAM stagin buffer has advantages when it is desired to initialize many sectors with the same contents; for example, when formatting a disk or creating a file system, because it may eliminate the need to continuously transfer the same data repeatedly over the RS232 link.

This routine returns FALSE if the CY flag was set (indicating an error) on the target. It can also return FALSE if the function is not able to be executed because of other errors, such as protocol problems. The routine returns TRUE if no protocol problems were encountered, and the INT 13h request was executed successfully on the target, and the resulting CY flag status was clear, indicating no error resulted from the INT 13h operation.

Request Format:

BOOLEAN MsgInt13(
IN USHORT *InAx*,
IN USHORT *InCx*,
IN USHORT *InDx*,
OUT USHORT **OutAx*,
OUT USHORT **OutCx*,
OUT USHORT **OutDx*,
);

Parameters:

InAx - A 16-bit value that will be placed in the AX CPU register before executing the INT 13h instruction on the target. Consult Chapter 23 section 4 for details on the INT 13h BIOS interface.

InCx - A 16-bit value that will be placed in the CX CPU register before executing the INT 13h instruction on the target. Consult Chapter 23 section 4 for details on the INT 13h BIOS interface.

InDx - A 16-bit value that will be placed in the DX CPU register before executing the INT 13h instruction on the target. Consult Chapter 23 section 4 for details on the INT 13h BIOS interface.

OutAx - A pointer to a 16-bit storage location where the contents of the AX CPU register will be stored after executing the INT 13h instruction on the target.

OutCx - A pointer to a 16-bit storage location where the contents of the CX CPU register will be stored after executing the INT 13h instruction on the target.

OutDx - A pointer to a 16-bit storage location where the contents of the DX CPU register will be stored after executing the INT 13h instruction on the target.

Chapter 15

ADVANCED POWER MANAGEMENT

EMBEDDED BIOS supports the salient features of the Advanced Power Management (APM) API jointly defined by Intel and Microsoft. APM is a specification that defines a layered cooperative environment which allows applications, operating systems, and the system BIOS to work together to reduce power consumption in Personal Computers. EMBEDDED BIOS extends this goal to embedded systems. The primary purpose of an APM implementation is to provide portable computer users increased productivity and greater system availability by extending the useful life of system batteries without degrading system performance.

The EMBEDDED BIOS Power Management Subsystem (PMS) controls power to BIOS-controlled devices specified by the OEM in the power management device tree. This tree of devices is described by the OEM using the **POWER_DEVID** macro in the project file (see Chapter 7 for details). Each participating device in the power management device tree receives notifications from PMS when the system's power is changing state, in an orderly manner.

15.1 APM System Model

APM defines four power states of a system: Ready, Standby, Suspended, and Off. Three of these states apply both to individual system components and to the system as a whole. The suspended state is a special low power condition that applies to the system as a whole, and not the individual components.

In the **Ready** state, the system or device is fully powered up and ready for use. The APM definition of Ready only indicates that the system or device is fully powered on; it does not differentiate between active and idle conditions of the operating system or application software.

Standby is an intermediate system dependent state which attempts to conserve power. Standby is entered when the CPU is idle an d no device activity is known to have occurred within a machine-defined period of time. The machine will not return to ready until one of the following events occur: (1) A device causes a hardware interrupt to be generated, or (2) Any controlled device is accessed. All data and operational parameters are preserved when the machine is in the Standby state.

The **Suspended** state is a system state that is defined to be the lowest level of power consumption available that preserves operational data and parameters. The Suspended state can be initiated by either the system BIOS or the software above the BIOS. The system BIOS may place the system into the suspended state without notification if it detects a situation which requires an immediate response such as the battery entering a critically low power state. When the system is in the Suspended state, computation will not be performed until normal activity is resumed. Resumption of activity will not occur until signalled by an external event such as a button press, timer alarm, etc.

When in the **Off** state, the system or device is powered down and is inactive. Data and operational parameters may or may not be preserved in the Off state.

The system and devices can change from one power state to another either by explicit command or automatically, based on APM parameters and system activity. The power capabilities of devices differ, and some devices may not be capable of achieving all states. Additionally, some devices may have built-in automatic power management functions that are invisible to the system, and therefore lie outside of the scope of the APM model.

15.2 APM Software Layers

The system BIOS is the lowest level of power management software in the system. EMBEDDED BIOS provides APM services that ultimately call hardware support functions in the CHIPSET and CPU Personality Modules.

The system BIOS is capable of providing power management functionality without any support from the operating system or applications. This support is analogous to the ad hoc power management methods implemented on some laptop systems. This functionality can be enhanced once an APM-aware operating system or environment establishes a cooperative connection with the BIOS. Once made, this connection establishes a protocol that allows the firmware to communicate power management events to the oeprating system and to wait for operating system concurrence if necessary. Details of the system BIOS operational changes are documented in Chapter 20, in the section on power management.

The operating system layer has three primary power management functions:

1. Pass calls and information between application and system BIOS layers; and

2. Arbitrate application power management calls in a multitasking environment; and

3. Identify power-saving opportunities not apparent at the application level.

Different operating systems will require different power management application-to-OS interfaces. On some systems this interface may best be implemented through software interrupts, while on other systems, a CALL and RET interface may be more appropriate.

Operating systems that can be enhanced by optional extensions may require specialized interfaces to allow the extensions to assist in the power management function.

The application layer assists the power management function by providing information that only the application is in a position to know or easily ascertain. Applications are not required to

support the power management function, but they can greatly increase its effectiveness, particularly on less-sophisticated operating systems. Under MS-DOS in particular, the application is often in the best position to know when it is idle and awaiting user input.

15.3 APM BIOS Interface

The system BIOS interface as defined in the APM specification can support up to three CPU modes (real mode, 16-bit protected mode, and 32-bit protected mode), although only real-mode support is required of the BIOS by APM.

The real-mode interface is required on all APM implementations and is the primary APM interface. This interface is implemented with an extension to the existing PC/AT INT 15h BIOS service. The system BIOS INT 15h interface must operate in either real mode, or virtual-86 mode on 80386 and above procesors. Since the INT 15h processor instruction normally disables interrupts, the system BIOS can typically expect to be entered with interrupts disabled. However, the BIOS routines should not depend on any particular setting, and shoud explicitly enable and disable interrupts as necessary. To avoid re-entering INT 15h, and ISR should not try to use any of the APM functions.

To better support the protected modes of 80286 and later processors, the system BIOS may optionally support a 16-bit and a 32-bit protected mode interface, directly callable from protected mode. The protected mode interfaces must first be initialized using the real mode INT 15h function 53h. Systems not supporting APM will return from this call with CY set and the AH CPU register set to 86h. EMBEDDED BIOS supports the APM INT 15h function 53h when **OPTION_SUPPORT_APM** is enabled.

The 16-bit and 32-bit protected mode interfaces must be initialized using the Protected Mode 16-bit Interface Connect and Protected Mode 32-bit Interface Connect functions before these interfaces can be used.

If an error condition is detected during the processing of a system BIOS APM function, upon return to the caller the CY flag will be set and the AH CPU register will contain an APM error code. The carry flag will be clear upon return from any successful APM call, and the contents of the AH CPU register will be dependent on the particular call.

To allow for direct control of devices, this interface adopts a convention for identifying a device class and specific subclasses within that class. For example, a class would be all Disk devices and the subclasses would be the physical unit numbers. APM calls take this parameter, in a word length register where the most significant byte is the device class and the least significant byte is the device subclass. The following classes and subclasses are defined by the APM interface:

0000h - System class, BIOS subclass.
0001h - System class, BIOS-supported devices.
01xxh - Video display devices.
02xxh - Secondary storage devices.
03xxh - Parallel ports devices.
04xxh - Serial ports devices.

For a complete list of the APM functions provided by EMBEDDED BIOS under this model, consult Chapter 21 under the INT 15h "General Services" section.

15.4 Power Management Subsystem (PMS)

The EMBEDDED BIOS Power Management system provides an hierarchical approach to the system's power control. Rather than only handle a system-wide power state, it implements an APM state machine for each device, and then responds to APM requests by transitioning each device participating in the system's power management in the proper order.

This ordering mechanism is critical for proper power management in embedded systems. Consider that a typical rotating magnetic IDE hard drive has a motor that can spin down, drive electronics that can go low-power, a Super I/O chip on the motherboard that can be powered down, and finally a CPU that can go to sleep. All of these components must be powered up in the proper order, and powered down in the proper order, in order to maintain data integrity and correct operation of the system.

15.4.1 POWER_DEVID Device Tree

As described in Chapter 7, the **POWER_DEVID** macro is used to define a tree of device dependencies for the power manager. Always at the top of the tree is the CPU itself. The CPU is the parent of all 1st-tier devices underneath it, such as Super I/O controllers, Flash arrays, PCMCIA controllers, and the like. Similarly, 1st-tier devices become parents of the devices they control, such as IDE drives and UARTs in the case of Super I/O controllers, PCMCIA cards in the case of PCMCIA controllers, and so on. EMBEDDED BIOS has a limit of eight (8) levels in its power management tree, which is more than adequate for anticipated designs.

The power management device tree is specified in a tabular format with **POWER_DEVID** entries. Each line in the table specifies a new device that will be participating in the system's power management, and begins with the identifying macro command, **POWER_DEVID**. Each line contains exactly four (4) operands, as in the following *hypothetical example*:

```
;       Power management device tree definition:
;       The POWER_DEVID entry for the CPU MUST be FIRST!
;
;               Device  Module: Parent: Setup text:
;
  POWER_DEVID   CPU,    Board,  CPU,    "Cpu"
  POWER_DEVID   IDE_0,  Ide,    CPU,    "IDE drive 0"
  POWER_DEVID   IDE_1,  Ide,    CPU,    "IDE drive 1"
  POWER_DEVID   SUPERIO,Board,  CPU,    "Super I/O"
  POWER_DEVID   PCMCIA, Board,  CPU,    "PCMCIA"
```

The first operand specifies the symbolic name of the participating device. These device names must have legal MASM or TASM symbol syntax, and should really be short names to keep the table simple. These symbols are case-sensitive, and are referred-to by the parent field in other entries of the table.

The second operand specifies the software component, usually a module name, that is responsible for management of the device. This operand is prepended to the string `PwrLvl` to produce a final name of a procedure in the BIOS that is responsible for managing the device's power level. This routine is called by the core BIOS's power management system at the appropriate time to instruct the module to change the device's power state. Because this operand specifies a name, and not an ordinal, it is possible to add OEM-defined device types to the system. General Software has provided the following types in the core BIOS:

`Board`	OEM Board Personality Module
`Ide`	IDE Hard Drives
`Media`	Media Control Layer (All RFD Devices)
`MtdRam`	RAM MTD
`MtdRom`	ROM MTD
`MtdAmd8_1`	AMD Flash 8-Bit 1-Way MTD
`MtdAmd8_2`	AMD Flash 8-Bit 2-Way MTD
`MtdAmd8_4`	AMD Flash 8-Bit 4-Way MTD
`MtdAmd16_1`	AMD Flash 16-Bit 1-Way MTD
`MtdAtm8_1`	Atmel Flash 8-Bit 1-Way MTD
`MtdBulk_1`	Bulk Erase Flash 8-Bit 1-Way MTD
`MtdInt16_1`	Intel Flash 16-Bit 1-Way MTD
`MtdInt16_2`	Intel Flash 16-Bit 2-Way MTD
`MtdInt8_1`	Intel Flash 8-Bit 1-Way MTD
`MtdInt8_2`	Intel Flash 8-Bit 2-Way MTD
`MtdInt8_A`	Intel Flash 28F016 in 8-Bit mode (1-Way) MTD
`MtdInt8_B`	Intel Boot Block Flash MTD
`MtdInt16A_1`	Intel 28F016/28F032 MTD
`MtdToshNand`	Toshiba/AMD NAND MTD

Note that the above list includes some “real” devices, and some “pseudo” devices. For example, the Flash parts managed by the MtdInt16_1 Media Technology Driver are very real. The IDE drives managed by the Ide module are also real. The Board module corresponds to real hardware if the OEM chooses to write the **BoardPwrLvl** routine to handle power management requests, and that routine must do whatever makes sense to manage the board’s “power”. Pseudo devices such as the one called Media are actually place-holders. It is necessary to allow these intermediate devices (this one routes Flash requests to the underlying MTDs), to play a role in managing power, so that they receive notification that they should suspend or resume the processing of their client’s requests if the power is suspended or resumed, respectively.

In later versions of EMBEDDED BIOS, additional power management devices may be provided. For information about the current list of supported power management devices, contact General Software.

The third operand specifies the device name of the device’s parent. For IDE drives, this is typically a Super I/O controller (the OEM would need to define the appropriate **SuperIoPwrLvl** routine in the Board Personality Module that is responsible for managing the Super I/O controller in the system). For UARTs, this might also be a Super I/O controller.

The fourth operand specifies an ASCII string in quotes that will appear in the power management SETUP screens so that the user can configure timeouts and enable and disable power management on a device basis. These strings should be kept simple and short, so as to fit within the space constraints of the SETUP screen system and also to be clear to the user.

15.4.2 Device Power Control Entrypoints

The ***Xxx*PwrLevel** function (where *Xxx* is the name of the device to participate in the power management device tree) is called with near procedure linkage by the EMBEDDED BIOS Power Management System to coordinate the device’s power with the rest of the system’s state.

This routine handles three situations; the power level may go up, down, or stay the same. If the power level stays the same, no action is required.

When the power level goes down, the routine must prepare the device by waiting for outstanding activities to complete, and then drop the power level to the device to the appropriate level. The routine may need to save the device's registers or other state as necessary in RAM or on the device itself. This state should be capable of being fully restored by the routine when power is brought back to the former level.

When the power level goes up, the routine must enable the device's power and reinitialize it as necessary to accept further requests that are appropriate for that level of operation.

One of the functions of this routine is to enable or disable the handling of the device's request handler. For example, for IDE drives that require the spindle to be rotating, the device's INT 13h request handler should test the saved status of the device to determine if the drive has been powered-down before allowing the request to flow through the driver, generating needless timeout conditions.

Once this routine returns to its caller (PMS), the state transition must have been completed. This routine is never reentered by PMS; that is, the calls to this routine are synchronized within PMS.

Input Parameters:

DS - Points to the extended BIOS data area (EBDA).
BX - Device index.
CL - New power level.
CH - Old power level.

Output Parameters:

None.

Unpreserved Registers:

Flags.

Chapter 16

SETUP AND DIAGNOSTICS SYSTEM

EMBEDDED BIOS provides a comprehensive SETUP screen system that includes built-in system diagnostics for system burn-in. This chapter describes this integrated subsystem, which is highly-configurable by the OEM.

16.1 SETUP Build Options

To enable the basic SETUP screen in the core BIOS, make sure **OPTION_SUPPORT_SETUP** is enabled in the project file. Then, enable main menu options as desired from the list below (for details about the options, see Chapter 7).

In addition to these options, the OEM can further configure the SETUP system by enabling or disabling **OPTION_SOFTERR_SETUP**, which specifies whether soft errors occuring during POST cause the SETUP screen system to be activated; and **OPTION_QUERY_ENTERSETUP**, which causes POST to unconditionally ask the user if SETUP should be entered before booting the operating system.

Finally, the default values used for the fields in the BASIC CMOS SETUP and SHADOWING SETUP screens are specified with the many parameters that take the form, **OPTION_CMOS_*x***, where *x* is the name of the field. For example, **OPTION_CMOS_MOUSE** is the option which when set, enables PS/2 mouse initialization during POST, and **OPTION_CMOS_SHADOW_DC00** is the option which when set, enables shadowing for the 16KB region at segment DC00h.

Because targets have widely varying hardware and SETUP requirements, not all SETUP screens are configured for every system. For example, if no Resident Flash Disk is used in a design, the FORMAT FLASH DISK menu is not supported. Similarly, if the integrated BIOS debugger is omitted from an adaptation, then the START SYSTEM BIOS DEBUGGER option is not supported. Consult Chapter 7 for details about these configuration options.

SETUP console I/O, including both keyboard and screen, can be redirected over an RS-232 link at the OEM's option. This is configured first by enabling **OPTION_SUPPORT_CON_REDIRECTOR**, and then setting **CONFIG_CONIO_SETUP** to the COM port number (1, 2, 3, 4) over which SETUP I/O will be routed. The special value 0

means that redirection occurs over the standard keyboard and screen, while still permitting the console redirector code to be enabled in the system for on-the-fly console switching. . When run over a serial port, SETUP generates ANSI escape sequences that can drive a terminal program to emulate a PC's screen.

16.2 Entering SETUP

The SETUP screen system is invoked after POST has completed, in response to one of the following:

- During POST's memory countup display, the console user can press the **<DEL>** key when the console is the PC keyboard and video monitor, or the **^C** key when POST I/O has been redirected over an RS-232 link. This causes the remainder of the memory test to be quickly performed, and then the SETUP screen takes over.

- If a soft error occurs during POST, then if **OPTION_SOFTERR_SETUP** is enabled, POST will enter SETUP automatically. Most soft errors occur as a result of losing CMOS information (when a CMOS part is present). This is usually the result of a dead battery. Examples of soft errors are memory size mismatch, keyboard failure, or disk drive failure.

- If POST attempts all six boot actions and all six fail to find a bootable system, then if no INT 18h handler is installed, POST will display a small menu that offers the user a way to enter SETUP, or other functions such as the debugger or Manufacturing Mode.

SETUP cannot be entered during normal system operation with a special key combination, because SETUP requires low memory to use for scratch space.

16.3 SETUP Screens

A fully-configured SETUP main menu looks like the following:

```
                 System Bios Setup - Utility v4.2
       (C) 1999 General Software, Inc. All rights reserved

                      >About Embedded BIOS
                    Basic CMOS Configuration
                      Custom Configuration
                      Shadow Configuration
                  Standard Diagnostic Routines
                   Start System BIOS Debugger
                 Start RS232 Manufacturing Link
                 Reset CMOS to last known values
                 Reset CMOS to factory defaults
                     Write to CMOS and Exit
                   Exit without changing CMOS

                  <Esc> to continue (no save)
```

Fig 1. The Embedded BIOS SETUP Menu Gives Access to the Embedded BIOS Pre-Boot Features.

Getting around in the SETUP screens is normally accomplished by using the arrow keys, pressing Enter, or even using the TAB or Shift-TAB keys. However, when the SETUP system is redirected over a serial port, control keys are used. Use ^E to go up or backwards, ^X to go down or forwards. TAB also goes forwards. On serial links, ^C is used to enter SETUP.

16.3.1 Basic CMOS Configuration Screen

The system's drive types, boot activities, and POST optimizations are configured from the Basic Setup Screen (Fig 2). In order to use disk drives with your system, you must select appropriate assignments of drive types in the left-hand column. Then, if you are using true floppy and IDE drives (not memory disks that emulate these drives), you need to configure the drive types themselves in the Floppy Drive Types and IDE Drive Geometry sections. Finally, you'll need to configure the boot sequence in the middle of the screen. Once these selections have been made, your system is ready to use.

```
                 System Bios Setup - Basic CMOS Configuration
             (C) 2000 General Software, Inc. All rights reserved

DRIVE ASSIGNMENT ORDER:       Date: Jan 01, 2000   Typematic Delay   : 250 ms
Drive A: Floppy 0             Time: 00 : 42 : 34   Typematic Rate    : 30 cps
Drive B: <None>               NumLock: >Disabled   Seek at Boot      : Floppy
Drive C: Ide 0/Pri Master                          Show "Hit Del"    : Enabled
Drive D: CD Hd/Pri Slave      BOOT ORDER:          Config Box        : Enabled
Drive E: <None>               Boot 1st: Drive A:   F1 Error Wait     : Enabled
Drive F: <None>               Boot 2nd: Drive C:   Parity Checking   : <Unused>
Drive G: <None>               Boot 3rd: <None>     Memory Test Tick  : Enabled
Drive H: <None>               Boot 4th: <None>     Test Above 1 MB   : Enabled
Drive I: <None>               Boot 5th: <None>     Debug Breakpoints: Disabled
Drive J: <None>               Boot 6th: <None>     Splash Screen     : Enabled
Drive K: <None>
Boot Method: Windows CE       IDE DRIVE GEOMETRY:  Sect  Hds  Cyls   Memory
                              Ide 0: 3 = AUTOCONFIG, LBA               Base:
FLOPPY DRIVE TYPES:           Ide 1: 3 = AUTOCONFIG, LBA                640KB
Floppy 0: 1.44 MB, 3.5"       Ide 2: 3 = AUTOCONFIG, LBA               Ext:
Floppy 1: 1.44 MB, 3.5"       Ide 3: 3 = AUTOCONFIG, LBA                28MB

        ↑/↓/←/→/<CR>/<Tab> to select  or  <PgUp>/<PgDn>/+/- to modify
                      <Esc> to return to main menu
```

Fig 2. The Embedded BIOS Basic Setup Screen is used to configure drives, boot actions, and POST.

16.3.1.1 Configuring Drive Assignments

Embedded BIOS allows the user to map a different file system to each drive letter. The BIOS allows file systems for each floppy (Floppy0 and Floppy1), each IDE drive (Ide0, Ide1, Ide2, and Ide3), and memory disks when configured (Flash0, ROM0, RAM0, etc.) Fig. 2 shows how the first floppy drive (Floppy0) is assigned to drive A: in the system, and then how the first IDE drive (Ide0) is assigned to drive C: in the system.

To switch two floppy disks around or two hard disks around, just map Floppy0 to B: and Floppy1 to A:, and for hard disks map Ide0 to D: and Ide1 to C:.

> Caution: Take care to not skip drive A: when making floppy disk assignments, as well as drive C: when making hard disk assignments. The first floppy should be A:, and the first hard drive should be C:. Also, do not assign the same file system to more than one drive letter. Thus, Floppy0 should not be used for both A: and

B:. The BIOS permits this to allow embedded devices to alias drives, but desktop operating systems may not be able to maintain cache coherency with such a mapping in place.

A special field in this section entitled "Boot Method: (Windows CE/Boot Sector)" is used to configure the CE Ready™ feature of the BIOS. For normal booting (DOS, Windows NT, etc.), select "Boot Sector" or "Unused".

16.3.1.2 Configuring Floppy Drive Types

If true floppy drive file systems (and not their emulators, such as ROM, RAM, or Flash disks) are mapped to drive letters, then the floppy drives themselves must be configured in this section. Floppy0 refers to the first floppy disk drive on the drive ribbon cable (normally drive A:), and Floppy1 refers to the second drive (drive B:).

16.3.1.3 Configuring IDE Drive Types

If true IDE disk file systems (and not their emulators, such as ROM, RAM, or Flash disks) are mapped to drive letters, then the IDE drives themselves must be configured in this section. The following table shows the drive assignments for Ide0-Ide3:

File System Name	Controller	Master/Slave
Ide0	Primary (1f0h)	Master
Ide1	Primary (1f0h)	Slave
Ide2	Secondary (170h)	Master
Ide3	Secondary (170h)	Slave

To use the primary master IDE drive in your system (the typical case), just configure Ide0 in this section, and map Ide0 to drive C: in the Configuring Drive Assignments section.

The IDE Drive Types section lets you select the type for each of the four IDE drives: None, User, Physical, LBA, or CHS.

The **User** type allows the user to select the maximum cylinders, heads, and sectors per track associated with the IDE drive. This method is rarely used since LBA is now in common use.

The **Physical** type instructs the BIOS to query the drive's geometry from the controller on each POST. No translation on the drive's geometry is performed, so this type is limited to drives of 512MB or less. Commonly, this is used with embedded ATA PC Cards.

The **LBA** type instructs the BIOS to query the drive's geometry from the controller on each POST, but then translate the geometry according to the industry-standard LBA convention. This supports up to 16GB drives. ***Use this method for all new drives.***

The **CHS** type instructs the BIOS to query the drive's geometry from the controller on each POST, but then translate the geometry according to the Phoenix CHS convention. Using this type on a drive previously formatted with LBA or Physical geometry might show data as being missing or corrupted.

16.3.1.4 Configuring Boot Actions

Embedded BIOS supports up to six different user-defined steps in the boot sequence. When the entire system has been initialized, POST executes these steps in order until an operating system successfully loads. In addition, other pre-boot features can be run before, after, or between operating system load attempts. The following actions can be used:

Drive A: - K: Boot operating system from specified drive. If “Loader” is set to “BootRecord” or “Unused”, then the standard boot record will be invoked, causing DOS, Windows95/98, Windows NT, or other industry-standard operating systems to load. If “Boot Method” is set to “Windows CE”, then the boot drive’s boot record will not be used, and instead the BIOS will attempt to load and execute the Windows CE Kernel file, NK.BIN, from the root directory of each boot device.

Debugger Launch the Integrated BIOS Debugger. To return exit the debugger environment, type “G” at the debugger prompt and press ENTER.

MFGMODE Initiate Manufacturing Mode, allowing the system to be configured remotely via an RS232 connect to a host computer.

WindowsCE Execute a ROM-resident copy of Windows CE, if available. This feature is not applicable unless configured by the OEM in the BIOS adaptation.

DOS in ROM Execute a ROM-resident copy of DOS, if available. This feature is not applicable unless an XIP copy of DOS, such as Embedded DOS-ROM, has been stored in the BIOS boot ROM. Copies of Embedded DOS-ROM may be obtained from General Software.

None No action; POST proceeds to the next activity in the sequence.

16.3.2 Custom Configuration Setup Screen

The system’s hardware-specific features are configured with the Custom Setup Screen (Fig 4). All features are straightforward except for the Redirect Debugger I/O option, which is an extra embedded feature that allows the user to select whether the Integrated BIOS Debugger should use standard keyboard and video or RS232 console redirection for interaction with the user. If no video is available, the debugger is always redirected.

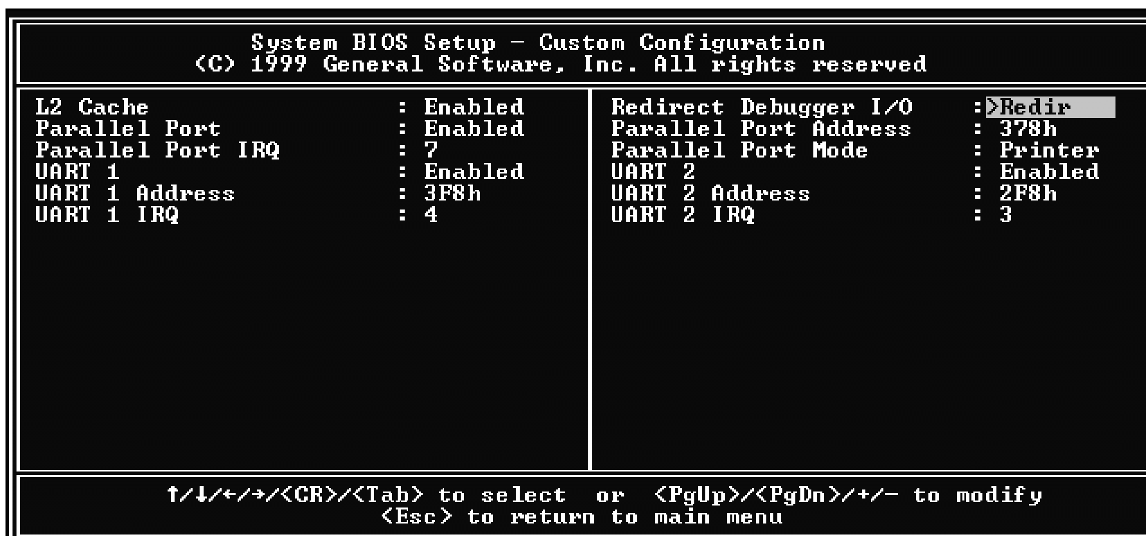


Fig 3. The Embedded BIOS Custom Setup Screen is used to configure configure low-level hardware.

16.3.3 Shadow Configuration Setup Screen

The system's Shadow Configuration Setup Screen (Fig 4) allows enabling and disabling of shadowing in 16KB sections, except for the top 64KB of the BIOS ROM, which is shadowed as a unit. Normally, shadowing should be enabled at C000/C400 to enhance VGA ROM BIOS performance, and E000-F000 should be shadowed to maximize system ROM BIOS performance.

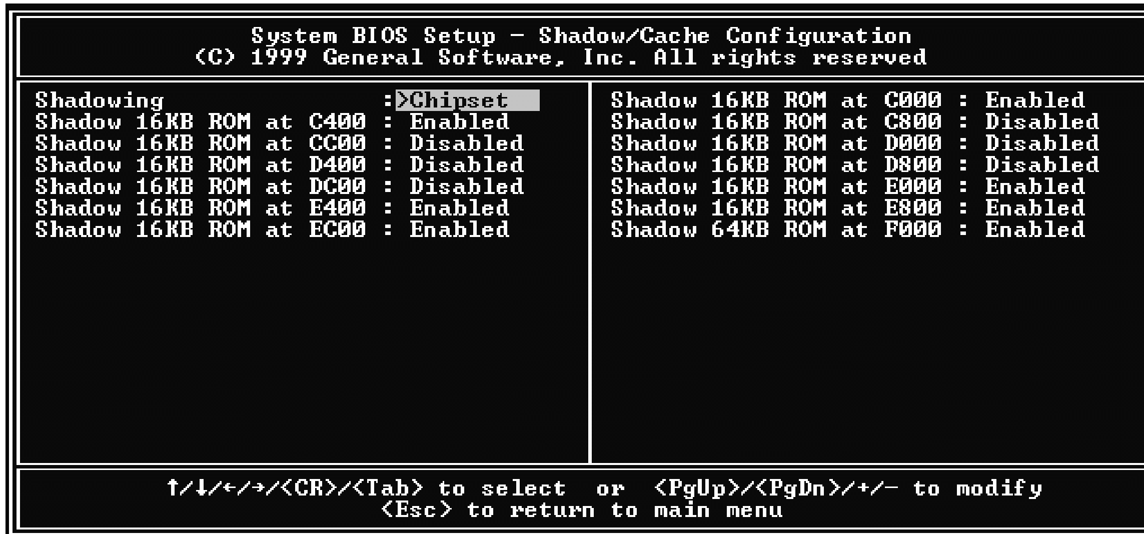


Fig 4. The Embedded BIOS Shadow Setup Screen is used to configure ROM Shadowing.

16.3.4 Standard Diagnostic Routines Setup Screen

Embedded systems may require automated burn-in testing in the development cycle. This facility is provided directly in the system BIOS through the Standard Diagnostics Routines Setup Screen (Fig 5). To use the system, selectively enable or disable features to be tested, and then enable the "Tests Begin on ESC?" option to cause the system test suite to be invoked. To repeat the system test battery continuously, you should also enable the "Continuous Testing" option. When continuous testing is started, the system will continue until an error is encountered.

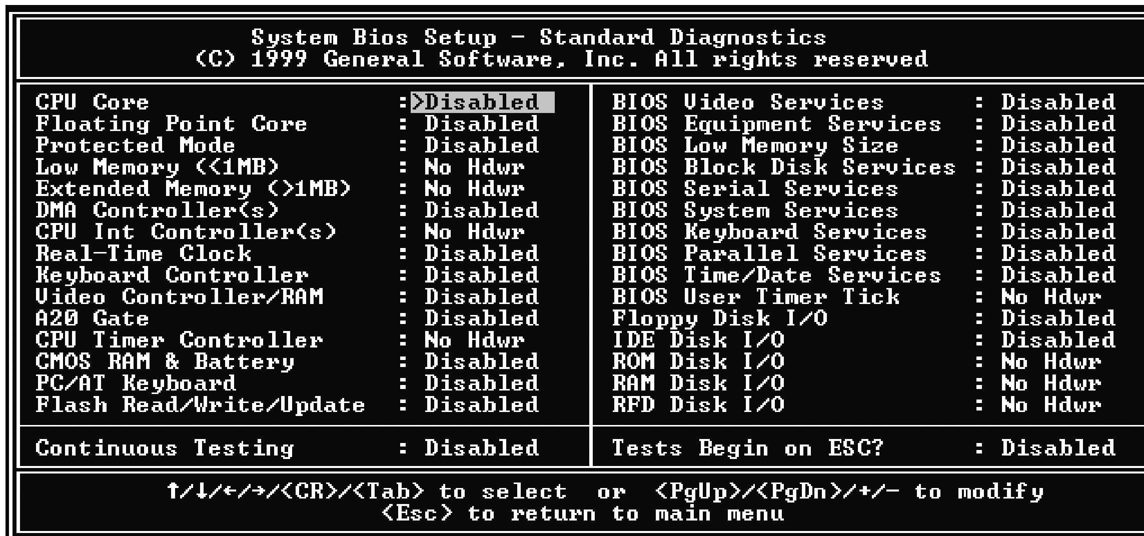


Fig 5. The Standard Diagnostic Routines Setup Screen Provides Burn-In Tests for Manufacturing.

Caution: The disk I/O diagnostics perform write operations on those drives; therefore, only spare drives should be used which do not contain data that could be harmed by the test.

Advisory: The keyboard test may fail when in fact the hardware is operating within reasonable limits. This is because although the device may produce occasional errors, the BIOS retries operations when failures occur during normal operation of the system.

Diagnostics can be very helpful when first bringing-up new hardware, and they can also be quite useful in the field when targets appear to have intermittant operation.

In the lab, you'll want to enable the standard diagnostics suite in the SETUP system once you have a keyboard and a screen working in your BIOS adaptation. This lets you test out all of the basic mechanisms that the rest of the BIOS will use during steady state operation of the system. For example, if you're unsure whether the A20 gate circuitry is working or if you've used the correct BIOS option to support your A20 gate hardware, run the A20 gate test continuously for a few hours, and let it find problems by itself.

Diagnostics can also be used while the system is in a controlled (excessive) environment to determine MTBF or maximum rating values. For example, to determine how the floppy disk and IDE disk respond in a high-temperature environment, place the unit in a temperature-controlled chamber and run the floppy disk and IDE disk diagnostics continuously until they fail. Then, you'll have the exact maximum ratings documented.

In the field, it is easy to spot failing memory and mechanical devices such as rotating disks by running diagnostics. This need not occur on the actual user's display of the target; it can operate over a serial link that is normally not used by the embedded application.

There are many uses for the standard diagnostics suite tests, and you'll benefit from extending it with your own OEM diagnostics suite to test your custom hardware, such as A/D controllers, muxes, process controllers, touch screens, and other equipment.

16.3.5 Start System BIOS Debugger Setup Screen

The Embedded BIOS Integrated Debugger may be invoked from the Setup Screen main menu, as well as a boot activity. Once invoked, the debugger will display the debugger prompt:

```
EB43DBG:
```

and await debugger commands. To resume back to the Setup Screen main menu, type the following command, which instructs the debugger to "go":

```
EB43DBG: G  (ENTER)
```

16.3.6 Start RS232 Manufacturing Link Setup Screen

The Embedded BIOS Manufacturing Mode may be invoked from the Setup Screen main menu, as well as a boot activity. Once invoked, Manufacturing Mode takes over the system and freezes

the console of the system (Fig 6). The host can resume operation of the system and give control back to the NEWBOARD Setup Screen system with special control software.

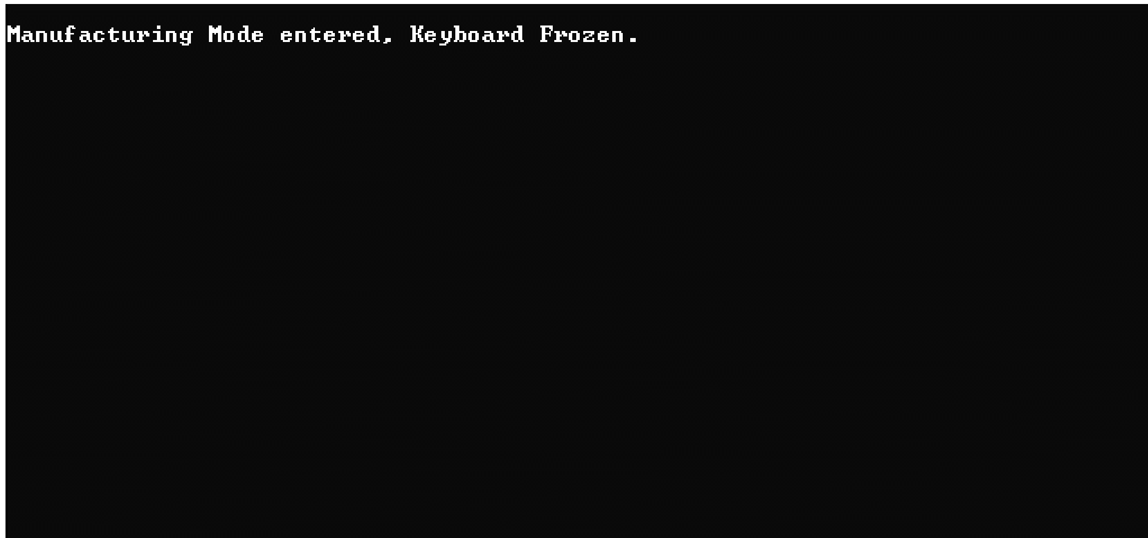


Fig 6. The Start RS232 Manufacturing Link Setup Screen Provides Access to Manufacturing Mode.

16.3.7 Other Pre-Boot Setup Screens

Embedded BIOS provides other Setup screens to the OEM as well. The following are available:

- DEMONSTRATION SCREEN

  This screen is enabled for sample BIOSes built for evaluation boards by General Software. It displays information about the company, its products, and how to contact General Software for further licensing information.

- POWER FEATURES CONFIGURATION

  The power management feature configuration screen (optional) contains fields which enable power management modes for the target. This allows the user to disable certain modes if they might interfere with the application.

- POWER TIMEOUTS CONFIGURATION

  The power management timeout configuration screen (optional) contains fields which define device inactivity timers which, when expired, cause the device to be powered down by the Power Management System.

- PASSWORD CONFIGURATION

  The password screen (optional) allows the user to set a password into the system's CMOS memory so that a password is required when the system boots.

- HARD DISK UTILITIES

This (optional) screen is available for OEM extensions when supporting special hard disks that require OEM drivers.

- FORMAT INTEGRATED RAM DISK

The RAM disk formatting screen (optional) reformats the RAM disk, if enabled. This is available for maintenance of the RAM disk in the field.

- FORMAT INTEGRATED FLASH DISK

The Flash disk formatting screen (optional) reformats the Resident Flash Disk (RFD), if enabled. This is available for maintenance of the RFD in the field.

- RESET CMOS TO LAST KNOWN VALUES

This option causes SETUP to restore the values it had prior to any edits performed during the current SETUP session.

- RESET CMOS TO FACTORY DEFAULTS

This option causes SETUP to reset CMOS with the values that are defined in INC\CONFIG.INC as factory defaults.

- WRITE TO CMOS AND EXIT

This option causes SETUP to save the current edits to CMOS and reboot the target, causing the new edits to take effect.

- EXIT WITHOUT CHANGING CMOS

This option causes SETUP to reboot the target without saving any changes made during the SETUP session.

Chapter 17

POWER ON SELF TEST (POST)

One of the most important functions of the BIOS is the Power-On Self-Test, or POST; a complex procedure that runs when a machine is first powered-on, or is warm-booted with any of a variety of mechanisms.

POST is responsible for every facet of initializing the hardware to a steady-state, so that the operating system or application can rely on BIOS services and various interrupts to perform their functions. Without POST, the BIOS services and hardware have no connection with one another. Indeed, without POST, the hardware cannot even begin working.

17.1 Initialization Without a Stack or RAM

The actual POST code begins executing at physical address 000FFFF0h, or in real-mode 16:16 terms, F000:FFF0. Because POST must assume that all of the peripherals are inoperative, it must also assume that the DRAM is inoperative, because its refresh controller is not yet initialized. This means that POST must begin its process without using any RAM, including using PUSH, POP, CALL, RET, INT, or IRET instructions.

In practice, this is quite a difficult task. While it is tempting to turn on the DRAM refresh right away, in fact quite a few peripherals must be initialized to begin the refresh process. In PC-compatible systems, the 8253 programmable interval timer's T1 timer must be tested and initialized to count at a rate of approximately 15us per step. Then, the primary 8237 DMA controller must be initialized and programmed so that a DMA channel begins performing verify operations in collaboration with the 8253 counter. Finally, the page register file on a PC/AT-compatible platform must be initialized to allow the refreshing to occur.

After DRAM refresh has started, the RAM still isn't ready to use. Because modern ISA-compatible motherboards are built around any of the hundreds of chipsets available on the market today, the chipset must be initialized (in its own proprietary fashion) before memory can be used. The reason for this is that chipsets need to be programmed to determine how many SIMMs (memory modules) are present, what size they are, and ultimately, how they are to be interleaved. Once the DRAM geometry is detected and the chipset is initialized, a stack can be used.

Even though a large portion of the POST procedure runs without any RAM support (meaning without a stack), EMBEDDED BIOS remains procedurized in a compact way. This is accomplished by using the BP CPU register as a return address when calling coroutines.

There are two sets of macros defined in the MACROS.INC file, that are used in the BIOS source code to define and call procedures that hide the details between stack-based procedures and register-return-based coroutines. These are illustrated below in sample code fragments.

17.1.1 Stack-Based Procedures

The first code fragment shows how a stack-based procedure is declared and called with the **DefProc**, **EndProc**, and Pcall macros. The **DefProc** macro creates the PROC assembly statement, and also adds a PUBLIC statement because the PUBLIC operand is specified. The EndProc macro automatically provides a RET instruction, and an ENDP statement to terminate the procedure. The Pcall macro translates into a CALL instruction that may be augmented by an EXTRN or EXTRNDEF assembly statement in the event that it determines that **Foo** is not in the current module.

```
DefProc Foo, PUBLIC             ; declare a stack-based procedure.
        ...                     ; the procedure body.
EndProc Foo                     ; end of the procedure.

        ...
        Pcall  Foo              ; invoke the procedure via CALL.
```

17.1.2 Register-Based Coroutines

The next code fragment shows how a register-based coroutine is declared and called with the **DefRtn**, **EndRtn**, and **Rcall** macros. The **DefRtn** macro creates a PROC assembly statement, and adds a PUBLIC statement as its **DefProc** counterpart does. The **EndRtn** macro adds a special JMP BP instruction to return to the coroutine's caller, and then an ENDP statement to terminate the procedure. The **Rcall** macro, distinguished from the **Pcall** macro, translates into a MOV instruction that sets the return address into the BP CPU register, and then a JMP instruction that jumps directly to the label. As with **Rcall**, the EXTRN or EXTRNDEF assembly statements may be assembled if the target label is not within the current module.

```
DefRtn Foo, PUBLIC              ; declare a register-based coroutine.
        ...                     ; the routine body.
EndRtn Foo                      ; end of the routine.

        ...
        Rcall  Foo              ; invoke the routine via a JMP.
```

The macros used to manage procedures and coroutines are strongly-typed; that is, they actually generate labels that are a modified form of the labels in the source code. For example, the **DefProc** macro translates the label "**Foo**" into an internal form "**_p_Foo**" that indicates **Foo** is a procedure. Similarly, the **DefRtn** macro translates the label "**Foo**" into "**_r_Foo**", indicating that **Foo** is a coroutine. This ensures that a coroutine is not called with stack-based linkage, for example.

17.1.3 Hybrid Procedures With Dual Linkage

A third type of procedure, called a hybrid procedure, is declared with the **DefHyb** and **EndHyb** macros. These procedures are callable from either the **Pcall** or Rcall macros. Hybrid procedures

work by generating a coroutine with the standard coroutine linkage, and then an additional procedural-based routine that calls the coroutine with the **Rcall** macro. Hybrid procedures enable Embedded BIOS to make use of code at POST time when no stack is available, and later during run-time with stack-based linkage.

The following code fragment illustrates how a hybrid procedure can be declared with the **DefHyb** and **EndHyb** macros, and then called with either the **Rcall** or Pc**a**ll macros.

```
DefHyb Foo, PUBLIC              ; declare a hybrid routine.
       ...                      ; the routine body.
EndHyb Foo                      ; end of the routine.

       ...
       Rcall  Foo               ; invoke the routine via a JMP.
       Pcall  Foo               ; invoke the routine as a procedure.
```

17.2 Early Initialization Process

EMBEDDED BIOS has a flexible POST architecture that allows the OEM to affect the standard initialization process in the CPU, Chipset, and Board Personality Modules. This section describes how POST proceeds, so that the OEM can determine where proprietary initialization code may be placed.

Mainline POST can be found in routine POST, in module `SYSTEM\POST.ASM`. Following along with the source code in that module can be helpful to understand how POST operates.

For detailed specifications about the BPM procedures described below, see Chapter 20.

17.2.1 BoardInit0 Processing

After disabling interrupts, POST calls **BoardInit0** in the BPM to perform extremely early initialization of the CPU, chipset, and board, and to determine whether a warm or cold boot is being processed. This routine, and the subsequent code that follows the call to the routine in POST, use the SP register to communicate about whether a bootstrap is cold or warm.

By default, BoardInit0 calls **CpuInit0** and **CsInit0** (in that order) to initialize the CPU and chipset, respectively. For 186-class machines, **CpuInit0** must establish chip selects so that the remainder of the BIOS can run properly.

This level of initialization focusses on determination of whether the target was booted warm or cold, and should limit activities to that issue. In some systems, the CPU or chipset registers may need to be initialized to a known state (i.e., shadowing disabled, cache disabled, etc.) so that further processing in POST does not produce erroneous results.

17.2.2 PostTestResetValue Processing

This routine is called by POST to determine if, on 286 platforms, POST was entered in the process of switching from protected mode to real mode. This method is obsolete on 386 and above platforms, and does not apply to 186 and below systems. Some systems,

17.2.3 PostCodeComInit Processing

This routine is called by POST to initialize the UART associated with the **POSTCODECOM** macro, if enabled in the BIOS build. This routine requires that the UART's I/O ports be visible in the I/O address space; that is, the UART must be enabled in the hardware. If the UART is implemented in a Super I/O part or on a chipset or CPU with a chipset such as the AMD SC300, SC310, SC400, or SC410 processors, then the UART must be enabled in a proprietary way in **BoardInit0** so that routine **PostCodeComInit** can access the UART's registers.

Because **POSTCODECOM** requires a prior call to **PostCodeComInit** to initialize the UART, and since this call is made after POST's call to **BoardInit0**, **POSTCODECOM** cannot be used in **BoardInit0**, unless an additional call is made to this routine in **BoardInit0** after enabling the UART.

17.2.4 BoardInit1 Processing

This routine is called by POST to perform the bulk of the initialization work, including chipset programming, high-integration CPU programming, and peripheral programming (Super I/O, FDC, and other components).

By default, routine **BoardInit1** calls **CpuInit1** and **CsInit1**, in that order, to allow the CPM and CSPM modules to perform the bulk of their initialization.

It is suggested that this routine load all of the chipset registers with appropriate values at this point during initialization. Some CPUs, such as the AMD SC300, may not initialize their internal configuration registers to the values specified in the manuals. These registers, when left uninitialized, can cause aberrant system operation in areas even unrelated to the registers' functions.

The most common way to load these values is to establish a table with entries consisting of three components each: the register index, the register contents, and a bitmask to mask the old contents of the register before OR'ing in the new value.

17.2.5 Main POST Processing

Once the early initialization calls to **BoardInit0** and **BoardInit1** are performed, POST calls many subroutines to initialize components necessary for the operation of DRAM. In order for DRAM to work, power must have stabilized, a default addressing mode must have been established in the chipset programming inside **BoardInit1**, and refresh must be active in some form (either handled by the chipset programming in **BoardInit1**, or with an 8237 and an 8254).

Many other components are actually prepared in order to make this happen. For example, PORT B on PC/AT class machines is initialized, as is the keyboard controller, I/O port 92h, and CMOS. Once these components are ready, and DRAM is ready to be analyzed, control passes to BoardMemConfig, which actually determines the geometry of DRAM.

17.2.6 BoardMemConfig Processing

This routine is called to determine the size, shape, configuration, and number of banks of DRAM in the system, and to program the chipset to support the DRAM in that configuration. This is not an easy job. Four elements conspire to make this task difficult: (1) there is no stack available for PUSH, POP, CALL, or RET instructions, making it difficult to use only the CPU registers; (2) the complexity of interpreting the chipset documentation, which may be in error and possibly

have published errata; (3) there are usually many possible configurations, a subset of which is generally used in any given adaptation; and (4) the task usually involves switching to protected mode and back to real mode again to inspect areas of memory, without any stack, using only **POSTCODECOM** or I/O writes to a 7-segment display.

Some chipset vendors publish recommended algorithms for identifying banks of memory, and for determining if a bank is EDO (Early Data Out) or not. These published algorithms are frequently wrong and out-of-date. Fortunately, General Software works with the major embedded chipset manufacturers to identify and test algorithms that work for specific board designs. These algorithms are then provided in BPMs to General Software customers.

Note that this code is in the BPM, and not in the CSPM. One might think at first glance that the chipset would be responsible for all memory initialization, but in actuality it is the way the chipset is used and the way memory is wired to the chipset that determines how memory works, and so the board is actually the best candidate for this code.

If you are writing a **BoardMemConfig** routine for a new board, it is best to survey available **BoardMemConfig** code to determine if code similar to your needs has already been written, so that it can be used as a base.

It should be noted that some implementors have chosen to place all of the chipset initialization code, including the code that would normally be placed in this routine, in **BoardInit1**. This is fine for many designs, and we must emphasize that these routines represent opportunities to fill-in the blanks, rather than requirements that certain processing happen at a certain point during POST.

17.2.7 Further POST Processing

Once DRAM is operational, POST initializes the remainder of the system components with the luxury of a stack. This initialization includes enabling the cache, shadow memory, DMA controllers, and interrupt controllers, until **BoardInit4** is called.

17.2.8 BoardInit4 Processing

Routine **BoardInit4** is called to allow the OEM to gain control shortly before the interrupt vectors are established and the keyboard and video are initialized. This makes it possible for the OEM to add proprietary code to the system that downloads firmware into the keyboard controller or video controller. This routine is rarely used.

17.2.9 BoardInit6 Processing

Routine **BoardInit6** is called after the keyboard and video controllers are initialized, including VGA BIOS extensions, so that the OEM can gain control at the right time to determine if console I/O should be diverted to a serial port.

Typically, this involves the testing of a bit in some I/O port to test for the presence of an RS-232 cable connection with a modem, or perhaps the existence of a jumper or shunt. Then, based on the hardware finding, the code can issue an INT 15h function to switch console I/O to the standard keyboard and screen, or a COM port. See Chapter 21 for details about this function.

This routine may be used effectively for other purposes as well, but it is ideally positioned for this particular purpose.

Because the code immediately after the call to **BoardInit6** in POST calls the PRINTF formatting package for output of the sign-on banner, code in **BoardInit6** can use PRINTF macro calls, and use the integrated debugger for debugging initialization of this routine or the rest of POST. To break into the debugger in **BoardInit6** or elsewhere following the call to **BoardInit6**, place an "`INT 3`" instruction at the point where the debugger should break.

17.2.10 Device Initialization Processing

The remainder of POST focusses mostly on device initialization. After the banner is printed with a call to **PostPowerOnMsg**, POST initializes the cache, tests low memory, tests extended memory, tests the floppy disks, tests IDE drives, calls the ROM BIOS extensions, initializes the printer ports, enables power management, and initializes the PS/2 mouse.

17.2.11 Final POST, BoardInit8 Processing

After all the hardware components have been initialized, POST transfers control to three major components: SETUP, Manufacturing Mode, and the Operating System Loader. These components only activate if system conditions dictate that they should be activated. For example, to enter SETUP, a soft error should have been encountered, or the user must have pressed <DEL> or ^C during the memory count-up.

After SETUP and Manufacturing Mode have their chance to run, a call to **BoardInit8** is made. This routine's purpose is to load the chipset and any Super I/O components with OEM-defined CMOS settings. **BoardInit8** does this by reading the unarchitected CMOS cells above the ones normally reserved for the core BIOS, interpreting their contents, and setting chipset registers and Super I/O registers to values that reflect the intent of the CMOS cell contents.

17.3 POST Codes

During the POST procedure, errors can be encountered. Errors can occur because peripherals are incorrectly configured, or even because hardware is not working properly. During early phases of POST, no video services are available. If available, the speaker is programmed to beep a number of times to indicate problems during early POST.

After video services are available, POST uses INT 10h services to display error messages. For example, when CMOS is corrupted or incorrectly configured for the hardware detected by the BIOS, POST displays a message that explains the nature of the CMOS problem, and leads the user to enter the Setup screen system.

POST uses writes to port 80h (or its equivalent, if this port is redefined by the OEM) to indicate its progress. If a logic analyzer is available, the progress can be monitored. Use file `INC\POST.INC` for the current POST codes.

17.3.1 Speaker POST Codes

Before video services are available, the following beep codes are used to indicate specific problems during early POST. These codes are defined in the `POSTERR.INC` include file.

```
POST_BEEP_REFRESH   =      1      ; memory refresh is not working.
POST_BEEP_PARITY    =      2      ; parity error in 1st 64KB.
POST_BEEP_BASE64KB  =      3      ; memory failure in 1st 64KB.
POST_BEEP_TIMER     =      4      ; timer T1 not operational.
POST_BEEP_CPU       =      5      ; CPU test failed.
POST_BEEP_GATEA20   =      6      ; gate A20 failure.
POST_BEEP_DMA       =      7      ; DMA page/base registers.
POST_BEEP_VIDEO     =      8      ; video error (nonfatal).
POST_BEEP_KEYBOARD  =      9      ; keyboard failure.
POST_BEEP_SHUTDOWN  =      10     ; CMOS shutdown register failed.
POST_BEEP_CACHE     =      11     ; external cache is not working.
POST_BEEP_BOARD     =      12     ; board initialization failure.
POST_BEEP_PASSWORD  =      1      ; incorrect password.
POST_BEEP_LOWMEM    =      13     ; exhaustive low memory test.
POST_BEEP_EXTMEM    =      14     ; exhaustive extended memory test.
POST_BEEP_CMOS      =      15     ; CMOS restart byte failed.
POST_BEEP_ADDRESS_LINE=    16     ; address line test failed.
POST_BEEP_DATA_LINE =      17     ; data line test failed.
POST_BEEP_INTERRUPT =      18     ; interrupt controller failure.
```

17.3.2 Video POST Messages

After video services are available, the following messages can be displayed during POST. Note that some messages are status only, and others indicate errors. These messages are defined in the POSTERR.INC include file.

```
;       Good POST messages.

POST_MSG_PRESSDEL       EQU     'Press <DEL> to enter SETUP . . .',0
POST_MSG_PLEASEWAIT     EQU     'Please Wait . . .',0
POST_MSG_BANNER1        EQU     '\r\r\r\r\rGeneral Software ',0
POST_MSG_BANNER2        EQU     ' Embedded BIOS (tm) Version $u.$03u\n',0
POST_MSG_BANNER3        EQU     'Copyr (C) 2000 General Software, Inc.\n',0
POST_MSG_BANNER4        EQU     'For BIOS Licensing, call (800) 850-5755 ...\n',0

;       Bad POST messages.

POST_MSG_CMOS           EQU     'CMOS failure',0
POST_MSG_GATEA20        EQU     'Gate A20 failure',0
POST_MSG_DMA            EQU     'DMA failure',0
POST_MSG_FDD            EQU     'Floppy controller failure',0
POST_MSG_HDD            EQU     'Hard disk controller failure',0
POST_MSG_INT            EQU     'Interrupt controller failure',0
POST_MSG_PARITY         EQU     'Memory parity error',0
POST_MSG_ADDRLINE       EQU     'Address line short',0
POST_MSG_CACHE          EQU     'Cache memory failure',0
POST_MSG_TIMER          EQU     'Timer failure',0
POST_MSG_CMOSBATTERY    EQU     'CMOS battery low',0
POST_MSG_CMOSCKSUM      EQU     'CMOS checksum lost',0
POST_MSG_CMOSNOTSET     EQU     'CMOS options not set',0
POST_MSG_CMOSDISPLAY    EQU     'CMOS display type mismatch',0
POST_MSG_CMOSMEMORY     EQU     'CMOS memory size mismatch',0
POST_MSG_CMOSTIME       EQU     'CMOS time/date not set',0
POST_MSG_DISKBOOT       EQU     'Diskette boot failure',0
POST_MSG_DISPSWITCH     EQU     'Display switch not set properly',0
```

```
POST_MSG_INVBOOTDISK     EQU     'Invalid boot diskette',0
POST_MSG_KBDLOCKED       EQU     'Keyboard is locked',0
POST_MSG_KEYBOARD        EQU     'Keyboard failure',0
POST_MSG_NOBOOT          EQU     'No bootable media',0
POST_MSG_LPT1            EQU     'LPT1 initialization failure',0
POST_MSG_LPT2            EQU     'LPT2 initialization failure',0
POST_MSG_LPT3            EQU     'LPT3 initialization failure',0

POST_MSG_BOARD           EQU     'Board configuration problem',0
POST_MSG_CHIPSET         EQU     'Chipset configuration problem',0
POST_MSG_CPU             EQU     'CPU configuration problem',0
```

Chapter 18

CPU PERSONALITY MODULES

EMBEDDED BIOS has a flexible, scalable architecture that enables it to support a wide spectrum of CPUs, from low-end 186-class microcontrollers to high-end Pentium and P6 CPUs, especially those CPUs with integrated peripheral controllers.

While the mainstream CPUs don't contain peripherals such as UARTs, DMA controllers, interrupt controllers, and DRAM refresh controllers, CPUs aimed at the embedded and consumer electronics markets do have extra logic. This logic must be initialized and configured, and the code required to do this is different for each CPU type.

The EMBEDDED BIOS core does not contain any CPU-specific code that deals with these peripheral controllers, but instead contains call-outs to a special module, called a CPU Personality Module (CPM), that contains the CPU-specific code.

The CPU class for a given BIOS build is selected with the **CPUCLASS** parameter in the project file. For systems with CPUs that have no integrated nonstandard controllers, the **NOCPU** CPU class is used. This includes generic 8086, 80286, 386, 486 CPUs; Intel Pentium, Pentium II, Pentium III, and Celeron processors; AMD K6, K6-2, K6-3, K6-E, and Athlon processors; and other compatible processors.

Intel's 80186-EC and 80386-EX CPUs, and AMD's Am186xx CPUs are examples of high-integration CPUs that are best supported with the CPM model.

Some CPUs with high integration are actually more chipset-like than they are CPU-like. Examples include AMD's SC300, SC400, and SC520 family of processors; Ali's M6117, and STMicroelectronics' STPC. These chips are best supported with the **NOCPU** CPU class, and then with Chipset Personality Modules (CSPMs), a topic of Chapter 19, and Board Personality Modules (BPMs), a topic of Chapter 20.

This chapter describes how CPMs fit into the EMBEDDED BIOS build, and the details of the CPM interface called by the core EMBEDDED BIOS code that supports different classes of CPUs.

18.1 How CPM Override Routines Work

The EMBEDDED BIOS build links together many modules that implement BIOS services, hardware managers, and function redirectors. Additionally, the module `CPU.ASM` in the `SYSTEM` directory is assembled and linked into the system build.

Actually, the `CPU.ASM` module is a shell that uses assembly `INCLUDE` statements to include the CPM selected with the **CPUCLASS** parameter in the Project file. The `CPU.ASM` module also contains default versions of the CPM routines, should the CPM only have a few routines defined. Thus, a CPM may be comprised of zero, any, or all, of the routines documented later in this chapter. The routines defined in a CPM are called *override routines*, and have the special "**OVERRIDE**" parameter in their **DefProc** or **DefRtn** procedure definition MACROs.

The **NOCPU** CPM is an excellent example of this default system in action. The **NOCPU** CPM contains no routines at all, so that when **NOCPU** is selected as the **CPUCLASS** parameter value, all of the default routines are used.

On the other hand, the **TEMPLATE** CPM is an excellent example of the reverse. This module is an example of a CPM that has all of the routines defined, although they happen to contain exactly the same code that the default routines do. The intent of providing this CPM with EMBEDDED BIOS is to enable the OEM to simply clone the **TEMPLATE** CPM, change routines that are necessary, and then delete the ones that go unchanged. This facilitates rapid development of new CPMs in the OEM environment.

18.2 How CPMs are Packaged in Files

From a project-management point of view, a CPM consists of at least two files; one is an .ASM file that contains the routines provided by the OEM that override the default CPM functions, and the other is an .INC file that contains manifest constants, macro definitions, and other definitions needed by the CPM, and possibly by the BPM under certain circumstances.

All CPMs have a name (from 1 to 8 filename characters), and the .ASM and .INC files must carry this name. For example, the NOCPU CPM consists of a `NOCPU.ASM` file and a `NOCPU.INC` file.

Each CPM's files are contained within a subdirectory of the CPUS directory. The subdirectory must have the same name as the CPM. For example, if the BIOS main directory is `C:\BIOS43`, then the **NOCPU** CPM's files would be contained in the `C:\BIOS43\CPUS\NOCPU` directory.

Additional source files may be located in the CPM's subdirectory, but it is up to the OEM to define their contents and ensure that they are assembled as a part of the BIOS build. It is recommended that no core BIOS files be modified to include these sources; instead, it is a good idea to include them with `INCLUDE` statements in the CPM's .ASM file.

18.3 Other CPU Personality Modules

If you don't want to write your own personality module for a new CPU, it may actually already be available through General Software. For example, the Intel 80186-EC and 80C386-EX CPUs and AMD Am186 CPUs are supported with separate CPMs available from General Software (contact General Software for more information).

18.4 The CPM Interface

All CPM implementations export the same set of functions callable from the core system BIOS. This section documents the functions that must be implemented in a CPM.

Be careful to define the proper function type when creating new CPMs. Some CPM functions are written as coroutines, using the **DefRtn**/**EndRtn** macros. Others are written as procedures, using the **DefProc**/**EndProc** macros. If these are confused, there will be errors during the link of the system BIOS.

Unless otherwise specified, routines always return to their caller, and do not modify any register contents. The CPU flags may be destroyed by CPM functions, except that the carry flag is normally used to indicate success if clear, or failure if false. Other flags, such as the zero flag, are destroyed at random by the functions. The direction flag, however, must be preserved.

Functions declared as coroutines cannot modify the BP CPU register, as it is used for return linkage. If they must use BP because they need to call another coroutine as a subroutine, then BP can be placed into another register to save the original return address.

Some functions are entered with the interrupt flag cleared (disallowing interrupts). These functions cannot enable interrupts for any reason, as they are used at times when interrupt management has not been established. Other functions are entered with a random interrupt flag as the context dictates. In this case, the routine can manipulate the interrupt flag if it wishes. Unless otherwise specified, a routine **may** modify the IF flag, but must restore it to its original state.

18.4.1 CpuBeep Routine

The **CpuBeep** function is called with routine linkage to cause the speaker to start or stop beeping.

If no hardware is available in the CPU, then this function should return with the carry flag clear.

This routine will be called **OPTION_SUPPORT_SOUND** is enabled; and then if **OPTION_SOUND_CPU** enabled, or if **OPTION_SOUND_BOARD** is enabled and **BoardBeep** calls this routine.

If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation.

Input Parameters:

AL - 1 to enable tone, 0 to disable tone.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

AX, CX, DX, Flags.

18.4.2 CpuDisableA20 Hybrid

The **CpuDisableA20** function is called with hybrid (dual) linkage to disable the A20 line gate hardware on-board the CPU, if it exists. Normally, the A20 gate on 80286 and above CPUs is provided by external components, such as the 8042 keyboard controller, port 92h, or the chipset.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be disabled.

This routine is entered with interrupts disabled and cannot reenable them. It can also be called with procedure linkage or routine linkage, since it is a hybrid function. Be careful not to modify BP or use any RAM while inside this function, as it may be called during a period of time where DRAM is inoperative. Thus, this routine cannot use any PUSH, POP, CALL, RET, INT, or IRET instructions.

This routine will be called if **OPTION_A20_CPU** enabled, or if **OPTION_A20_BOARD** is enabled and **BoardDisableA20** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

18.4.3 CpuDisableCache Procedure

The **CpuDisableCache** function is called with procedure linkage to disable the L1 cache hardware on-board the CPU, if it exists. Intel i486 and Pentium CPUs have on-board caches that significantly improve performance when this feature is implemented.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be disabled.

This routine will be called if **OPTION_SUPPORT_CACHE** is enabled; and then if **OPTION_CACHE_CPU** enabled, or if **OPTION_CACHE_BOARD** is enabled and **BoardDisableCache** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

18.4.4 CpuDisableDmaCtrl Routine

The **CpuDisableDmaCtrl** function is called with routine linkage to disable the integrated DMA controller hardware on-board the CPU, if it exists. Disabling means to reset the controller so that no DMA processes are running after the routine returns to its caller.

If no DMA controller hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be disabled.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_DMA_CPU** enabled, or if **OPTION_DMA_BOARD** is enabled and **BoardDisableDmaCtrl** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.5 CpuDisableIntCtrl Hybrid

The **CpuDisableIntCtrl** function is called with hybrid (dual) linkage to disable the integrated interrupt controller hardware on-board the CPU, if it exists. Disabling in this context means to cause the interrupt control to reset to the condition where there are no pending interrupts to be serviced, and no interrupt levels enabled (or unmasked) upon return from this routine.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be disabled.

This routine is entered with interrupts disabled and cannot reenable them. It can also be called with procedure linkage or routine linkage, since it is a hybrid function. Be careful not to modify BP or use any RAM while inside this function, as it may be called during a period of time where

DRAM is inoperative. Thus, this routine cannot use any PUSH, POP, CALL, RET, INT, or IRET instructions.

This routine will be called if **OPTION_INT_CPU** enabled, or if **OPTION_INT_BOARD** is enabled and **BoardDisableIntCtrl** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

18.4.6 CpuDisableWatchdog Procedure

The **CpuDisableWatchdog** function is called with procedure linkage to disable the integrated watchdog timer controller hardware on-board the CPU, if it exists. Disabling in this context means to cause the watchdog timer to stop running, so that it will not possibly expire without restarting it.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be disabled.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_WATCHDOG** is enabled; and then if **OPTION_WATCHDOG_CPU** enabled, or if **OPTION_WATCHDOG_BOARD** is enabled and **BoardDisableWatchdog** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

18.4.7 CpuEnableA20 Hybrid

The **CpuEnableA20** function is called with hybrid (dual) linkage to enable the A20 line gate hardware on-board the CPU, if it exists. Normally, the A20 gate on 80286 and above CPUs is provided by external components, such as the 8042 keyboard controller, port 92h, or the chipset.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be enabled.

This routine is entered with interrupts disabled and cannot reenable them. It can also be called with procedure linkage or routine linkage, since it is a hybrid function. Be careful not to modify BP or use any RAM while inside this function, as it may be called during a period of time where DRAM is inoperative. Thus, this routine cannot use any PUSH, POP, CALL, RET, INT, or IRET instructions.

This routine will be called if **OPTION_A20_CPU** enabled, or if **OPTION_A20_BOARD** is enabled and **BoardEnableA20** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

18.4.8 CpuEnableCache Procedure

The **CpuEnableCache** function is called with procedure linkage to enable the L1 cache hardware on-board the CPU, if it exists. Intel i486 and Pentium CPUs have on-board caches that significantly improve performance when this feature is implemented.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be enabled.

This routine will be called if **OPTION_SUPPORT_CACHE** is enabled; and then if **OPTION_CACHE_CPU** enabled, or if **OPTION_CACHE_BOARD** is enabled and **BoardEnableCache** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

18.4.9 CpuEnableApm Procedure

The **CpuEnableApm** function is called during POST with procedure linkage to enable whatever APM functionality is available in the CPU.

This routine will be called if **OPTION_SUPPORT_APM** is enabled; and then if **OPTION_POWERMAN_CPU** enabled.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

All.

18.4.10 CpuEnableDmaCtrl Routine

The **CpuEnableDmaCtrl** function is called with routine linkage to enable the integrated DMA controller hardware on-board the CPU, if it exists. Enabling means to cause the DMA controller to be ready to accept a programming sequence for a DMA operation. This routine is provided for symmetry with **CpuDisableDmaCtrl**, and is rarely used.

If no DMA controller hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be enabled.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_DMA_CPU** enabled, or if **OPTION_DMA_BOARD** is enabled and **BoardEnableDmaCtrl** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.11 CpuEnableIntCtrl Hybrid

The **CpuEnableIntCtrl** function is called with hybrid (dual) linkage to enable the integrated interrupt controller hardware on-board the CPU, if it exists. Enabling in this context means to cause the interrupt controller to be ready to receive unmask or EOI commands and handle interrupts from that point forward. This routine is rarely used, but is provided for symmetry with **CpuDisableIntCtrl**.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be enabled.

This routine is entered with interrupts disabled and cannot reenable them. It can also be called with procedure linkage or routine linkage, since it is a hybrid function. Be careful not to modify BP or use any RAM while inside this function, as it may be called during a period of time where DRAM is inoperative. Thus, this routine cannot use any PUSH, POP, CALL, RET, INT, or IRET instructions.

This routine will be called if **OPTION_INT_CPU** enabled, or if **OPTION_INT_BOARD** is enabled and **BoardEnableIntCtrl** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

18.4.12 CpuEnableWatchdog Procedure

The **CpuEnableWatchdog** function is called with procedure linkage to enable the integrated watchdog timer controller hardware on-board the CPU, if it exists. Enabling in this context means to start the watchdog timer running, so that if the timer is not reset within one expiration period, the timer will expire.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be enabled.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_WATCHDOG** is enabled; and then if **OPTION_WATCHDOG_CPU** enabled, or if **OPTION_WATCHDOG_BOARD** is enabled and **BoardEnableWatchdog** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

18.4.13 CpuEoi Procedure

The **CpuEoi** function is called with procedure linkage to issue an "end-of-interrupt" command to the on-board CPU interrupt controller. CPUs such as 80186 and 80386-EX have on-board programmable interrupt controllers that must be reset after an interrupt is processed.

This routine will be called if **OPTION_INT_CPU** enabled, or if **OPTION_INT_BOARD** is enabled and **BoardEoi** calls this routine.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

None, NOT EVEN FLAGS.

18.4.14 CpuExtRwCtrl Procedure

The **CpuExtRwCtrl** function is called with procedure linkage to read from or write to the modem control register associated with an on-board serial port, if CPU-integrated serial port hardware exists.

If no hardware is available in the CPU, then this function should return with the carry flag set. If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation.

Input Parameters:

DX - serial port to read/write, as follows:

01h - first CPU port.
02h - second CPU port.
03h - third CPU port.
04h - fourth CPU port.

AL - subfunction code, as follows:

00h - read control register.
01h - write control register.

BL - if writing, this is the value to write.

DI - timeout in 18.2 Hz ticks to use for the operation.

Output Parameters:

CY - set if failure, else clear if success.
BL - control register data, as follows:

bits 7,6,5 - reserved.
bit 4 - loop for testing if set.
bit 3 - OUT2.
bit 2 - OUT1.
bit 1 - request to send.
bit 0 - data terminal ready.

Unpreserved Registers:

Flags.

18.4.15 CpuFloppyDma Procedure

The **CpuFloppyDma** function is called with procedure linkage to program the CPU DMA controller to perform a DMA operation for floppy disk I/O. Typically, this routine sets the AL register to the proper DMA channel for floppy disk DMA, and calls **CpuSetupDma**, although other actions, such as enabling package pins on high-integration CPUs, may be necessary.

If on-board DMA hardware is not available, this routine should return with carry set. Otherwise, it should perform the operation and return with carry clear if the operation was successful, or set if the DMA operation failed.

This routine will be called if **OPTION_SUPPORT_FLOPPY** and **OPTION_FLOPPY_DMA** are enabled; and then if **OPTION_DMA_CPU** enabled, or if **OPTION_DMA_BOARD** is enabled and **BoardFloppyDma** calls this routine.

Input Parameters:

DX:BX - 32-bit linear address of buffer.
CX - 16-bit byte count for transfer.
AH - DMA operation code, as follows:

00h - DMA to memory from I/O (read).
01h - DMA from memory to I/O (write/format).
02h - DMA from I/O, no memory (verify).

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

AX, BX, CX, DX, Flags.

18.4.16 CpuGetProcessorName Procedure

The **CpuGetProcessorName** function is called with procedure linkage to return a pointer to an ASCIIZ string that corresponds to the type of CPU running in the target.

The default **NOCPU** module uses Intel-based names, such as 80386, 80486, Pentium, Pentium II, and so on, for its processor name table. For other processor names, separate CPU modules should be used with different strings in the processor table.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.
CS:AX - if successful, 16:16 pointer to ASCIIZ CPU name.

Unpreserved Registers:

All but SS, SP.

18.4.17 CpuGetProcessorType Procedure

The **CpuGetProcessorType** function is called with procedure linkage to return an ordinal number that specifies what type of CPU is running in the target, as well as the model number if applicable.

As an implementation caution, the adaptation engineer is advised that the techniques used in the **NOCPU** CPM to implement this function will not port to the NEC V-Series CPU line, since the **NOCPU** tests rely on side-effects of instructions that may be implemented differently by NEC.

The techniques used in the default **NOCPU** CPM use instructions that are supported at the level indicated by the **CPU_TYPE** configuration parameter.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.
AX - if successful, processor ordinal as follows:

0000h - Intel 8086 or compatible.
0001h - Intel 80186 or compatible.
0002h - Intel 80286 or compatible.
0003h - Intel 80386 or compatible.
0004h - Intel i486, 4x86, MediaGX or compatible.
0005h - Intel Pentium, MediaGX MMX, 5x86, 6x86, or compatible.
0006h - Intel Pentium II, Pentium III, Celeron, 6x86MX, K6, etc.

BX - if successful, processor model, or FFFFh if no step available.

Unpreserved Registers:

All but SS, SP.

18.4.18 CpuHookVectors Procedure

The **CpuHookVectors** function is called with procedure linkage to re-vector any interrupt vectors that need to e changed to support this CPU. For example, the 80C186-EC module gets timer ticks through INT 76h, and calls the **Int08Isr** routine manually from there with an INT 08h instruction.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

All.

18.4.19 CpuInit0 Routine

The **CpuInit0** function is normally called with routine linkage to perform very early initialization of the CPU, including participating in the decision about whether a warm boot or a cold boot has occurred.

Called from **BoardInit0**, this function accepts as a parameter in the SP register a value of 0 if a cold boot has occurred, or -1 if the BPM has determined that a warm boot may have occurred. If this routine determines that a cold boot has occurred, then it sets SP to 0. Similarly, if it determines that a warm boot may have occurred, then it sets SP to -1. If it has no additional information to provide POST during the boot detection process, then it does not change SP.

Commonly, this routine is also used by 186-class CPMs to initialize the chip selects so that the entire BIOS ROM image is visible in the address space.

This routine is entered with interrupts disabled and cannot enable them.

The implementor of routine **BoardInit0** may choose to handle initialization of the CPU in the BPM; consequently, this routine may not be called in some adaptations.

Input Parameters:

DS - BIOS data segment (40h).
SP - 0 if BPM detects cold boot, else -1 if possible warm boot.

Output Parameters:

DS - BIOS data segment (40h).
SP - 0 if CPM detects cold boot, else -1 if possible warm boot.

Unpreserved Registers:

All but DS, BP, including SS.

18.4.20 CpuInit1 Routine

The **CpuInit1** function is called with routine linkage to perform the bulk of initialization of the CPU, particularly related to the initialization of high-integration CPU's DRAM controller, bus clocking, internal peripherals, and other functions.

Called from **BoardInit1**, this function typically loads the CPU with default operating values that can then be changed by the BPM's **BoardInit1** routine after the call to **CpuInit1** has returned.

This routine is entered with interrupts disabled and cannot enable them.

The implementor of routine **BoardInit1** may choose to handle initialization of the CPU in the BPM; consequently, this routine may not be called in some adaptations.

Input Parameters:

DS - BIOS data segment (40h).

Output Parameters:

CY - set if failure, else clear if success.
DS - BIOS data segment (40h).

Unpreserved Registers:

All but DS, BP, including SS.

18.4.21 CpuInitDma Routine

The **CpuInitDma** function is called with routine linkage to test and initialize the integrated DMA controller hardware on-board the CPU, and to return a status that indicates whether there are any failures in the hardware.

If no DMA controller hardware is available in the CPU, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_DMA_CPU** enabled, or if **OPTION_DMA_BOARD** is enabled and **BoardInitDmaCtrl** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.22 CpuInitIntCtrl Routine

The **CpuInitIntCtrl** function is called with routine linkage to test and initialize the integrated interrupt controller hardware on-board the CPU, and to return a status that indicates whether there are any failures in the hardware.

If no interrupt controller hardware is available in the CPU, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_INT_CPU** enabled, or if **OPTION_INT_BOARD** is enabled and **BoardInitIntCtrl** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.23 CpuInitParallel Routine

The **CpuInitParallel** function is called with routine linkage to test and initialize parallel I/O hardware on-board the CPU, and to return a status that indicates whether there are any failures in the hardware.

If no parallel port hardware is available in the CPU, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_PARALLEL_CPU** enabled.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.24 CpuInitRefresh Routine

The **CpuInitRefresh** function is called with routine linkage to test and initialize the integrated DRAM refresh controller hardware on-board the CPU, and to return a status that indicates whether there are any failures in the hardware.

If no refresh controller hardware is available in the CPU, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_REFRESH_CPU** is enabled; or **OPTION_REFRESH_BOARD** is enabled and routine **BoardInitRefresh** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.25 CpuInitSerBios Procedure

The **CpuInitSerBios** function is called with procedure linkage during POST to initialize the on-board serial port hardware, if available. CPUs such as 80186 and 80386-EX have on-board programmable serial ports that are initialized when the BIOS calls this function. This call occurs after the standard 8250-compatible UARTs have been initialized by POST, so that the standard UARTs have lower-numbered COM port assignments than the CPU UARTs.

If on-board serial port hardware is not available, this routine should return with carry clear. Otherwise, it should initialize the hardware and return with carry clear if the operation was successful, or carry set if the hardware could not be initialized.

This routine will be called if **OPTION_SUPPORT_SERIAL** and **OPTION_SERIAL_CPU** are enabled.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

18.4.26 CpuInitSerial Routine

The **CpuInitSerial** function is called with routine linkage to test and initialize asynchronous I/O hardware on-board the CPU, and to return a status that indicates whether there are any failures in the hardware.

If no asynchronous serial ports hardware is available in the CPU, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_SERIAL** and **OPTION_SERIAL_CPU** are enabled.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.27 CpuInitTimer Routine

The **CpuInitTimer** function is called with routine linkage to test and initialize (i.e., start running) any internal timers located within the CPU, and to return a status that indicates whether there are any failures in the timer hardware.

If no timers are available in the CPU, or if the adaptation engineer does not wish to test the timers, then this function should return with the carry flag clear. If timers are tested and initialize properly, then this routine returns with the carry flag clear if the timers are operational, or set if they fail the test.

This function should not make a decision about whether the CPU's timers will be used over external hardware. All policy decisions are made before this function is called in the core system BIOS.

This function may need to interact with the **CpuHookVectors** function, so that the appropriate timers can be routed to ISA-standard interrupt vectors, as on-board timers are usually hardwired to nonstandard interrupt assignments.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_TIMER_CPU** is enabled, or if **OPTION_TIMER_BOARD** is enabled and **BoardInitTimer** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.28 CpuInitWatchdog Routine

The **CpuInitWatchdog** function is called with routine linkage to test and initialize the watchdog timer hardware on-board the CPU, and to return a status that indicates whether there are any failures in the watchdog timer hardware.

If no watchdog timer is available in the CPU, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_WATCHDOG** is enabled; and then if **OPTION_WATCHDOG_CPU** is enabled, or if **OPTION_WATCHDOG_BOARD** is enabled and **BoardInitWatchdog** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.29 CpuKickWatchdog Procedure

The **CpuKickWatchdog** function is called with procedure linkage to restart the integrated watchdog timer controller hardware on-board the CPU, if it exists. Restarting causes the timer to be reloaded, so that it will take the entire expiration period for the timer to expire.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be restarted.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_WATCHDOG** is enabled; and then if **OPTION_WATCHDOG_CPU** is enabled, or if **OPTION_WATCHDOG_BOARD** is enabled and **BoardKickWatchdog** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

18.4.30 CpuPwrLvl Procedure

The **CpuPwrLvl** function is called with procedure linkage to notify the CPU of a change in power state in the system. This function is called by the Power Management Subsystem in the core BIOS, when the **Cpu** device is enabled in the **POWER_DEVID** device tree.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation.

This routine is called if **OPTION_SUPPORT_POWERMAN** is enabled; and then **OPTION_POWERMAN_CPU** is enabled and the Cpu device is in the device tree.

Input Parameters:

DS - segment of the extended BIOS data area (EBDA).
BX - device index of the CPU itself.
CL - new power level.
CH - old power level.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

DS, BX, CL, Flags.

18.4.31 CpuSerGetCh Procedure

The **CpuSerGetCh** function is called with procedure linkage to read a character from an on-board serial port, if CPU-integrated serial port hardware exists. If the specified amount of time passes before a character can be read from the serial port, then this routine returns failure by setting the CY flag.

If no hardware is available in the CPU, then this function should return with the carry flag set. If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation within the specified interval of time.

This routine will be called if **OPTION_SUPPORT_SERIAL** and **OPTION_SERIAL_CPU** are enabled.

Input Parameters:

DX - port to read from, as follows:

01h - first CPU port.
02h - second CPU port.
03h - third CPU port.
04h - fourth CPU port.

DI - timeout in 18.2 Hz ticks to use for the operation.

Output Parameters:

CY - set if failure, else clear if success.
AL - if success, character read from port.

Unpreserved Registers:

Flags.

18.4.32 CpuSerGetStatus Procedure

The **CpuSerGetStatus** function is called with procedure linkage to read the status of an on-board serial port, if CPU-integrated serial port hardware exists.

If no hardware is available in the CPU, then this function should return with the carry flag set. If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation.

This routine will be called if **OPTION_SUPPORT_SERIAL** and **OPTION_SERIAL_CPU** are enabled.

Input Parameters:

DX - port to read status from, as follows:

01h - first CPU port.
02h - second CPU port.
03h - third CPU port.
04h - fourth CPU port.

DI - timeout in 18.2 Hz ticks to use for the operation.

Output Parameters:

CY - set if failure, else clear if success.
AH - if success, line status register in 8250 format.
AL - if success, modem status register in 8250 format.

Unpreserved Registers:

Flags.

18.4.33 CpuSerInit Procedure

The **CpuSerInit** function is called with procedure linkage to initialize the communications parameters associated with an on-board serial port, if CPU-integrated serial port hardware exists.

If no hardware is available in the CPU, then this function should return with the carry flag set. If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation.

This routine will be called if **OPTION_SUPPORT_SERIAL** and **OPTION_SERIAL_CPU** are enabled.

Input Parameters:

DX - port to initialize, as follows:

01h - first CPU port.
02h - second CPU port.
03h - third CPU port.
04h - fourth CPU port.

AL - Serial port initialization parameters:

bbb00000b - Baud rate, as follows:

000b - 110 baud.
001b - 150 baud.
010b - 300 baud.
011b - 600 baud.
100b - 1200 baud.
101b - 2400 baud.

110b - 4800 baud.
111b - 9600 baud.

000pp000b - Parity, as follows:

00b - No parity.
01b - Odd parity.
10b - No parity.
11b - Even parity.

00000s00b - Stop bits, as follows:

0b - One stop bit.
1b - Two stop bits.

00000011b - Data bits, as follows:

10b - 7 data bits.
11b - 8 data bits.

DI - timeout in 18.2 Hz ticks to use for the operation.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

18.4.34 CpuSerInitExt Procedure

The **CpuSerInitExt** function is called with procedure linkage to perform an extended initialize with communications parameters associated with an on-board serial port, if CPU-integrated serial port hardware exists.

If no hardware is available in the CPU, then this function should return with the carry flag set. If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation.

This routine will be called if **OPTION_SUPPORT_SERIAL** and **OPTION_SERIAL_CPU** are enabled.

Input Parameters:

DX - port to initialize, as follows:

01h - first CPU port.
02h - second CPU port.
03h - third CPU port.

04h - fourth CPU port.

AL - 00h if no break signal, 01h if break signal.
BH - Parity, as follows:

00h - no parity.
01h - odd parity.
02h - even parity.
03h - stick parity odd.
04h - stick parity even.

BL - Stop bits, as follows:

00h - 1 stop bit.
01h - 2 stop bits if data length is 6, 7, or 8 bits.
02h - 1.5 stop bits if data length is 5 bits.

CH - Data length, as follows:

00h - 5 bits.
01h - 6 bits.
02h - 7 bits.
03h - 8 bits.

CL - Baud rate, as follows:

00h - 110 baud.
01h - 150 baud.
02h - 300 baud.
03h - 600 baud.
04h - 1200 baud.
05h - 2400 baud.
06h - 4800 baud.
07h - 9600 baud.
08h - 19.2 kbaud.
09h - 38.4 kbaud.
0ah - 56 kbaud.
0bh - 115 kbaud.

DI - timeout in 18.2 Hz ticks to use for the operation.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

18.4.35 CpuSerPutCh Procedure

The **CpuSerPutCh** function is called with procedure linkage to write a character to an on-board serial port, if CPU-integrated serial port hardware exists.

If no hardware is available in the CPU, then this function should return with the carry flag set. If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation within the specified interval of time.

This routine will be called if **OPTION_SUPPORT_SERIAL** and **OPTION_SERIAL_CPU** are enabled.

<u>Input Parameters:</u>

AL - character to write.
DX - port to write to, as follows:

01h - first CPU port.
02h - second CPU port.
03h - third CPU port.
04h - fourth CPU port.

DI - timeout in 18.2 Hz ticks to use for the operation.

<u>Output Parameters:</u>

CY - set if failure, else clear if success.

<u>Unpreserved Registers:</u>

Flags.

18.4.36 CpuSetFastSpeed Procedure

The **CpuSetFastSpeed** function is called with procedure linkage to switch the CPU clocking to the highest speed supported by the hardware, if speed-switching hardware exists.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be programmed.

This routine is called if **OPTION_SPEED_CPU** is enabled; or **OPTION_SPEED_BOARD** is enabled and **BoardSetFastSpeed** calls this routine.

<u>Input Parameters:</u>

None.

<u>Output Parameters:</u>

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

18.4.37 CpuSetSlowSpeed Procedure

The **CpuSetSlowSpeed** function is called with procedure linkage to switch the CPU clocking to the slowest speed supported by the hardware, if speed-switching hardware exists.

If no hardware is available in the CPU, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be programmed.

This routine is called if **OPTION_SPEED_CPU** is enabled; or **OPTION_SPEED_BOARD** is enabled and **BoardSetSlowSpeed** calls this routine.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

18.4.38 CpuStartDma Procedure

The **CpuStartDma** function is called with procedure linkage to program the CPU DMA controller to perform a DMA operation, if available. CPUs such as 80186 and 80386-EX have on-board programmable DMA controllers that are started when the BIOS calls this function.

If on-board DMA hardware is not available, this routine should return with carry set. Otherwise, it should perform the operation and return with carry clear if the operation was successful, or set if the DMA operation failed.

This function is not called to perform floppy disk I/O directly; instead, routine **BoardFloppyDma** is called by the core BIOS, which calls **CpuFloppyDma**, and that routine ultimately calls this one.

This routine is called if **OPTION_SUPPORT_FLOPPY** and **OPTION_FLOPPY_DMA** are enabled; and then if **OPTION_DMA_CPU** is enabled and **CpuFloppyDma** calls this routine, or **OPTION_DMA_BOARD** is enabled and **BoardStartDma** calls this routine.

Input Parameters:

DX:BX - 32-bit linear address of buffer.
CX - 16-bit byte count for transfer.
AH - DMA operation code, as follows:

00h - DMA to memory from I/O (read).
01h - DMA from memory to I/O (write/format).
02h - DMA from I/O, no memory (verify).
AL - DMA channel (0-3).

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

AX, BX, CX, DX, Flags.

18.4.39 CpuTestSyncIo Routine

The **CpuTestSyncIo** function is called with routine linkage to test the integrity of any synchronous I/O hardware on-board the CPU, and to return a status that indicates whether there are any failures in the hardware.

If no synchronous I/O unit is available in the CPU, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

18.4.40 CpuUnmaskInt Procedure

The **CpuUnmaskInt** function is called with procedure linkage to enable a specific interrupt level at the CPU interrupt controller. CPUs such as 80186 and 80386-EX have on-board programmable interrupt controllers that must be programmed to enable interrupts from specific devices, each assigned a level.

This routine will be called if **OPTION_INT_CPU** enabled, or if **OPTION_INT_BOARD** is enabled and **BoardUnmaskInt** calls this routine.

Input Parameters:

AL - interrupt vector level to initialize, as follows:

00h - IRQ0, or INT 08h.
01h - IRQ1, or INT 09h.
02h - IRQ2, or INT 0ah.
03h - IRQ3, or INT 0bh.
04h - IRQ4, or INT 0ch.
05h - IRQ5, or INT 0dh.
06h - IRQ6, or INT 0eh.
08h - IRQ7, or INT 0fh.

Output Parameters:

None.

Unpreserved Registers:

Flags.

Chapter 19

CHIPSET PERSONALITY MODULES

EMBEDDED BIOS has a flexible, scalable architecture that enables it to support a wide spectrum of chipsets, the hardware glue that binds the major building blocks in virtually every desktop PC design, and many embedded designs. Typically consisting of one or two large packages of VLSI, they may contain bus control logic, a few UARTs, DMA controllers, interrupt controllers, a counter-timer unit, DRAM refresh controller, floppy disk controller, and other functional items like wait state generators.

Commonly, chipsets consist of Northbridge and Southbridge components. The Northbridge component normally handles communication with high-bandwidth devices, such as DRAM, PCI and other high-performance busses, and bridges to other devices, such as AGP.

Southbridge components normally deal with slower legacy functions, such as the ISA bus, port 92h, as well as ISA legacy controllers such as the 8259s, 8237As, 8254s, and so on. Southbridge components may include a keyboard controller (8042), Floppy Disk Controller (FDC), and other controllers, although a given design may choose not to use portions of them so that an external Super I/O part might be used instead.

Some vendors have chosen to define “Eastbridge” and “Westbridge” functions, although these concepts are not well-established in the industry. You may find reference to these terms when data sheets talk about high speed connections between the processor or Northbridge and a second “glue” chip that performs other functions.

There are many chipset manufacturers, and each manufacturer provides a line of chipsets that is constantly being improved. Thus, some chipsets become dated and obsolete, and new ones become available with improved features or that respond to new market needs. There are no industry standards for chipset design or chipset software programming, and so the initialization and configuration of these devices is different for each chipset.

For this reason, EMBEDDED BIOS does not include the chipset initialization and configuration code in the core BIOS itself. Rather, the core BIOS contains call-outs to a special module, called a Chipset Personality Module (CSPM), that can be edited by the OEM to support any chipset.

The chipset for a given BIOS build is selected with the **CHIPSET** parameter in the project file. For systems with no chipsets, the **NOCHPSET** chipset is used. Note that some high-integration CPUs are more chipset-like than CPU-like. For example, AMD's SC300, SC310, SC400, SC410, and SC520 CPUs are all modeled as a standard 386 or 486 CPU core with chipset functionality bound in the same package.

This chapter describes how CSPMs fit into the EMBEDDED BIOS build, and the details of the CSPM interface called by the core EMBEDDED BIOS code that supports different chipsets.

19.1 How CSPM Override Routines Work

The EMBEDDED BIOS build links together many modules that implement BIOS services, hardware managers, and function redirectors. Additionally, the module `CHIPSET.ASM` in the `SYSTEM` directory is assembled and linked into the system build.

Actually, the CHIPSET.ASM module is a shell that uses assembly `INCLUDE` statements to include the CSPM selected with the **CHIPSET** parameter in the Project file. The `CHIPSET.ASM` module also contains default versions of the CSPM routines, should the CSPM only have a few routines defined. Thus, a CSPM may be comprised of zero, any, or all, of the routines documented later in this chapter. The routines defined in a CSPM are called *override routines*, and have the special "**OVERRIDE**" parameter in their **DefProc** or **DefRtn** procedure definition MACROs.

The **NOCHPSET** CSPM is an excellent example of this default system in action. The **NOCHPSET** CSPM contains no routines at all, so that when **NOCHPSET** is selected as the **CHIPSET** parameter value, all of the default routines are used.

On the other hand, the **TEMPLATE** CSPM is an excellent example of the reverse. This module is an example of a CSPM that has all of the routines defined, although they happen to contain exactly the same code that the default routines do. The intent of providing this CSPM with EMBEDDED BIOS is to enable the OEM to simply clone the **TEMPLATE** CSPM, change routines that are necessary, and then delete the ones that go unchanged. This facilitates rapid development of new CSPMs in the OEM environment.

19.2 How CSPMs are Packaged in Files

From a project-management point of view, a CSPM consists of at least two files; one is an .ASM file that contains the routines provided by the OEM that override the default CSPM functions, and the other is an .INC file that contains manifest constants, macro definitions, and other definitions needed by the CSPM. In some circumstances, the Board Personality Module (BPM) may need to use the definitions in the CSPM's .INC file, so this file should be constructed so that it does not define data storage areas where these areas would be duplicated in both modules.

A third file, CS1632.ASM, must be defined if **OPTION_SUPPORT_BIOS32** is to be enabled in project files for 32-bit BIOS support. When this option is enabled, then dual-build (16-bit and 32-bit) routines in the CSPM are assumed to be located in this file. When this option is not enabled, then they are found in the primary CSPM .ASM file. New CSPM implementations should always have this file.

All CSPMs have a name (from 1 to 8 filename characters), and the .ASM and .INC files (except for CS1632.ASM) must carry this name. For example, the NOCHPSET CSPM consists of a `NOCHPSET.ASM` file and a `NOCHPSET.INC` file.

Each CSPM's files are contained within a subdirectory of the CHIPSETS directory. The subdirectory must have the same name as the CSPM. For example, if the BIOS main directory is C:\BIOS43, then the **NOCHPSET** CSPM's files would be contained in the C:\BIOS43\CHIPSETS\NOCHPSET directory.

Additional source files may be located in the CSPM's subdirectory, but it is up to the OEM to define their contents and ensure that they are assembled as a part of the BIOS build. It is recommended that no core BIOS files be modified to include these sources; instead, it is a good idea to include them with INCLUDE statements in the CSPM's .ASM file.

19.3 Other Chipset Personality Modules

If you don't want to write your own personality module for a new chipset, it may actually already be available through General Software. For example, the Ali M1487, M6117, and M1541 chipsets are all supported with separate CSPMs available from General Software (contact General Software for more information).

19.4 The CSPM Interface

All CSPM implementations export the same set of functions callable from the core system BIOS. This section documents the functions that must be implemented in a CSPM.

Be careful to define the proper function type when creating new CSPMs. Some CSPM functions are written as coroutines, using the **DefRtn**/**EndRtn** macros. Others are written as procedures, using the **DefProc**/**EndProc** macros. If these are confused, there will be errors during the link of the system BIOS.

Unless otherwise specified, routines always return to their caller, and do not modify any register contents. The CPU flags may be destroyed by CSPM functions, except that the carry flag is normally used to indicate success if clear, or failure if false. Other flags, such as the zero flag, are destroyed at random by the functions. The direction flag, however, must be preserved.

Functions declared as coroutines cannot modify the BP CPU register, as it is used for return linkage. If they must use BP because they need to call another coroutine as a subroutine, then BP can be placed into another register to save the original return address.

Some functions are entered with the interrupt flag cleared (disallowing interrupts). These functions cannot enable interrupts for any reason, as they are used at times when interrupt management has not been established. Other functions are entered with a random interrupt flag as the context dictates. In this case, the routine can manipulate the interrupt flag if it wishes. Unless otherwise specified, a routine **may** modify the IF flag.

19.4.1 CsAssignPciIrq Procedure

The **CsAssignPciIrq** function is called with procedure linkage from routine **BoardAssignPciIrq** on behalf of the PCI Configuration Manager in the core BIOS to map a system IRQ level to a PCI interrupt line.

This function is only called in PCI-based systems, if **OPTION_SUPPORT_PCI** is enabled, and then only if **BoardAssignPciIrq** calls this routine.

This function must be implemented in the dual-build (16-bit/32-bit) file of the CSPM (CS1632.ASM) in order to support both the 16-bit and 32-bit PCI services.

Input Parameters:

AH - PCI Interrupt assignment: 0=A, 1=B, 2=C, 3=D.
AL - System IRQ Level (0-15, 16=disable).

Output Parameters:

CY - set if failure (i.e., interrupt line not available), else clear if success.

Unpreserved Registers:

Flags.

19.4.2 CsDisableA20 Procedure

The **CsDisableA20** function is called with procedure linkage to program the chipset to disable its A20 gate, if the chipset supports it.

This routine will be called if **OPTION_A20_CHIPSET** enabled, or if **OPTION_A20_BOARD** is enabled and **BoardDisableA20** calls this routine.

If no hardware exists, then this routine should return with carry set. If hardware exists and the A20 gate cannot be programmed, then the carry is set upon return. Otherwise, carry is cleared.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

19.4.3 CsDisableCache Procedure

The **CsDisableCache** function is called with procedure linkage to program the chipset to disable the external (L2) cache, if available.

This routine is called if **OPTION_SUPPORT_CACHE** is enabled; and **OPTION_CACHE_CHIPSET** is set, or **OPTION_CACHE_BOARD** is set and the **BoardDisableCache** function calls this routine.

If no cache is available, this routine should return with carry clear. If cache is available, this routine should return with carry clear if the cache could be disabled, or set if it could not be disabled.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

19.4.4 CsDisableShadow Procedure

The **CsDisableShadow** function is called with procedure linkage to program the chipset to disable all shadowing, so that the underlying ROMs are used instead.

This routine is called if **OPTION_SUPPORT_SHADOW** is enabled, and the **BoardDisableShadow** function calls this routine.

If no shadowing is available, this routine should return with carry clear. If shadowing hardware is available, this routine should return with carry clear if the shadow RAM could be disabled, or set if it could not be disabled.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

19.4.5 CsDisableWatchdog Procedure

The **CsDisableWatchdog** function is called with procedure linkage to stop the watchdog timer hardware supported by the chipset.

This routine is called if **OPTION_SUPPORT_WATCHDOG** is enabled; and **OPTION_WATCHDOG_CHIPSET** is set, or **OPTION_WATCHDOG_BOARD** is set and the **BoardDisableWatchdog** function calls this routine.

If no hardware exists, then this routine should return with carry set. If hardware exists and the watchdog timer cannot be programmed, then the carry is set upon return. Otherwise, carry is cleared.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

19.4.6 CsDisplayChipset Procedure

The **CsDisplayChipset** function is called with procedure linkage to display an OEM-defined message associated with the chipset implementation. Typically, this would state the chipset manufacturer's name, the chipset part number, and the date the Chipset Personality Module code was written.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

19.4.7 CsEnableA20 Procedure

The **CsEnableA20** function is called with procedure linkage to program the chipset to enable its A20 gate, if the chipset supports it.

This routine is called if **OPTION_A20_CHIPSET** is set, or **OPTION_A20_BOARD** is set and the **BoardEnableA20** function calls this routine.

If no hardware exists, then this routine should return with carry set. If hardware exists and the A20 gate cannot be programmed, then the carry is set upon return. Otherwise, carry is cleared.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

19.4.8 CsEnableApm Procedure

The **CsEnableApm** function is called during POST with procedure linkage to enable whatever APM functionality is available in the chipset.

This routine will be called if **OPTION_SUPPORT_APM** is enabled; and then if **OPTION_POWERMAN_CHIPSET** enabled.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

19.4.9 CsEnableCache Procedure

The **CsEnableCache** function is called with procedure linkage to program the chipset to enable the external cache, if available.

This routine is called if **OPTION_SUPPORT_CACHE** is enabled; and **OPTION_CACHE_CHIPSET** is set, or **OPTION_CACHE_BOARD** is set and the **BoardEnableCache** function calls this routine.

If no cache is available, this routine should return with carry clear. If cache is available, this routine should return with carry clear if the cache could be enabled, or set if it could not be enabled.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

19.4.10 CsEnableWatchdog Procedure

The **CsEnableWatchdog** function is called with procedure linkage to start the watchdog timer hardware supported by the chipset.

This routine is called if **OPTION_SUPPORT_WATCHDOG** is enabled; and **OPTION_WATCHDOG_CHIPSET** is set, or **OPTION_WATCHDOG_BOARD** is set and the **BoardEnableWatchdog** function calls this routine.

If no hardware exists, then this routine should return with carry set. If hardware exists and the watchdog cannot be programmed, then the carry is set upon return. Otherwise, carry is cleared.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

19.4.11 CsGetPciInfo Procedure

The **CsGetPciInfo** function is called with procedure linkage from routine **BoardGetPciInfo** on behalf of the PCI Configuration Manager in the core BIOS to return a bitmask in the BX CPU register of the system IRQ levels that are assigned for PCI use.

Only those interrupt levels supported by the chipset should be returned. If no interrupts are assignable by the chipset, the value 0000h should be returned.

This function is only called in PCI-based systems if **OPTION_SUPPORT_PCI** is enabled, and then only if **BoardGetPciInfo** calls this routine.

This function must be implemented in the dual-build (16-bit/32-bit) file of the CSPM (CS1632.ASM) in order to support both the 16-bit and 32-bit PCI services.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.
BX - bitmask of assignable IRQ levels (bits set-IRQs assignable).

Unpreserved Registers:

Flags.

19.4.12 CsInit0 Routine

The **CsInit0** function is called with routine linkage to perform very early initialization of the chipset, including participating in the decision about whether a warm boot or a cold boot has occurred.

Normally called from the BPM's **BoardInit0**, this function accepts as a parameter in the SP register a value of 0 if a cold boot has occurred, or -1 if the BPM has determined that a warm boot may have occurred. If this routine determines that a cold boot has occurred, then it sets SP to 0. Similarly, if it determines that a warm boot may have occurred, then it sets SP to -1. If it has no additional information to provide POST during the boot detection process, then it does not change SP.

A less-common function of this machine is the initialization of certain chipset functions that must be disabled prior to POST's commencement.

At the BPM implementor's discretion, this routine may be called at any time during **BoardInit0** processing, or not at all. See **BoardInit0** for details.

This routine is entered with interrupts disabled and cannot reenable them.

Input Parameters:

DS - BIOS data segment (40h).
SP - 0 if BPM detects cold boot, else -1 if possible warm boot.

Output Parameters:

DS - BIOS data segment (40h).
SP - 0 if CSPM detects cold boot, else -1 if possible warm boot.

Unpreserved Registers:

All but DS, SP, BP and SS.

19.4.13 CsInit1 Routine

The **CsInit1** function is called with routine linkage to perform very early initialization of the chipset, including participating in the decision about whether a warm boot or a cold boot has occurred.

The **CsInit1** function is called with routine linkage to perform the bulk of initialization of the chipset, particularly related to the initialization of the DRAM controller, bus clocking, internal peripherals, and other functions.

Called from **BoardInit1**, this function typically loads the chipset with default operating values that can then be changed by the BPM's **BoardInit1** routine after the call to **CsInit1** has returned.

At the BPM implementor's discretion, this routine may be called at any time during **BoardInit1** processing, or not at all. See **BoardInit1** for details.

While this function is used to program wait states, CPU clocking, and other parameters, DRAM geometry analysis may be performed if this is not convenient for handling in routine **CsMemConfig**.

Those chipset-like CPUs with on-board LCD controllers may need to have fonts loaded; this is an ideal time to perform this function as well.

This routine is entered with interrupts disabled and cannot reenable them.

Input Parameters:

DS - BIOS data segment (40h).
SP - 0 if BPM detects cold boot, else -1 if warm boot.

Output Parameters:

DS - BIOS data segment (40h).

Unpreserved Registers:

All but DS, SP, BP and SS.

19.4.14 CsInitRefresh Routine

The **CsInitRefresh** function is called with routine linkage to program the chipset to initialize the DRAM refresh controller in the chipset, if decoupled refresh is supported by the chipset.

This routine is called if **OPTION_REFRESH_CHIPSET** is enabled; or if **OPTION_REFRESH_BOARD** is enabled and **BoardInitRefresh** calls this routine.

If no hardware is available, this routine should return with carry clear. If available, this routine should return with carry clear if the refresh controller did not fail, or set if it failed.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but DS, BP and SS.

19.4.15 CsInitWatchdog Routine

The **CsInitWatchdog** function is called with routine linkage to program the chipset to initialize the watchdog timer in the chipset, if supported by the chipset. In this context, initialization means to stop any ongoing timing operation, and then start the watchdog timer running.

This routine is called if **OPTION_WATCHDOG_CHIPSET** is enabled; or if **OPTION_WATCHDOG_BOARD** is enabled and **BoardInitWatchdog** calls this routine.

If no hardware is available, this routine should return with carry clear. If available, this routine should return with carry clear if the watchdog timer did not fail, or set if it failed.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but DS, BP and SS.

19.4.16 CsKickWatchdog Routine

The **CsKickWatchdog** function is called with routine linkage to program the chipset to reset the watchdog timer and restart its timer, whether it was previously running or not.

This routine is called if **OPTION_WATCHDOG_CHIPSET** is enabled; or if **OPTION_WATCHDOG_BOARD** is enabled and **BoardKickWatchdog** calls this routine.

If no hardware is available, this routine should return with carry clear. If available, this routine should return with carry clear if the watchdog timer did not fail, or set if it failed.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but DS, BP and SS.

19.4.17 CsMapAddress Procedure

The **CsMapAddress** function is normally called by the BPM's **BoardMapAddress** with routine linkage to translate a 32-bit media address to either a real-mode windowed address or a 32-bit physical address.

If the chipset has memory mapping hardware (i.e., hardware EMS pages), then this routine should program the chipset to map the specified address into an EMS window, and return the translated address in real-mode 16:16 format, along with the number of bytes visible in the window at that location.

If the chipset has no MMU, then this routine should clear the CY flag, indicating that no windowing translation has occurred. Then, it should perform whatever modifications on the 32-bit media address are required to transform it into a physical address reachable with the CPU's protected mode.

Commonly, no such further translation is necessary, but a provision is made for translating the address if necessary. Note that some CPUs, notably the AMD SC400 and SC410, have programmable mapping of ROM chip select lines, and therefore an artificial media address architecture is invented by the CSPM to allow access to all ROMs in the system.

This routine is called if **OPTION_SUPPORT_MCL** is enabled, and then if **BoardMapAddress** calls this routine. The BPM's implementor may choose to handle the mapping of media addresses at the board level, so this routine may not be called in an actual adaptation.

If a direct physical mapping is to be performed, then the CY flag should be cleared on exit.

Input Parameters:

DX:AX - 32-bit media address (see Chapter 13 for details).

Output Parameters:

CY - set if result is 16:16 real mode, else clear if 32-bit physical address.

If real mode translation:

DX:AX - 16:16 real-mode translation of the input address.
CX - number of bytes visible at the address within the window.

If physical translation:

DX:AX - 32-bit physical address corresponding to the 32-bit media address.

Unpreserved Registers:

Flags.

19.4.18 CsMemConfig Routine

The **CsMemConfig** routine is normally called by the BPM's **BoardMemConfig** routine with routine linkage to program the chipset to determine how many memory banks are available, what size of SIMMs are in each bank, and how the banks are to be interleaved and positioned in the address space.

This routine is called after the chipset has been initialized and DRAM refresh has been activated. However, no stack yet exists because low memory (the base 64KB) has not been tested. Therefore, this routine is not called with a valid stack, but may create one temporarily in the bottom 64KB of memory in order to perform certain functions. This would be a rare case, since all DRAM configuration methods implicitly require manipulation of chipset registers to change the RAS and CAS assignments of banks of DRAM in the system, and changing this while the stack has valid data on it will cause the data to be mapped to different locations or be made unavailable.

It may be necessary to switch to protected mode temporarily during this routine's operation to test memory above 1MB. For sample code, consult routines **ToProt** and **ToReal** in HELPER.ASM. During extended memory testing, make sure that the A20 gate is properly enabled, or memory addresses will wrap to lower memory.

Typically, the algorithm by which the memory geometry is determined is provided by the chipset manufacturer. If you need help implementing this routine, consult the manufacturer for the recommended algorithm. Remember that you can either perform this work in this function, or in the **CsInit1** routine, although **CsInit1** would need to enable DRAM refresh if that were the case.

This routine is called if **BoardMemConfig** calls this routine. The BPM's implementor may choose to handle DRAM sizing at the board level, so this routine may not be called in an actual adaptation.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.
DI - if failure, POST_BEEP_CHIPSET.

Unpreserved Registers:

All but BP and SS.

18.4.19 CsPwrLvl Procedure

The **CsPwrLvl** function is called with procedure linkage to notify the chipset of a change in power state in the system. This function is called by the Power Management Subsystem in the core BIOS, when the **Cs** (chipset) device is enabled in the **POWER_DEVID** device tree.

If no hardware is available in the chipset, then this function should return with the carry flag clear.

If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation.

Input Parameters:

DS - segment of the extended BIOS data area (EBDA).
BX - device index of the chipset itself.
CL - new power level.
CH - old power level.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

DS, BX, CL, Flags.

19.4.20 CsReadReg Procedure

The **CsReadReg** function is called with procedure linkage by the debugger module to read a chipset configuration register.

Note that some chipsets have 16-bit-wide configuration registers, and others have 8-bit configuration registers. The interface supports both by providing a 16-bit data path.

This routine is not used unless **OPTION_SUPPORT_DEBUGGER** is enabled.

Input Parameters:

AX - Index of register to read.

Output Parameters:

CY - Set if failure, else clear if success.
AX - If success, contents of register.

Unpreserved Registers:

Flags.

19.4.21 CsReboot Procedure

The **CsReboot** function is called with coroutine linkage to handle a reboot request that originates from the debugger, the CTL-ALT-DEL mechanism, or other internal functions of the BIOS.

This routine is called if **OPTION_REBOOT_CHIPSET** is enabled; or if **OPTION_REBOOT_BOARD** is enabled and then if **BoardMapAddress** calls this routine. The BPM's implementor may choose to handle the reboot process at the board level, so this routine may not be called in an actual adaptation.

Input Parameters:

None.

Output Parameters:

None; should not return unless reboot cannot be performed.

Unpreserved Registers:

Flags.

19.4.22 CsSetFastSpeed Procedure

The **CsSetFastSpeed** function is called with procedure linkage to program the chipset to set the CPU clocking to the highest value, if clocking hardware controls are available.

This routine is called if **OPTION_SPEED_CHIPSET** is enabled; or if **OPTION_SPEED_BOARD** is enabled and then if **BoardSetFastSpeed** calls this routine. The BPM's implementor may choose to handle speed controls at the board level, so this routine may not be called in an actual adaptation.

If no hardware is available, this routine should return with carry clear. If available, this routine should return with carry clear if the speed could be changed, or set if it could not be changed.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

19.4.23 CsSetSlowSpeed Procedure

The **CsSetSlowSpeed** function is called with procedure linkage to program the chipset to set the CPU clocking to the lowest value (i.e., 4.77 Mhz or lower), if clocking hardware controls are available.

This routine is called if **OPTION_SPEED_CHIPSET** is enabled; or if **OPTION_SPEED_BOARD** is enabled and then if **BoardSetSlowSpeed** calls this routine. The BPM's implementor may choose to handle speed controls at the board level, so this routine may not be called in an actual adaptation.

If no hardware is available, this routine should return with carry clear. If available, this routine should return with carry clear if the speed could be changed, or set if it could not be changed.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

19.4.24 CsShadowArea Procedure

The **CsShadowArea** function is normally called by the BPM's **BoardShadowArea** function with procedure linkage to shadow an area of ROM using RAM remapped by the chipset. This routine must perform two operations to make this happen.

First, it must program the chipset to map pieces of shadow RAM to the address assigned to the ROM. This is done with chipset register manipulations.

Next, this routine must copy the contents of the ROM into the shadow RAM. This is accomplished in either of two ways. Some chipsets support a special "copy mode" that causes reads to occur from ROM at a given address, but writes to be written to RAM at the same address. If this mode is supported, it can be used by the **CsShadowArea** routine to perform a copy in place. Then, when the mode is disabled, both reads and writes are directed to the shadow RAM.

If the chipset does not support the special "copy mode" feature, then the shadowing can be accomplished by copying the 16KB ROM contents to a temporary location in low memory, then switching shadow RAM for the ROM, and copying the data back to RAM. Note that this technique must be used with care when shadowing the system BIOS itself, since the processor is executing instructions from the area of shadow RAM that has not been initialized until after the copy. To solve this problem, this routine should copy its code to low memory and run the rest of itself there until the shadow RAM contents are valid; then control can resume from RAM.

This routine is called if **OPTION_SUPPORT_SHADOW** is enabled, and at least one region of ROM is to be shadowed; and then if **BoardShadowArea** calls this routine. The BPM's implementor may choose to handle shadowing at the board level, so this routine may not be called in an actual adaptation.

Input Parameters:

DI - index of area to shadow, as follows:

0000h - segment C000h.
0001h - segment C400h.
0002h - segment C800h.
0003h - segment CC00h.
0004h - segment D000h.
0005h - segment D400h.
0006h - segment D800h.
0007h - segment DC00h.
0008h - segment E000h.
0009h - segment E400h.
000ah - segment E800h.
000bh - segment EC00h.
000ch - segment F000h.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

19.4.25 CsShadowWriteCtl Procedure

The **CsShadowWriteCtl** function is normally called by the BPM's **BoardShadowWriteCtl** function with procedure linkage to enable or disable write protection for the entire system. This function is normally used by the PCI option ROM shadowing support in the core BIOS.

If the chipset uses write protect bits in the hardware to remember which areas have been shadowed, then these bits may be saved/restored using some location in the extended BIOS data area.

This routine is called if **OPTION_SUPPORT_SHADOW** is enabled, and then if **BoardShadowWriteCtl** calls this routine. The BPM's implementor may choose to handle shadowing at the board level, so this routine may not be called in an actual adaptation.

Input Parameters:

AX - 0 to write-protect shadow RAM, or 1 to write-enable shadow RAM.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

19.4.26 CsTimerTick Procedure

The **CsTimerTick** function is called with procedure linkage from the BPM's BoardTimerTick function, so that the CSPM has a way to receive notification that timer ticks have occurred.

The OEM or chipset adaptation engineer may use this routine for any purpose that helps to implement other CSPM functions, such as timing out certain requests. The OEM may also want to use the routine for proprietary value-added functions, such as the regular polling of hardware in the background, for example.

Commonly, this routine is coded to interact with other routines in the CSPM through memory fields in the Extended BIOS Data Area (EBDA), and most commonly, in the **CsData** byte array that is reserved for CSPM use (see `DATA.INC` for layout of the EBDA).

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

19.4.27 CsUnMapAddress Procedure

The **CsUnMapAddress** function is called with procedure linkage by the BPM's **BoardUnMapAddress** routine on behalf of MCL when it has completed calling an MTD for a requested function, so that the chipset's mapping registers (if any), may be restored to their initial values before the **CsMapAddress** function changed them.

It is the responsibility of the CSPM to keep track of whether any chipset-level hardware registers have been programmed, perhaps with a RAM variable in the EBDA's **CsData** field, so that the **CsUnMapAddress** procedure can determine if it should perform any work at all.

This routine is not used unless **OPTION_SUPPORT_MCL** is enabled.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

19.4.28 CsWriteReg Procedure

The **CsWriteReg** function is called with procedure linkage by the debugger module to write a chipset configuration register with a specific value.

Note that some chipsets have 16-bit-wide configuration registers, and others have 8-bit configuration registers. The interface supports both by providing a 16-bit data path.

This routine is not used unless **OPTION_SUPPORT_DEBUGGER** is enabled.

Input Parameters:

AX - Index of register to write.
DX - Data to write to register.

Output Parameters:

CY - Set if failure, else clear if success.

Unpreserved Registers:

Flags.

Chapter 20

BOARD PERSONALITY MODULES

EMBEDDED BIOS contains an architected interface that allows it to work with OEM-written code that initializes and controls OEM-specific hardware glue, collectively referred to as the board, in a custom design.

Consider that many issues, including A20 gate control, shadowing, cache management, floppy DMA channel assignments, PCI interrupt structure, and Super I/O programming to name a few, must be customizable at the board level by the OEM. For example, while one design may gate A20 with the chipset or port 92h, another may use the keyboard controller and some other proprietary I/O port mechanism. EMBEDDED BIOS provides a way to handle these designs and all the other possible ones, without requiring the OEM to write code for reference designs, such as the commonly-available evaluation boards from silicon vendors.

The EMBEDDED BIOS core does not contain any board-specific code that deals with board-level issues, but instead contains call-outs to a special module, called a Board Personality Module (BPM), that contains the board-specific code.

The specific board for a given BIOS build is selected with the **BOARD** parameter in the project file. For systems with unremarkable boards (i.e., boards without any glue logic whatsoever), the **NOBOARD** board name is used.

This chapter describes how BPMs fit into the EMBEDDED BIOS build, and the details of the BPM interface called by the core EMBEDDED BIOS code that supports different designs.

The routines in the BPM call CPM and CSPM routines to perform much of their work by default. For more information about the CPM and CSPM helpers, see Chapters 18 and 19, respectively.

20.1 How BPM Override Routines Work

The EMBEDDED BIOS build links together many modules that implement BIOS services, hardware managers, and function redirectors. Additionally, the module `BOARD.ASM` in the `SYSTEM` directory is assembled and linked into the system build.

Actually, the BOARD.ASM module is a shell that uses assembly INCLUDE statements to include the CPM selected with the **BOARD** parameter in the Project file. The BOARD.ASM module also contains default versions of the BPM routines, should the BPM only have a few routines defined. Thus, a BPM may be comprised of zero, any, or all, of the routines documented later in this chapter. The routines defined in a BPM are called *override routines*, and have the special "**OVERRIDE**" parameter in their **DefProc** or **DefRtn** procedure definition MACROs.

The **NOBOARD** BPM is an excellent example of this default system in action. The **NOBOARD** BPM contains no routines at all, so that when **NOBOARD** is selected as the **BOARD** parameter value, all of the default routines are used.

On the other hand, the **TEMPLATE** BPM is an excellent example of the reverse. This module is an example of a BPM that has all of the routines defined, although they happen to contain exactly the same code that the default routines do. The intent of providing this BPM with EMBEDDED BIOS is to enable the OEM to simply clone the **TEMPLATE** BPM, change routines that are necessary, and then delete the ones that go unchanged. This facilitates rapid development of new BPMs in the OEM environment.

20.2 How BPMs are Packaged in Files

From a project-management point of view, a BPM consists of at least two files; one is an .ASM file that contains the routines provided by the OEM that override the default BPM functions, and the other is an .INC file that contains manifest constants, macro definitions, and other definitions needed by the BPM.

A third file, BPM1632.ASM, must be defined if **OPTION_SUPPORT_BIOS32** is to be enabled in project files for 32-bit BIOS support. When this option is enabled, then dual-build (16-bit and 32-bit) routines in the BPM are assumed to be located in this file. When this option is not enabled, then they are found in the primary BPM .ASM file. New BPM implementations should always have this file.

All BPMs have a name (from 1 to 8 filename characters), and the .ASM and .INC files (except for BPM1632.ASM) must carry this name. For example, the NOBOARD BPM consists of a NOBOARD.ASM file and a NOBOARD.INC file.

Each BPM's files are contained within a subdirectory of the BOARDS directory. The subdirectory must have the same name as the BPM. For example, if the BIOS main directory is C:\BIOS43, then the **NOBOARD** BPM's files would be contained in the C:\BIOS43\BOARDS\NOBOARD directory.

Additional source files may be located in the BPM's subdirectory, but it is up to the OEM to define their contents and ensure that they are assembled as a part of the BIOS build. It is recommended that no core BIOS files be modified to include these sources; instead, it is a good idea to include them with INCLUDE statements in the BPM's .ASM file.

20.3 Other Board Personality Modules

If you don't want to write your own personality module for a new board design, it may actually already be available through General Software. Most standard reference designs provided by the silicon vendors have BPMs available from General Software or a Technology Center. Contact General Software for more information.

20.4 The BPM Interface

All BPM implementations export the same set of functions callable from the core system BIOS. This section documents the functions that must be implemented in a BPM.

More often than not, the default board routines call CPM or CSPM functions to perform similar functions. For example, the **BoardSetFastSpeed** function calls the **CpuSetFastSpeed** and/or **CsSetFastSpeed** functions, depending on whether **OPTION_SPEED_CPU** and/or **OPTION_SPEED_CHIPSET** are enabled. If **OPTION_SPEED_BOARD** is not enabled, then the core BIOS doesn't call the board-level routine, but instead calls the CPU or chipset-level routine directly. This sounds complex, until it is understood that the system is designed to do the right thing for a generic system if all the defaults are used, and only when proprietary routines are added does it become necessary to select options that define how the core BIOS should use the new code. In practice, OEMs usually start with a BPM for a standard reference design, and slowly metamorphise it into their design's BPM, making the job easier.

Be careful to define the proper function type when creating new BPMs. Some BPM functions are written as coroutines, using the **DefRtn/EndRtn** macros. Others are written as procedures, using the **DefProc**/**EndProc** macros. If these are confused, there will be errors during the link of the system BIOS.

Unless otherwise specified, routines always return to their caller, and do not modify any register contents. The CPU flags may be destroyed by BPM functions, except that the carry flag is normally used to indicate success if clear, or failure if false. Other flags, such as the zero flag, are destroyed at random by the functions. The direction flag, however, must be preserved.

Functions declared as coroutines cannot modify the BP CPU register, as it is used for return linkage. If they must use BP because they need to call another coroutine as a subroutine, then BP can be placed into another register to save the original return address.

Some functions are entered with the interrupt flag cleared (disallowing interrupts). These functions cannot enable interrupts for any reason, as they are used at times when interrupt management has not been established. Other functions are entered with a random interrupt flag as the context dictates. In this case, the routine can manipulate the interrupt flag if it wishes. Unless otherwise specified, a routine **may** modify the IF flag, but must restore it to its value at the time the routine was called.

20.4.1 BoardApmMode Procedure

The **BoardApmMode** function is called with procedure linkage from the APM request router to handle a mode change request or to get status or event information.

This routine will be called if **OPTION_SUPPORT_APM** is enabled and requests are received by the APM request router from the operating system, application software, or other components of the core BIOS.

Input Parameters:

DL - command code, as follows:

00h - Transition system to READY state.

01h - Transition system to STANDBY state.
02h - Transition system to SUSPEND state.
03h - Transition system to OFF state.
feh - get event information (returns event code in DH).
ffh - get status information (returns last return code in DH).

Output Parameters:

CY - set if failure, else clear if success.
All others - as set by APM functions (see Chapter 21 for details).

Unpreserved Registers:

Flags.

20.4.2 BoardAssignPciIrq Procedure

The **BoardAssignPciIrq** function is called with procedure linkage from the PCI Configuration Manager in the core BIOS to map a system IRQ level to a PCI interrupt line.

This routine will be called if **OPTION_SUPPORT_PCI** is enabled. Normally, this routine calls **CsAssignPciIrq** to route the request to the chipset module.

This function must be implemented in the dual-build (16-bit/32-bit) file of the BPM (BPM1632.ASM) in order to support both the 16-bit and 32-bit PCI services.

Input Parameters:

AH - PCI Interrupt assignment: 0=A, 1=B, 2=C, 3=D.
AL - System IRQ Level (0-15, 16=disable).

Output Parameters:

CY - set if failure (i.e., interrupt line not available), else clear if success.

Unpreserved Registers:

Flags.

20.4.3 BoardAutoRedirect Procedure

The **BoardAutoRedirect** function is called with procedure linkage from POST to allow the BPM a chance to detect video devices and cancel console redirection if any other devices are detected.

The default routine checks for a VGA BIOS ROM at segment **CONFIG_VIDEO_ROM_SCAN** (usually, set to C000h), and if found, any redirection established for the console is reset to the primary keyboard and screen.

This policy can be overridden by the OEM by redefining this routine in the BPM.

This routine is only called, and automatic redirection is only enabled, if **OPTION_CON_REDIR_AUTO** and **OPTION_SUPPORT_CON_REDIRECTOR** are both enabled. The normal usage of this feature calls for enabling the above parameters, then setting **CONFIG_CON_REDIR_STD**, **CONFIG_CON_REDIR_DEBUG**, and **CONFIG_CON_REDIR_SETUP** to nonzero values specifying the default COM port to use for console redirection, for those components.

Input Parameters:

None.

Output Parameters:

CY – set if failure, else clear if success.

Unpreserved Registers:

Flags.

20.4.4 BoardBeep Routine

The **BoardBeep** function is called with routine linkage to cause the speaker to start or stop beeping.

This routine will be called if **OPTION_SUPPORT_SOUND** and **OPTION_SOUND_BOARD** are both enabled. Normally, this routine calls **CpuBeep** or **CsBeep** to route the request to the CPU or chipset modules, respectively.

Input Parameters:

AL - 1 to enable tone, 0 to disable tone.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

AX, CX, DX, Flags.

20.4.5 BoardDelayUsec Routine

The **BoardDelayUsec** function is called with routine linkage to cause a microdelay to occur, so that core BIOS functions are able to time device operations that are very short.

The default implementation of this routine works as follows: If an 8254 counter/timer controller is present in the system, then this routine uses it to perform the delay. Otherwise, if Port B is available, the refresh toggle bit is used. Lastly, if neither of these options are available, then a CPU spin loop is executed, with the **CPU_MHZ** parameter used as a factor in the loop counter.

The OEM can redefine this routine to change the way time is measured in the system, in the event that these methods are not adequate for a given design, or if they produce incorrect results because the design uses different clocking or timing.

This routine is called with interrupts disabled, and it must not enabled interrupts for any length of time.

Input Parameters:

DX - Number of microseconds to delay, from 0-50,000.

Output Parameters:

None.

Unpreserved Registers:

AX, BX, CX, DX, SI, DI, Flags.

20.4.6 BoardDisableA20 Hybrid

The **BoardDisableA20** function is called with hybrid (dual) linkage to disable the A20 line gate hardware on-board the CPU, if it exists. Normally, the A20 gate on 80286 and above CPUs is provided by external components, such as the 8042 keyboard controller, port 92h, or the chipset.

If the function cannot be performed, then this function should return with the carry flag set; otherwise, clear.

This routine is entered with interrupts disabled and cannot reenable them. It can also be called with procedure linkage or routine linkage, since it is a hybrid function. Be careful not to modify BP or use any RAM while inside this function, as it may be called during a period of time where DRAM is inoperative. Thus, this routine cannot use any PUSH, POP, CALL, RET, INT, or IRET instructions.

This routine will be called if **OPTION_A20_BOARD** is enabled. Normally, this routine calls **CpuDisableA20** or **CsDisableA20** to route the request to the CPU or chipset modules, respectively.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

20.4.7 BoardDisableCache Procedure

The **BoardDisableCache** function is called with procedure linkage to disable all L2 cache memory in the system. This routine explicitly does not disable the L1 (CPU) cache.

If the function cannot be performed, then this function should return with the carry flag set, else clear.

This routine will be called if **OPTION_SUPPORT_CACHE** and **OPTION_CACHE_BOARD** are enabled. Normally, this routine calls **CsEnableCache** to route the request to the chipset module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

20.4.8 BoardDisableDmaCtrl Routine

The **BoardDisableDmaCtrl** function is called with routine linkage to disable DMA controller hardware in the system. Disabling means to reset the controller so that no DMA processes are running after the routine returns to its caller.

If the function cannot be performed, then this function should return with the carry flag set, else clear.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_DMA_BOARD** is enabled. Normally, this routine calls **CpuDisableDma** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

20.4.9 BoardDisableIntCtrl Hybrid

The **BoardDisableIntCtrl** function is called with hybrid (dual) linkage to disable the interrupt controller hardware in the system. Disabling in this context means to cause the interrupt control to reset to the condition where there are no pending interrupts to be serviced, and no interrupt levels enabled (or unmasked) upon return from this routine.

If the function cannot be performed, then this routine should return with the carry flag set, else clear.

This routine is entered with interrupts disabled and cannot reenable them. It can also be called with procedure linkage or routine linkage, since it is a hybrid function. Be careful not to modify BP or use any RAM while inside this function, as it may be called during a period of time where DRAM is inoperative. Thus, this routine cannot use any PUSH, POP, CALL, RET, INT, or IRET instructions.

This routine will be called if **OPTION_INT_BOARD** is enabled. Normally, this routine calls **CpuDisableIntCtrl** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

20.4.10 BoardDisableShadow Procedure

The **BoardDisableShadow** function is called with procedure linkage to disable all shadowing in the system, so that the underlying ROMs are used instead.

This routine will be called if **OPTION_SUPPORT_SHADOW** is enabled. Normally, this routine calls **CsDisableShadow** to route the request to the chipset module.

If no shadowing is available, this routine should return with carry clear. If shadowing hardware is available, this routine should return with carry clear if the shadow RAM could be disabled, or set if it could not be disabled.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

20.4.11 BoardDisableTestMode Procedure

The **BoardDisableTestMode** function is called with procedure linkage immediately upon entry to Manufacturing Mode, to reset any software condition used by BoardTestMode to enter the mode.

It is possible for the implementor of the **BoardTestMode** routine to cause Manufacturing Mode to be entered if the hardware signal is present, or if a software-programmable register is set to a specific value. This allows a target to be booted, and with the debugger, the register can be set by the OEM in the lab. Upon rebooting the system, Manufacturing Mode is then entered. Immediately upon entry into Manufacturing Mode, the system calls **BoardDisableTestMode**, which allows the BPM implementor to reset this software-programmable register so that the system does not continually enter Manufacturing Mode thereafter.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

20.4.12 BoardDisableWatchdog Procedure

The **BoardDisableWatchdog** function is called with procedure linkage to disable the watchdog timer controller hardware in the system. Disabling in this context means to cause the watchdog timer to stop running, so that it will not possibly expire without restarting it.

If the function cannot be performed, then this routine should return with the carry flag set; else clear.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_WATCHDOG** and **OPTION_WATCHDOG_BOARD** are enabled. Normally, this routine calls **CpuDisableWatchdog** or **CsDisableWatchdog** to route the request to the CPU or chipset modules, respectively.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

20.4.13 BoardDisableWrites Procedure

The **BoardDisableWrites** function is called with procedure linkage by the Media Control Layer (MCL) to disable any programming voltage (Vpp) and enable any write protect signals associated with Flash or other similar components as a measure to save power and preserve integrity of the device(s).

This routine allows the core BIOS to support arbitrarily-complex methods used by OEM hardware for performing these functions without modifying the core BIOS itself.

While MCL has a delayed Vpp disable feature, this is embodied solely in MCL itself. When this routine is called, Vpp should be disabled immediately, as the delay has already occurred.

This routine will be called if **OPTION_SUPPORT_MCL** is enabled.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

None, not even flags.

20.4.14 BoardEnableA20 Hybrid

The **BoardEnableA20** function is called with hybrid (dual) linkage to enable the A20 line gate hardware in the system; normally, a chipset or high-integration CPU function.

If the function cannot be performed, then this routine should return with the carry flag set; else clear.

This routine is entered with interrupts disabled and cannot reenable them. It can also be called with procedure linkage or routine linkage, since it is a hybrid function. Be careful not to modify

BP or use any RAM while inside this function, as it may be called during a period of time where DRAM is inoperative. Thus, this routine cannot use any PUSH, POP, CALL, RET, INT, or IRET instructions.

This routine will be called if **OPTION_A20_BOARD** is enabled. Normally, this routine calls **CpuEnableA20** or **CsEnableA20** to route the request to the CPU or chipset modules, respectively.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

20.4.15 BoardEnableApm Procedure

The **BoardEnableApm** function is called with procedure linkage to initialize any APM hardware or data structures managed by the BPM. This hardware and any data fields may be used by the **BoardApmMode** function to satisfy APM requests.

This routine will be called if **OPTION_SUPPORT_APM** option is enabled.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

20.4.16 BoardEnableCache Procedure

The **BoardEnableCache** function is called with procedure linkage to enable all L2 caches in the system. This routine specifically does not enable the CPU (L1) cache.

If the function cannot be performed, then this routine should return with the carry flag set; else clear.

This routine will be called if **OPTION_SUPPORT_CACHE** and **OPTION_CACHE_BOARD** are enabled. Normally, this routine calls **CsEnableCache** to route the request to the chipset module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All.

20.4.17 BoardEnableDmaCtrl Routine

The **BoardEnableDmaCtrl** function is called with routine linkage to enable the DMA controller hardware in the system. Enabling means to cause the DMA controller to be ready to accept a programming sequence for a DMA operation. This routine is provided for symmetry with **BoardDisableDmaCtrl**, and is rarely used.

If the function cannot be performed, then this routine should return with the carry flag set; else clear.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_DMA_BOARD** is enabled. Normally, this routine calls **CpuEnableDma** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

20.4.18 BoardEnableIntCtrl Hybrid

The **BoardEnableIntCtrl** function is called with hybrid (dual) linkage to enable the interrupt controller hardware in the system. Enabling in this context means to cause the interrupt controller to be ready to receive unmask or EOI commands and handle interrupts from that point forward. This routine is rarely used, but is provided for symmetry with **BoardDisableIntCtrl**.

If the function cannot be performed, then this routine should return with the carry flag set; else clear.

This routine is entered with interrupts disabled and cannot reenable them. It can also be called with procedure linkage or routine linkage, since it is a hybrid function. Be careful not to modify BP or use any RAM while inside this function, as it may be called during a period of time where DRAM is inoperative. Thus, this routine cannot use any PUSH, POP, CALL, RET, INT, or IRET instructions.

This routine will be called if **OPTION_INT_BOARD** is enabled. Normally, this routine calls **CpuEnableIntCtrl** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

20.4.19 BoardEnablePciRegion Procedure

The **BoardEnablePciRegion** function is called with procedure linkage by the core PCI subsystem when shadowing PCI option ROMs to give the BPM the chance to program chipset registers that would mark selective regions as mapped for PCI purposes.

Not all chipsets require this work, but some do; it is up to the BIOS adaptation engineer to determine whether this requirement exists, and if so, to implement this routine appropriately.

The PCI subsystem automatically calls this routine before calling **BoardShadowArea** to create shadow memory for the region to be used for shadowing the PCI option ROM; therefore, this routine does not perform the tasks that **BoardShadowArea** is specified to perform.

This routine will be called if **OPTION_SUPPORT_PCI** is enabled.

Input Parameters:

DI - index of area to be mapped for PCI access, as follows:

0000h - segment C000h.
0001h - segment C400h.
0002h - segment C800h.
0003h - segment CC00h.
0004h - segment D000h.
0005h - segment D400h.

0006h - segment D800h.
0007h - segment DC00h.
0008h - segment E000h.
0009h - segment E400h.
000ah - segment E800h.
000bh - segment EC00h.
000ch - segment F000h.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

20.4.20 BoardEnableWatchdog Procedure

The **BoardEnableWatchdog** function is called with procedure linkage to enable the integrated watchdog timer controller hardware in the system. Enabling in this context means to start the watchdog timer running, so that if the timer is not reset within one expiration period, the timer will expire.

If the function cannot be performed, then this routine should return with the carry flag set; else clear.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_WATCHDOG** and **OPTION_WATCHDOG_BOARD** are enabled. Normally, this routine calls **CpuEnableWatchdog** or **CsEnableWatchdog** to route the request to the CPU or chipset modules, respectively.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

20.4.21 BoardEnableWrites Procedure

The **BoardEnableWrites** function is called with procedure linkage by the Media Control Layer (MCL) to enable any programming voltage (Vpp) and disable any write protect signals

associated with Flash or other similar components so that the device(s) can be written, erased, locked, or otherwise programmed.

This routine allows the core BIOS to support arbitrarily-complex methods used by OEM hardware for performing these functions without modifying the core BIOS itself.

While MCL has a delayed Vpp disable feature, this is embodied solely in MCL itself. When this routine is called, Vpp should be enabled immediately, and this routine must delay until Vpp has stabilized sufficiently to provide ample current to the device(s).

This routine will be called if **OPTION_SUPPORT_MCL** is enabled.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

None, not even flags.

20.4.22 BoardEoi Procedure

The **BoardEoi** function is called with procedure linkage to issue an "end-of-interrupt" command to the board's interrupt controller.

This routine will be called if **OPTION_INT_BOARD** is enabled. Normally, this routine calls **CpuEoi** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

None, NOT EVEN FLAGS.

20.4.23 BoardFloppyDma Procedure

The **BoardFloppyDma** function is called with procedure linkage to program the DMA controller to perform a DMA operation for floppy disk I/O. Typically, this routine calls **CpuFloppyDma**, although other actions, such coordinating activities between a Super I/O package and a chipset, may need to be handled here.

If on-board DMA hardware is not available, this routine should return with carry set. Otherwise, it should perform the operation and return with carry clear if the operation was successful, or set if the DMA operation failed.

This routine will be called if **OPTION_SUPPORT_FLOPPY**, **OPTION_FLOPPY_DMA**, and **OPTION_DMA_BOARD** are enabled. Normally, this routine calls **CpuFloppyDma** to route the request to the CPU module.

Input Parameters:

DX:BX - 32-bit linear address of buffer.
CX - 16-bit byte count for transfer.
AH - DMA operation code, as follows:

00h - DMA to memory from I/O (read).
01h - DMA from memory to I/O (write/format).
02h - DMA from I/O, no memory (verify).

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

AX, BX, CX, DX, Flags.

20.4.24 BoardFsInit Procedure

The **BoardFsInit** function is called with procedure linkage by the File System Control Layer (FSCL) during POST to give the BPM the chance to intervene on the initialization of a file system.

Normally, FSCL makes FsInit calls to all of the enabled file systems defined in the **FILE_SYSTEM** table in the project file. However, the board-level design may require that certain file system definitions be modified, such as the size and location of ROM, RAM, or Flash memory to be used by a certain file system.

The mechanism that the board module uses to determine the availability of ROM, RAM, Flash, or other media in the system is OEM-specific. For example, a jumper could be read from an I/O port or an 8042 pin. Or, a memory test could be performed to verify the existence of the memory in the address space. For applications where a file system engages a host machine via protocol over a transmission media such as Ethernet or RS-232, the board module may use this opportunity to initialize any OEM-specific hardware that allows the transmission to take place.

This function gets called on every FsInit call. This means the board module may need to distinguish between file systems by inspecting the **FS_UNIT** and **FS_PACKET** data structures. For information about the organization of these structures, and the FSCL in general, see Chapter 12.

This entrypoint obsoletes the **BoardInitAppRom** function; use of the latter is discouraged.

Input Parameters:

AH - Major function code (**FSCALL_INIT**).
SI - Subfunction code (**FSCALL_HARD** or **FSCALL_SOFT**).
DX:CX - 0:0 (logical block address for a possible read of the boot record).
DS:DI - 16:16 segment offset pointer to the **FS_UNIT** structure for this file system.
SS:BP - 16:16 segment offset pointer to the **FS_PACKET** structure for this file system.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

20.4.25 BoardGetPciInfo Procedure

The **BoardGetPciInfo** function is called with procedure linkage from the PCI Configuration Manager in the core BIOS to return a bitmask in the BX CPU register of the system IRQ levels that are assigned for PCI use.

Only those interrupt levels supported by the chipset should be returned. If no interrupts are assignable by the chipset, the value 0000h should be returned.

This routine will be called if **OPTION_SUPPORT_PCI** is enabled. Normally, this routine calls **CsGetPciInfo** to route the request to the chipset module.

This function must be implemented in the dual-build (16-bit/32-bit) file of the BPM (BPM1632.ASM) in order to support both the 16-bit and 32-bit PCI services.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.
BX - bitmask of assignable IRQ levels (bits set-IRQs assignable).

Unpreserved Registers:

Flags.

20.4.26 BoardHelp1 Procedure

The **BoardHelp1** function is called with procedure linkage from certain implementations of CPM or CSPM functions, to perform board-level functions for the CPM or CSPM. Generally, BPM routines call down to CPM and CSPM functions, but occasionally, there is a need for CPM

and CSPM functions to access code specific to a board's design. The purpose of this routine (and **BoardHelp2**) is to provide an unarchitected entrypoint that is always present, that can perform whatever functions required by the OEM.

An example of the use of this unarchitected entrypoint is in the implementation of the AMD SC300 CSPM, where the CSPM must become aware of the board's use of the LCD controller before making a decision about how to program the LCD controller.

The core BIOS does not call this function. It is reserved for the OEM.

Input Parameters:

Unarchitected.

Output Parameters:

Unarchitected.

Unpreserved Registers:

Unarchitected.

20.4.27 BoardHelp2 Procedure

The **BoardHelp2** function is called with procedure linkage from certain implementations of CPM or CSPM functions, to perform board-level functions for the CPM or CSPM. Generally, BPM routines call down to CPM and CSPM functions, but occasionally, there is a need for CPM and CSPM functions to access code specific to a board's design. The purpose of this routine (and **BoardHelp1**) is to provide an unarchitected entrypoint that is always present, that can perform whatever functions required by the OEM.

The core BIOS does not call this function. It is reserved for the OEM.

Input Parameters:

Unarchitected.

Output Parameters:

Unarchitected.

Unpreserved Registers:

Unarchitected.

20.4.28 BoardIdeAutoDetect Procedure

The **BoardAutoDetectIde** function is called with procedure linkage from the IDE/ATA file system driver to allow the BPM to perform custom IDE autodetection and set board/chipset parameters as necessary to accommodate faster data transfer rates.

Input Parameters:

ES:DI – 16:16 pointer to IDE drive identification table.
AX – Sectors per cylinder from FSD's autodetection.
BX – Sectors per track from FSD's autodetection.
CX – Number of heads from FSD's autodetection.
DX – Number of cylinders from FSD's autodetection.

Output Parameters:

AX – Sectors per cylinder as modified by BPM.
BX – Sectors per track as modified by BPM.
CX – Number of heads as modified by BPM.
DX – Number of cylinders as modified by BPM.

Unpreserved Registers:

Flags.

20.4.29 BoardInit0 Routine

The **BoardInit0** function is normally called with routine linkage to perform very early initialization of the system, including participating in the decision about whether a warm boot or a cold boot has occurred.

This routine is called before the CMOS shutdown byte is tested during POST. Code should not be placed in this routine that could otherwise be placed in **BoardInit1**, since the bulk of initialization should be placed there unless there is a good reason not to place it there.

Called from POST, this function accepts as a parameter in the SP register a value of 0 if a cold boot has occurred, or -1 if POST has determined that a warm boot may have occurred. If this routine determines that a cold boot has occurred, then it sets SP to 0. Similarly, if it determines that a warm boot may have occurred, then it sets SP to -1. If it has no additional information to provide POST during the boot detection process, then it does not change SP.

Commonly, this routine calls **CpuInit0** and **CsInit0**, to initialize the CPU and chipset respectively, and to allow these modules to participate in the determination about whether a warm or cold boot has occurred.

This routine is entered with interrupts disabled and cannot enable them.

Input Parameters:

DS - BIOS data segment (40h).
SP - 0 if POST detects cold boot, else -1 if possible warm boot.

Output Parameters:

DS - BIOS data segment (40h).
SP - 0 if BPM detects cold boot, else -1 if possible warm boot.

Unpreserved Registers:

All but DS, BP, including SS.

20.4.30 BoardInit1 Routine

The **BoardInit1** function is called with routine linkage to perform the bulk of initialization of the system. This may include the CPU, chipset, Super I/O controllers, and all other functions of the board that must be initialized, especially relating to DRAM control, bus clocking, and enabling or disabling of certain functions as the design requires.

Normally, this routine calls **CpuInit1** and **CsInit1** to initialize the CPU and chipset, respectively. The BPM implementor may elect to remove either or both of these calls, and handle the relevant initialization directly in the BPM, if desired.

Alternatively, the BPM implementor may provide additional initialization beyond what **CpuInit1** and **CsInit1** provide. For example, a Super I/O controller's function control registers may need to be initialized so that UARTs, a floppy disk controller, parallel ports, and an IDE interface be made available to the system. In other cases, it may be necessary to disable certain duplicated functions found in both the chipset and in a Super I/O controller.

This routine is entered with interrupts disabled and cannot enable them.

Input Parameters:

DS - BIOS data segment (40h).

Output Parameters:

CY - set if failure, else clear if success.
DS - BIOS data segment (40h).

Unpreserved Registers:

All but DS, BP, including SS.

20.4.31 BoardInit4 Procedure

The **BoardInit4** function is called in the middle of POST with procedure linkage to perform any initialization that may be required by the BPM after DRAM is operational, but before the keyboard and video controllers are initialized.

If no custom initialization is required at this point by the OEM's design, then this routine need not be implemented by the OEM.

Input Parameters:

None.

<u>Output Parameters:</u>

CY - set if failure, else clear if success.
DI - POST_BEEP_BOARD, if failure.

<u>Unpreserved Registers:</u>

All but DS, SS, and SP.

20.4.32 BoardInit6 Procedure

The **BoardInit6** function is called in the middle of POST with procedure linkage to perform any initialization that may be required by the BPM after the keyboard and video controllers have been initialized, but before the power-on message is displayed.

If no custom initialization is required at this point by the OEM's design, then this routine need not be implemented by the OEM.

<u>Input Parameters:</u>

None.

<u>Output Parameters:</u>

None.

<u>Unpreserved Registers:</u>

All but DS, SS, and SP.

20.4.33 BoardInit8 Procedure

The **BoardInit8** function is called at the end of POST, after Manufacturing Mode initialization has taken place, but before the Configuration Box has been displayed, in preparation for booting the operating system. This function's architected purpose is to allow the BPM to receive control at a time when it is safe to configure the chipset, Super I/O, and/or high-integration CPU in special ways according to the values stored in the BPM's proprietary CMOS cells.

See also **BoardInitFields**, which also reads the contents of these unarchitected CMOS cells and stores the information in the RAM array used by the Custom SETUP screen, and **BoardSaveFields**, which is called by the Custom SETUP screen to save changes in RAM back to CMOS.

If no custom SETUP is required in the design, then this routine need not be implemented by the OEM.

<u>Input Parameters:</u>

None.

Output Parameters:

None.

Unpreserved Registers:

All but SS and SP.

20.4.34 BoardInitAppRom Routine

The **BoardInitAppRom** function is called with procedure linkage during POST right before the application (or operating system) ROM at the usual E000h segment is called as a ROM extension. This allows the BPM to receive control so that it may cause the ROM to be visible in the address space at the time the call is made, in systems where this segment serves several purposes.

The segment value E000h is configurable. The actual configuration parameter that is used to change this value is **CONFIG_MINI_DOS_SCAN**, documented in Chapter 7.

This function has been made obsolete by function **BoardFsInit**. Use of **BoardInitAppRom** in future designs is discouraged.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

20.4.35 BoardInitDma Routine

The **BoardInitDma** function is called with routine linkage to test and initialize the DMA controller hardware in the system, and to return a status that indicates whether there are any failures in the hardware.

If no DMA controller hardware is available, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_DMA_BOARD** is enabled. Normally, this routine calls **CpuInitDma** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

20.4.36 BoardInitFields Procedure

The **BoardInitFields** function is called by the SETUP screen system to initialize the RAM scratch area used by the Custom SETUP screen with values read from the unarchitected CMOS cells maintained by the BPM.

The Custom SETUP screen never manipulates these CMOS cells directly; rather, it requests that **BoardInitFields** interpret and initialize the RAM fields accordingly (which it then uses), and then upon exit from the Custom SETUP screen, **BoardSaveFields** is called to interpret the RAM scratch area fields and store appropriate values in CMOS.

See also **BoardInit8**, which also reads the contents of these unarchitected CMOS cells and uses that information to program the chipset and Super I/O controller, and **BoardSaveFields**, which is called by the Custom SETUP screen to save changes in RAM back to CMOS.

If no custom SETUP is required in the design, then this routine need not be implemented by the OEM.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

All but SS and SP.

20.4.37 BoardInitIntCtrl Routine

The **BoardInitIntCtrl** function is called with routine linkage to test and initialize the interrupt controller hardware in the system, and to return a status that indicates whether there are any failures in the hardware.

If no interrupt controller hardware is available, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_INT_BOARD** is enabled. Normally, this routine calls **CpuInitIntCtrl** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

20.4.38 BoardInitRefresh Routine

The **BoardInitRefresh** function is called with routine linkage to test and initialize the DRAM refresh controller hardware in the system, and to return a status that indicates whether there are any failures in the hardware.

If no refresh controller hardware is available, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_REFRESH_BOARD** is enabled. Normally, this routine calls **CpuInitRefresh** and **CsInitRefresh** to route the request to the CPU and chipset modules, respectively.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

20.4.39 BoardInitTimer Routine

The **BoardInitTimer** function is called with routine linkage to test and initialize (i.e., start running) any timers in the system, and to return a status that indicates whether there are any failures in the timer hardware.

If no timers are available, or if the adaptation engineer does not wish to test the timers, then this function should return with the carry flag clear. If timers are tested and initialize properly, then this routine returns with the carry flag clear if the timers are operational, or set if they fail the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_TIMER_BOARD** is enabled. Normally, this routine calls **CpuInitTimer** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

20.4.40 BoardInitWatchdog Routine

The **BoardInitWatchdog** function is called with routine linkage to test and initialize the watchdog timer hardware in the system, and to return a status that indicates whether there are any failures in the watchdog timer hardware.

If no watchdog timer is available, or if the adaptation engineer does not wish to test the hardware, then this function should return with the carry flag clear. If the hardware tested and initialized properly, then this routine returns with the carry flag clear if the hardware is found to be operational, or set if it fails the test.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_WATCHDOG** and **OPTION_WATCHDOG_BOARD** are enabled. Normally, this routine calls **CpuInitWatchdog** or **CsInitWatchdog** to route the request to the CPU or chipset modules, respectively.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP, including SS.

20.4.41 BoardKickWatchdog Procedure

The **BoardKickWatchdog** function is called with procedure linkage to restart the watchdog timer controller hardware in the system, if it exists. Restarting causes the timer to be reloaded, so that it will take the entire expiration period for the timer to expire.

If no hardware is available, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be restarted.

This routine is entered with interrupts disabled and cannot reenable them.

This routine will be called if **OPTION_SUPPORT_WATCHDOG** and **OPTION_WATCHDOG_BOARD** are enabled. Normally, this routine calls **CpuKickWatchdog** or **CsKickWatchdog** to route the request to the CPU or chipset modules, respectively.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

All but BP.

20.4.42 BoardMapAddress Procedure

The **BoardMapAddress** function is normally called by Media Control Layer in the core BIOS with procedure linkage to translate a 32-bit media address to either a real-mode windowed address or a 32-bit physical address.

Normally, this routine calls the underlying **CsMapAddress** function in the CSPM, so that the chipset has a chance to perform a windowing function to keep the mapping in real mode, if possible. Then, if the CSPM does not implement the windowing function, it uses direct physical mapping if the address is over 1MB, and real mode mapping if under 1MB.

Occasionally, it may be necessary for this routine to adjust the 32-bit media address before passing it on to the **CsMapAddress** function. For example, if a discontiguous Flash array consisted of separate devices in different address spaces, this routine could modify incoming addresses (perhaps through a lookup table) and present a contiguous media address space, even though the parts were discontiguous in the address space.

This routine is called if **OPTION_SUPPORT_MCL** is enabled.

If a direct physical mapping is to be performed, then the CY flag should be cleared on exit.

Input Parameters:

DX:AX - 32-bit media address (see Chapter 13 for details).

Output Parameters:

CY - set if result is 16:16 real mode, else clear if 32-bit physical address.

If real mode translation:

DX:AX - 16:16 real-mode translation of the input address.
CX - number of bytes visible at the address within the window.

If physical translation:

DX:AX - 32-bit physical address corresponding to the 32-bit media address.

Unpreserved Registers:

Flags.

20.4.43 BoardPciControl Procedure

The **BoardPciControl** function is called with procedure linkage from several places in the PCI device manager of the 16-bit core BIOS to provide notification and opportunity for control at various places during initialization. It may be used to debug and control PCI VGA boot device selection and System IRQ to PCI INT routing.

This routine is notified when PCI VGA devices have been detected, when enumeration is completed, when the PCI VGA boot device is being assigned, and when the PCI VGA scan is completed.

BoardPciControl accepts may be used to modify the policy used for selection of the PCI VGA boot device. When this routine is called, the caller within the core BIOS passes a context associated with the call in the top half of the EAX general purpose register. This BPM routine can then inspect the EAX register to determine whether the call is of interest or not.

When called with the context set to **BPC_FUNC_VGA_SCAN** or **BPC_FUNC_VGA_SCAN0**, then AX contains the encoded bus, device, and function value for a detected PCI VGA device. The value in AX may be inspected and stored if it is the best candidate found thus far, or a list

may be stored to allow a final decision to be made when this routine is called with context **BPC_FUNC_GET_VGA_LOC**. In order to scan all PCI VGA devices, the AX register should be set to 0FFFFh before returning from the **BPC_FUNC_VGA_SCAN** and **BPC_FUNC_VGA_SCAN0** invocations. Finally, when this routine is called with the **BPC_FUNC_GET_VGA_LOC** context, the AX register should be set by this routine to the chosen bus, device, and function number (encoded appropriately) of the PCI VGA device used for POST.

The default policy for selecting the PCI VGA boot device is to select the PCI VGA device found on the highest-numbered PCI bus, or the highest-numbered device found on PCI bus 0. The reasoning for this policy is that rack-mounted equipment with Compact PCI, for example, might have the highest-numbered PCI bus available for a technician to plug-in a VGA card. There might exist VGA devices elsewhere on the bus hierarchy that do not have connectors. Likewise, on bus 0, the board may implement an on-board VGA and it would be desirable to be able to override it with a PCI VGA add-in card.

For System IRQ to PCI INT routing, this BPM routine is called when the initial routing has been made, when the routing is modified by the **PciIrqTbl**, and when the final assignment is made just before **BoardAssignPciIrq** is called for each PCI INT line, where at least one PCI device has requested the assignment of an IRQ.

For **BPC_FUNC_SCAN** or **BPC_FUNC_VGA_SCAN0** contexts, this routine must only return in AX the incoming value of AX or 0FFFFh. Other values will cause undefined and most likely erroneous system behavior.

This routine must not respond to unknown BPC_FUNC values; in these cases, all registers must be preserved, for forward compatibility.

This routine will be called if **OPTION_SUPPORT_PCI** is enabled.

Input Parameters:

High portion of EAX – A 16-bit context, defined in INC\PCI.INC, that defines the reason for the call to this BPM routine. The following pre-defined values are architected at the time of this writing:

BPC_FUNC_VGA_SCAN – Scan found VGA, not on PCI bus 0.
BPC_FUNC_ENUM_DONE – PCI enumeration completed.
BPC_FUNC_VGA_SCAN0 – Scan found VGA on PCI bus 0.
BPC_FUNC_GET_VGA_LOC – Return desired boot VGA bus/dev/func.
BPC_FUNC_SCAN_VIDEO_DONE – Video scan completed.
BPC_FUNC_INIT_PCI_IRQ – (EDX) = initial IRQ to PCI INT routing.
BPC_FUNC_IRQ_REROUTE – Rerouting PCI INT line due to PciIrqTbl.
BPC_FUNC_FINAL_PCI_IRQ – (EDX) = final IRQ to PCI INT routing.

Other input parameters depend on the input context, as follows:

BPC_FUNC_VGA_SCAN:

AX – Detected PCI VGA bus/dev/func.

BPC_FUNC_VGA_SCAN0:

AX – Detected PCI VGA bus/dev/func.

BPC_FUNC_ENUM_DONE:

No input parameters.

BPC_FUNC_GET_VGA_LOC:

AX – Proposed PCI VGA bus/dev/func for VGA boot device. A value of 0FFFFh indicates no PCI VGA boot device has been found or selected yet.

BPC_FUNC_SCAN_VIDEO_DONE:

No input parameters.

BPC_FUNC_INIT_PCI_IRQ:

EDX – Contains the initial routing of System IRQs to PCI INTs, one routing assignment per 8-bit byte. Thus bits 0-7 contain the IRQ assignment for PCI INT#A, bits 8-15 contain the IRQ for PCI INT#B, and so on. Note that IRQ 0 is not a valid assignment for PCI INTs and is used to indicate an initial assignment could not be made.

BPC_FUNC_IRQ_REROUTE:

BX – Specifies the PCI bus/dev/func whose **PciIrqTbl** entry conflicts with the current System IRQ to PCI INT routing.

SI – Specifies the PCI INT line (0=A, 1=B, 2=C, 3=D) being rerouted because of a conflict with a **PciIrqTbl** entry.

AH – Specifies the current IRQ routed to the line specified by the parameter in the SI register.

AL – Specifies the new IRQ assignment for the line specified by the parameter in the SI register.

BPC_FUNC_FINAL_PCI_IRQ:

EDX – Contains the final routing of System IRQs to PCI INTs, one routing assignment per 8-bit byte. Thus bits 0-7 contain the IRQ assignment for PCI INT#A, bits 8-15 contain the IRQ for PCI INT#B, and so on. Note that IRQ 0 is not a valid assignment for PCI INTs and is used to indicate an initial assignment could not be made.

Output Parameters:

Output parameters depend on the input context, as follows:

BPC_FUNC_VGA_SCAN:

AX – Accepted PCI VGA bus/dev/func, or 0FFFFh to continue the scan.

BPC_FUNC_VGA_SCAN0:

AX – Passed through bus/dev/func from entry only of PCI VGA device as PCI bus 0 is scanned, or 0FFFFh to continue scanning.

BPC_FUNC_ENUM_DONE:

No output parameters.

BPC_FUNC_GET_VGA_LOC:

AX – Accepted PCI VGA bus/dev/func for VGA boot device. A value of 0FFFFh indicates no PCI VGA boot device has been found or selected.

BPC_FUNC_SCAN_VIDEO_DONE:

No output parameters.

BPC_FUNC_INIT_PCI_IRQ:

EDX – Contains the initial routing of System IRQs to PCI INTs, which may be modified from the input parameter, but is likely to be overridden by the PciIrqTbl. Use BPC_FUNC_FINAL_PCI_IRQ to control the final assignment of PCI IRQ routing.

BPC_FUNC_IRQ_REROUTE:

AL – Specifies the new IRQ assignment for the line specified by the parameter in the SI register. Changing the value passed in is discouraged since it should be possible to fix the problem by editing the PciIrqTbl, but there may be circumstances where this might be necessary.

BPC_FUNC_FINAL_PCI_IRQ:

EDX – Contains the final routing of System IRQs to PCI INTs, as modified (or untouched) by this BPM function.

Unpreserved Registers:

Flags.

20.4.44 BoardPciReadScratch Procedure

The **BoardPciReadScratch** function is called with procedure linkage from the PCI subsystem in the core BIOS to read from an 8-bit hardware location that does not involve a RAM access. There is no requirement that the contents of this storage location survive across a warm or cold boot. This mechanism is necessary because it may be called in a protected mode context in which there is no addressibility to the BIOS Data Area or the Extended BIOS Data Area.

This routine will be called if **OPTION_SUPPORT_PCI** is enabled. The default BPM routine uses an architected CMOS cell for this purpose. The OEM may supply an alternate mechanism

for reading and writing this data by defining new implementations for the **BoardPciReadScratch** and **BoardPciWriteScratch** routines.

This function must be implemented in the dual-build (16-bit/32-bit) file of the BPM (BPM1632.ASM) in order to support both the 16-bit and 32-bit PCI services.

Input Parameters:

None.

Output Parameters:

AL – value read from storage area.

Unpreserved Registers:

Flags.

20.4.45 BoardPciWriteScratch Procedure

The **BoardPciWriteScratch** function is called with procedure linkage from the PCI subsystem in the core BIOS to write to an 8-bit hardware location that does not involve a RAM access. There is no requirement that the contents of this storage location survive across a warm or cold boot. This mechanism is necessary because it may be called in a protected mode context in which there is no addressibility to the BIOS Data Area or the Extended BIOS Data Area.

This routine will be called if **OPTION_SUPPORT_PCI** is enabled. The default BPM routine uses an architected CMOS cell for this purpose. The OEM may supply an alternate mechanism for reading and writing this data by defining new implementations for the **BoardPciReadScratch** and **BoardPciWriteScratch** routines.

This function must be implemented in the dual-build (16-bit/32-bit) file of the BPM (BPM1632.ASM) in order to support both the 16-bit and 32-bit PCI services.

Input Parameters:

AL – value to write to storage area.

Output Parameters:

None.

Unpreserved Registers:

Flags.

20.4.46 BoardPostCodeCom Routine

The **BoardPostCodeCom** function is called with routine linkage from the **POSTCODECOM** macro to cause a null-terminated ASCII string to be displayed on the **POSTCODECOM**

debugging device managed by this BPM routine. Because this routine has register linkage and not stack linkage, and because it is called by the POSTCODECOM macro in situations where destroyed registers could negatively impact POST, this routine must be carefully crafted to avoid using any memory, stack, or destroy additional registers besides those allowed to be destroyed, as indicated below.

The default implementation of this BPM routine writes the ASCIIZ string to an 8250-compatible UART device addressed at the I/O location specified by the **CONFIG_POST_PROGRESS_COM**. The UART's communication parameters are set by the **BoardPostCodeComInit** BPM function, described elsewhere in this Chapter.

The **CONFIG_WAIT_PROGRESS_COM** parameter is used by the default implementation of this routine to determine how the routine will handle handshaking with the monitoring host. By default, RTS/CTS is used (the value of this parameter is set to 0). When this value is nonzero, then the value is interpreted by the default version of this BPM routine as a delay factor to pause between each character being printed. This allows the feature to work with very minimal hardware setup established. Note that the spin loop, by its very nature, can behave differently when cache is enabled/disabled, and when the BIOS is shadowed or not. Therefore, the delay may have to be established with an excessively high value to accommodate fast processors with L2 cache and shadowing enabled. This will result in slow progress messages during early POST. These restrictions only apply when the OEM changes the value of this parameter to accommodate a hardware-limited situation where RTS/CTS cannot be used.

Note that if the BPM subsequently reprograms the UART or the component containing the UART (such as a Super I/O component or Southbridge part), then this function may appear to stop working. The responsibility for not reprogramming the address or communications parameters of the UART being used for this service to ensure continued operation during POST remains with the OEM.

This routine has a very special linkage convention because of its use throughout the core BIOS in different (and sometimes limited) contexts. Particularly, the return address is automatically calculated by advancing BP to the location immediately following the zero byte, where the next instruction after the **POSTCODECOM** macro invocation can be found. Thus, the following is a general format of a POSTCODECOM macro expansion within a framework of surrounding code:

```
... executable instructions here ...
Rcall BoardPostCodeCom
db    'Here is the string to print.', 0
... more executable instructions here ...
```

Input Parameters:

CS:BP – Specifies a 16:16 real-mode pointer to an ASCIIZ (null-byte terminated) string.

Output Parameters:

None.

Unpreserved Registers:

AL, DX, and Flags.

20.4.47 BoardPostCodeComInit Routine

The **BoardPostCodeComInit** function is called with routine linkage from the mainline POST to initialize the **POSTCODECOM** debugging system, used by the **POSTCODECOM** macro to generate ASCIIZ progress messages on a debugging output device.

The default implementation of this BPM routine initializes the communication parameters of an 8250-compatible UART device addressed at the I/O location specified by the **CONFIG_POST_PROGRESS_COM**. The parameters are set to no parity, 8 data bits, one stop bit, and a baud rate specified by **CONFIG_POST_PROGRESS_BAUD**. Note that if the UART being addressed is embedded in a Super I/O or Southbridge component, the large component must be properly configured, usually through additional programming, to cause the UART device to be enabled so that the programming in this routine can operate. This programming should be placed in this routine as well.

Note that if the BPM subsequently reprograms the UART or the component containing the UART (such as a Super I/O component or Southbridge part), then this function may appear to stop working. The responsibility for not reprogramming the address or communications parameters of the UART being used for this service to ensure continued operation during POST remains with the OEM.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

AX, DX, and Flags.

20.4.48 BoardMemConfig Routine

The **BoardMemConfig** routine is normally called by POST with routine linkage to program the chipset to determine how many memory banks are available, what size of SIMMs are in each bank, and how the banks are to be interleaved and positioned in the address space.

Normally, this routine calls **CsMemConfig** in the CSPM to perform the work; however, at the BPM implementor's option, this work may be done in the BPM in this routine by inserting the code in this routine and removing the call to **CsMemConfig**.

This routine is called after the chipset has been initialized and DRAM refresh has been activated. However, no stack yet exists because low memory (the base 64KB) has not been tested. Therefore, this routine is not called with a valid stack, but may create one temporarily in the bottom 64KB of memory in order to perform certain functions. This would be a rare case, since all DRAM configuration methods implicitly require manipulation of chipset registers to change the RAS and CAS assignments of banks of DRAM in the system, and changing this while the

stack has valid data on it will cause the data to mapped to different locations or made unavailable.

It may be necessary to switch to protected mode temporarily during this routine's operation to test memory above 1MB. For sample code, consult routines **ToProt** and **ToReal** in HELPER.ASM. During extended memory testing, make sure that the A20 gate is properly enabled, or memory addresses will wrap to lower memory.

Typically, the algorithm by which the memory geometry is determined is provided by the chipset manufacturer. If you need help implementing this routine, consult the manufacturer for the recommended algorithm. Remember that you can either perform this work in this function, or in the **BoardInit1** routine, although **BoardInit1** would need to enable DRAM refresh if that were the case.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.
DI - if failure, **POST_BEEP_CHIPSET**.

Unpreserved Registers:

All but BP and SS.

20.4.49 BoardPwrLvl Procedure

The **BoardPwrLvl** function is called with procedure linkage to notify the BPM of a change in power state in the system. This function is called by the Power Management Subsystem in the core BIOS, when the **Board** device is enabled in the **POWER_DEVID** device tree.

If no power control hardware is available, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware fails to perform the requested operation.

This routine is called if **OPTION_SUPPORT_POWERMAN** is enabled; and then **OPTION_POWERMAN_BOARD** is enabled and the **Board** device is in the device tree.

Input Parameters:

DS - segment of the extended BIOS data area (EBDA).
BX - device index of the board itself.
CL - new power level.
CH - old power level.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

DS, BX, CL, Flags.

20.4.50 BoardPostError Routine

The **BoardPostError** function is called with routine linkage by POST when a critical error occurs. This allows the BPM to take actions for purposes of logging errors, reporting a malfunction with a visual indicator, or interacting with a debugger in the lab.

The default version of this routine calls **BoardTestMode** to determine if the Manufacturing Mode hardware is attached, and if so, it enters Manufacturing Mode. The OEM may replace this functionality with whatever is required for a given design.

This routine is called if **OPTION_CRITICAL_BOARD** is enabled.

If this routine returns, then control will continue in POST to the next critical error handler (such as the speaker beep code). If this routine does not return, then it may perform any desired action, such as rebooting the system or halting.

Input Parameters:

DI - Beep code (see POSTERR.INC), normally used as a number of speaker beeps.

Output Parameters:

DI - Beep code as above.

Unpreserved Registers:

All but DI.

20.4.51 BoardReboot Procedure

The **BoardReboot** function is called with coroutine linkage to handle a reboot request that originates from the debugger, the **CTL-ALT-DEL** mechanism, or other internal functions of the BIOS.

This routine will be called if **OPTION_REBOOT_BOARD** is enabled. Normally, this routine calls **CpuReboot** or **CsReboot** to route the request to the CPU or chipset modules, respectively.

Input Parameters:

None.

Output Parameters:

None; should not return unless reboot cannot be performed.

Unpreserved Registers:

Flags.

20.4.52 BoardResetCmos Routine

The **BoardResetCmos** function is called with routine linkage by the core BIOS when the CMOS must be initialized with factory default values. This routine may indicate to its caller that the factory default table (initialized with configuration options maintained in the Project file), or the routine may perform the work itself.

This routine's purpose is to provide a way for the BPM implementor to write the CMOS data to nonvolatile storage when the CMOS storage is volatile; i.e., when it does not employ a battery in the design to save CMOS contents when power is removed from the system.

This routine will be called if **OPTION_SUPPORT_CMOS** is enabled.

Input Parameters:

None.

Output Parameters:

CY - set if factory default table should be used; clear if this routine performed the work.

Unpreserved Registers:

Flags.

20.4.53 BoardSaveCmos Procedure

The **BoardSaveCmos** function is called with procedure linkage by the SETUP screen system in response to the user's selection of "WRITE TO CMOS AND EXIT," so that the BPM can save the CMOS cell data to nonvolatile storage when the CMOS memory itself is volatile (no battery is used in the design, for example).

Routine **BoardResetCmos** may be used in conjunction with this routine to provide the OEM with a way to change the factory defaults in the production line or in the field without patching the BIOS or rebuilding it.

This routine will be called if **OPTION_SUPPORT_CMOS** is enabled.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

20.4.54 BoardSaveFields Procedure

The **BoardSaveFields** function is called by the SETUP screen system to interpret the fields in the RAM scratch area used by the Custom SETUP screen and store appropriate values into the unarchitected CMOS cells maintained by the BPM that represent the RAM scratch area fields' contents.

The Custom SETUP screen never manipulates these CMOS cells directly; rather, it requests that **BoardInitFields** interpret and initialize the RAM fields accordingly (which it then uses), and then upon exit from the Custom SETUP screen, **BoardSaveFields** is called to interpret the RAM scratch area fields and store appropriate values in CMOS.

See also **BoardInit8**, which also reads the contents of these unarchitected CMOS cells and uses that information to program the chipset and Super I/O controller, and **BoardInitFields**, which is called by the Custom SETUP screen to read the CMOS cells and initialize the RAM scratch area.

If no custom SETUP is required in the design, then this routine need not be implemented by the OEM.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

All but SS and SP.

20.4.55 BoardSetFastSpeed Procedure

The **BoardSetFastSpeed** function is called with procedure linkage to switch the system's clocking to the highest speed supported by the hardware, if speed-switching hardware exists.

If no hardware is available, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be programmed.

This routine will be called if **OPTION_SPEED_BOARD** is enabled. Normally, this routine calls **CpuSetFastSpeed** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

20.4.56 BoardSetSlowSpeed Procedure

The **BoardSetSlowSpeed** function is called with procedure linkage to switch the system's clocking to the slowest speed supported by the hardware, if speed-switching hardware exists.

If no hardware is available, then this function should return with the carry flag clear. If the hardware exists, then this routine returns with the carry flag set if the hardware cannot be programmed.

This routine will be called if **OPTION_SPEED_BOARD** is enabled. Normally, this routine calls **CpuSetSlowSpeed** to route the request to the CPU module.

Input Parameters:

None.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

20.4.57 BoardSetVideoMode Procedure

The **BoardSetVideoMode** function is called with procedure linkage from the 6845 CRT controller driver software in the core BIOS when it receives a request to set the video mode.

This call-out gives the BPM a chance to adjust the BIOS Data Area (BDA) fields set by the core BIOS before they are actually used, so that the number of colums, rows, and other fields can be changed. Changing these parameters in this routine is necessary to support certain LCD controllers that support screen geometries other than than 25 lines and 80 columns.

This routine should perform no action if no translation of parameters is necessary. The OEM should feel free to review VIDEO.ASM (the 6845 CRT driver in the core BIOS) to see the conditions under which this routine is called, and use those conditions as necessary to make the system work properly.

Input Parameters:

See VIDEO.ASM.

Output Parameters:

See VIDEO.ASM.

Unpreserved Registers:

See VIDEO.ASM.

20.4.58 BoardSioReadReg Procedure

The **BoardSioReadReg** function is called with procedure linkage from the debugger to read a register from a board-level entity, usually a Super I/O part or South Bridge PCI component. The exact semantics are left to the implementor of the BPM.

This routine will be called if **OPTION_SUPPORT_DEBUGGER** is enabled.

Input Parameters:

AX – 16-bit register number to be read.

Output Parameters:

CY – set if failure to read register, else clear if success.
AX – if success, the 16-bit value read from the specified register.

Unpreserved Registers:

Flags.

20.4.59 BoardSioWriteReg Procedure

The **BoardSioWriteReg** function is called with procedure linkage from the debugger to write a 16-bit value to a register in a board-level entity, usually a Super I/O part or South Bridge PCI component. The exact semantics are left to the implementor of the BPM.

This routine will be called if **OPTION_SUPPORT_DEBUGGER** is enabled.

Input Parameters:

AX – 16-bit register number to be written.
DX – 16-bit value to be written to the register.

Output Parameters:

CY – set if failure to read register, else clear if success.

Unpreserved Registers:

Flags.

20.4.60 BoardShadowArea Procedure

The **BoardShadowArea** function is normally called by POST with procedure linkage to shadow an area of ROM using RAM remapped by the chipset. For procedural details about how shadowing is typically performed in a system, see Chapter 19, routine **CsShadowArea**.

This routine will be called if **OPTION_SUPPORT_SHADOW** is enabled. Normally, this routine calls **CsShadowArea** to route the request to the chipset module.

Input Parameters:

DI - index of area to shadow, as follows:

0000h - segment C000h.
0001h - segment C400h.
0002h - segment C800h.
0003h - segment CC00h.
0004h - segment D000h.
0005h - segment D400h.
0006h - segment D800h.
0007h - segment DC00h.
0008h - segment E000h.
0009h - segment E400h.
000ah - segment E800h.
000bh - segment EC00h.
000ch - segment F000h.

Output Parameters:

CY - set if failure, else clear if success.

Unpreserved Registers:

Flags.

20.4.61 BoardTestMode Procedure

The **BoardTestMode** function is called with procedure linkage to determine if the system should enter Manufacturing Mode. This call-out allows the BPM implementor to provide code which checks the status of a package pin, shunt, or other hardware mechanism in the hardware to make this determination.

Because the BPM's decision may be arbitrarily complex, it is possible for the implementor of this routine to cause Manufacturing Mode to be entered if the hardware signal is present, or if a

software-programmable register is set to a specific value. This allows a target to be booted, and with the debugger, the register can be set by the OEM in the lab. Upon rebooting the system, Manufacturing Mode is then entered. Immediately upon entry into Manufacturing Mode, the system calls **BoardDisableTestMode**, which allows the BPM implementor to reset this software-programmable register so that the system does not continually enter Manufacturing Mode thereafter.

The default version of this routine always returns a “do not enter Manufacturing Mode” value, so that POST can continue to boot the operating system rather than enter Manufacturing Mode.

Input Parameters:

None.

Output Parameters:

CY - Set to enter Manufacturing Mode, else clear.

Unpreserved Registers:

Flags.

20.4.62 BoardTimerTick Procedure

The **BoardTimerTick** function is called with procedure linkage from the INT 08h timer tick handler in the core BIOS, so that the BPM has a way to receive notification that timer ticks have occurred.

The OEM may use this routine for any purpose that helps to implement other BPM functions, such as timing out certain requests. The OEM may also want to use the routine for proprietary value-added functions, such as the regular polling of hardware in the background, for example.

Commonly, this routine is coded to interact with other routines in the BPM through memory fields in the Extended BIOS Data Area (EBDA), and most commonly, in the **BoardData** byte array that is reserved for BPM use (see `DATA.INC` for layout of the EBDA).

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

20.4.63 BoardUnMapAddress Procedure

The **BoardUnMapAddress** function is normally called by Media Control Layer in the core BIOS with procedure linkage upon completion of a Media API request to allow the BPM implementor to restore the registers previously changed by the BoardMapAddress procedure to their original, user-defined, state.

Implementation of this routine is called for if **BoardMapAddress** is implemented, and if user-level application code depends on the state of certain board-level or chipset-level hardware to remain under the sole control of the application. It is not necessary to implement this routine unless the restoration of mapping hardware registers is a requirement.

It should be noted that this routine is called once for each Media API request made, whether or not that API request caused any calls to **BoardMapAddress** or not. The **BoardUnMapAddress** function should be able to handle the condition where the **BoardMapAddress** function had previously been called zero, one, or more times for the given Media API request that is causing **BoardUnMapAddress** to be invoked. The recommended implementation method for the engineer desiring to incorporate an unmap function is to use RAM variables located in the EBDA's **BoardData** array to save the old contents of the mapping registers in the **BoardMapAddress** function, and then use an additional variable to indicate that the hardware has been programmed and needs to be restored. Subsequent invocations of **BoardMapAddress** can inspect the flag and not save additional copies of the registers until the flag is cleared. Then, **BoardUnMapAddress** can inspect the flag to determine if the registers need to be restored or not, and on that basis, use the values saved in the **BoardData** array by **BoardMapAddress** to reprogram the original contents of the hardware mapping registers.

This routine is called if **OPTION_SUPPORT_MCL** is enabled.

Input Parameters:

None.

Output Parameters:

None.

Unpreserved Registers:

Flags.

20.4.64 BoardUnmaskInt Procedure

The **BoardUnmaskInt** function is called with procedure linkage to enable a specific interrupt level in the system.

This routine will be called if **OPTION_INT_BOARD** is enabled. Normally, this routine calls **CpuUnmaskInt** to route the request to the CPU module.

Input Parameters:

AL - interrupt vector level to initialize, as follows:

00h - IRQ0, or INT 08h.

01h - IRQ1, or INT 09h.
02h - IRQ2, or INT 0ah.
03h - IRQ3, or INT 0bh.
04h - IRQ4, or INT 0ch.
05h - IRQ5, or INT 0dh.
06h - IRQ6, or INT 0eh.
08h - IRQ7, or INT 0fh.

Output Parameters:

None.

Unpreserved Registers:

Flags.

PART III

BIOS FUNCTION REFERENCE

This part of the EMBEDDED BIOS reference documentation describes the Application Programming Interface (API) exposed by the core BIOS to DOS, Windows and application programs.

Chapter 21

BIOS FUNCTION REFERENCE

This chapter defines the application programming interface (API) to the BIOS services supported by Embedded BIOS.

It is not intended to document interrupts that strictly do not provide services; consult Chapter 3 for an architectural overview of EMBEDDED BIOS including BIOS up-calls and tables pointed to by interrupt vectors.

21.1 INT 10h, Video BIOS Services

This section explains the video BIOS application program interface (API). The video BIOS is called through software interrupt 10H. Many services are provided to modify or inspect the contents of the video display.

21.1.1 Set Video Mode (00h)

The Set Video Mode video BIOS function is called to set the video mode registers on the video controller for the specified mode of operation. It then clears the screen, positions the cursor at the upper left hand corner of the screen (0,0) and resets the color palette to known values.

Input Parameters:

AH - 00h, indicating the Set Video Mode Function.
AL - Video mode byte.

00h - Text mode, 16 colors, 40x25, 320x200.
01h - Text mode, 16 colors, 40x25, 320x200.
02h - Text mode, 16 colors, 80x25, 640x200.
03h - Text mode, 16 colors, 80x25, 640x200.
04h - Graphics, 4 colors, 40x25, 320x200.
05h - Graphics, 4 colors, 40x25, 320x200.
06h - Graphics, 2 colors, 80x25, 640x200.

07h - Text mode, monochrome, 80x25.
0dh - Graphics, 16 colors, 40x25, 320x200.
0eh - Graphics, 16 colors, 80x25, 640x200.
0fh - Graphics, monochrome, 80x25.
10h - Graphics, 4/16 colors, 80x25, 640x350.

Output Parameters:

AL - Video mode as actually set.

21.1.2 Set Cursor Size (01h)

The Set Cursor Size video BIOS function is called to set the size of the cursor in text modes. The parameters are simply the top and bottom scan lines, in the form of bit masks.

Input Parameters:

AH - 01h, indicating the Set Cursor Size Function.
CH - Top scan line, a complex field as follows:

zz000000b - Must be zero.
00100000b - Shut cursor off.
000xxxxxb - Top scan line.

CL - Bottom scan line, a complex field as follows:

zzz00000b - Must be zero.
000xxxxxb - Bottom scan line.

Output Parameters:

none.

21.1.3 Set Cursor Position (02h)

The Set Cursor Position video BIOS function is called to set the (X,Y) coordinates of the hardware cursor on the screen. The X coordinate is expressed as a column number, beginning with 0 equal to the left-most column on the screen. The Y coordinate is a row number, beginning with 0 equal to the top-most row on the screen.

Input Parameters:

AH - 02h, indicating the Set Cursor Position Function.
BH - Video page number (0 for first page).
DH - Row number (0=top-most row).
DL - Column number (0=left-most column).

Output Parameters:

AX - 0000h.

21.1.4 Read Cursor Position (03h)

The Read Cursor Position video BIOS function is called to return the (X,Y) coordinates of the hardware cursor on the screen. The X coordinate is expressed as a column number, beginning with 0 equal to the left-most column on the screen. The Y coordinate is a row number, beginning with 0 equal to the top-most row on the screen.

Input Parameters:

AH - 03h, indicating the Read Cursor Position Function.
BH - Video page number (0 for first page).

Output Parameters:

AX - 0000h.
CH - Starting cursor scan line.
CL - Ending cursor scan line.
DH - Row number (0=top-most row).
DL - Column number (0=left-most column).

21.1.5 Read Light Pen Position (04h)

The Read Light Pen video BIOS function is called to return the position status of a lightpen. EMBEDDED BIOS does nothing when this function is called.

Input Parameters:

AH - 04h, indicating the Read Light Pen Position Function.

Output Parameters:

AH - Activity flag:

00h - Light pen is not active.
01h - Coordinates returned.

BX - Pixel column (0-319).
CH - Raster line (0-199).
CL - Raster line (0-..)..
DH - Row number (0=top-most row).
DL - Column number (0=left-most column).

21.1.6 Select Video Page (05h)

The Select Video Page video BIOS function is called to change the page of the video buffer that is displayed by the 6845 and mapped into its screen regen area (B000H for monochrome, or B800H for color).

Input Parameters:

AH - 05h, indicating the Select Video Page Function.
AL - Page number, where 00h is the first page.

Output Parameters:

none.

21.1.7 Scroll Up Window (06h)

The Scroll Up Window video BIOS function is called to move the contents of a rectangular area on the screen up by a specified number of lines. If the window is specified to cover the entire screen, then the entire screen is scrolled.

Input Parameters:

AH - 06h, indicating the Scroll Up Window Function.
AL - Distance to scroll, in lines (0=blank window).
BH - Attribute byte to use on new lines.
CH - Top row of window.
CL - Left column of window.
DH - Bottom row of window.
DL - Right column of window.

Output Parameters:

none.

21.1.8 Scroll Down Window (07h)

The Scroll Down Window video BIOS function is called to move the contents of a rectangular area on the screen down by a specified number of lines. If the window is specified to cover the entire screen, then the entire screen is scrolled.

Input Parameters:

AH - 07h, indicating the Scroll Down Window Function.
AL - Distance to scroll, in lines (0=blank window).
BH - Attribute byte to use on new lines.
CH - Top row of window.
CL - Left column of window.
DH - Bottom row of window.
DL - Right column of window.

Output Parameters:

none.

21.1.9 Read Char/Attr From Screen (08h)

The Read Char/Attr Pair video BIOS function is called to return the character and attribute located at the current cursor position for the specified page.

Input Parameters:

AH - 08h, indicating the Read Char/Attr Pair Function.
BH - Video page number (0=first page).

Output Parameters:

AH - Attribute byte.
AL - Character.

21.1.10 Write Char/Attr to Screen (09h)

The Write Char/Attr Pair video BIOS function is called to store a character and attribute at the current cursor position for the specified page, without advancing the cursor. The function also allows a repeat count to store multiple characters in sequential columns on the screen.

Input Parameters:

AH - 09h, indicating the Write Char/Attr Pair Function.
AL - Character to store.
BH - Video page number (0=first page).
BL - Attribute byte.
CX - Repeat count.

Output Parameters:

none.

21.1.11 Write Character to Screen (0ah)

The Write Character video BIOS function is called to store a character at the current cursor position for the specified page, without advancing the cursor, using the attribute already defined for that cursor position. The function also allows a repeat count to store multiple characters in sequential columns on the screen.

Input Parameters:

AH - 0ah, indicating the Write Character Function.
AL - Character to store.
BH - Video page number (0=first page).
CX - Repeat count.

Output Parameters:

none.

21.1.12 Set Color Palette (0bh)

The Set Color Palette video BIOS function is called to initialize the 6845's video palette register for graphics modes.

Input Parameters:

AH - 0bh, indicating the Set Color Palette Function.
BH - 00h to set the background color for 320x200 graphics modes, or the border color for 320x200 text modes, or foreground color for 640x200 graphics mode. Otherwise, 01h to set the palette register for 320x200 graphics mode.
BL - Value to set.

Output Parameters:

none.

21.1.13 Write Pixel (0ch)

The Write Pixel video BIOS function is called to store a color value in a pixel addressed by a specified row and column number in graphics mode.

Input Parameters:

AH - 0ch, indicating the Write Pixel Function.
AL - Color to set (set bit 10000000b for XOR mode).
BH - Video page number.
CX - Pixel column number.
DX - Pixel row number.

Output Parameters:

none.

21.1.14 Read Pixel (0dh)

The Read Pixel video BIOS function is called to return a color value in a pixel addressed by a specified row and column number in graphics mode.

Input Parameters:

AH - 0dh, indicating the Read Pixel Function.
BH - Video page number.
CX - Pixel column number.

DX - Pixel row number.

Output Parameters:

AL - Color of pixel.

21.1.15 Write Teletype Mode (0eh)

The Write Teletype video BIOS function is called to write a character to the display at the current cursor location, advancing the cursor to the next column. If the column would extend off the right edge of the screen, the column is reset to 0, and the row is incremented. If the row would move off the end of the screen, then the entire screen is scrolled.

The carriage-return, line-feed, and bell characters perform the same functions that they do on a real teletype.

Input Parameters:

AH - 0eh, indicating the Write Teletype Mode Function.
AL - Character to write.
BL - Foreground color, but only for graphics modes.
BH - Video page number.

Output Parameters:

none.

21.1.16 Return Video Status (0fh)

The Return Video Status video BIOS function is called to return the number of columns on the screen, the current video mode, and the active page number.

Input Parameters:

AH - 0fh, indicating the Return Video Status Function.

Output Parameters:

AH - Columns on the screen.
AL - Current display mode.
BH - Video page number.

21.2 INT 11h, Equipment List Service

The Equipment Status Interrupt is invoked to return the device flags as defined in the BIOS Data Area's **DevFlags** field.

Invocation:

INT 11H

Input Parameters:

none.

Output Parameters:

AX - Device flags.

21.3 INT 12h, Low Memory Size Service

The Low Memory Size Interrupt is invoked to return the size of low memory in kilobytes. This function automatically decrements the returned size by the 1KB Extended BIOS Data Area, located in the top 1KB of low memory.

To determine the amount of available extended memory, use INT 15h function 88h.

Invocation:

INT 12H

Input Parameters:

none.

Output Parameters:

AX - Kilobytes of low memory.

21.4 INT 13h, Disk Services

This section explains the disk BIOS application program interface (API). The disk BIOS is called through software interrupt 13H. Services are provided to reset the disk system, read the status of the last operation, read diskette sectors, write diskette sectors, verify diskette sectors, format disk tracks, read drive parameters, read drive types, detect media changes, set the media type, and set the media type for formatting.

Not all functions are available for all disk types. Note restrictions on each function that apply. For example, the ROM disk does not support write-oriented operations such as Write Sectors, Format Track, and so on.

The following error codes are returned by INT 13h services:

Status	Description
00h	No error
01h	Invalid function
02h	Address mark not found

03h	Media write-protected
04h	Sector not found
05h	Reset failed
06h	Media changed
07h	Hard drive parameter is invalid
08h	DMA overflow occurred
09h	DMA operation crossed 64KB boundary
0ah	Hard drive bad sector
0bh	Hard drive bad track
0ch	Invalid floppy disk media type
0dh	Invalid number of sectors
0eh	Control data address mark found
0fh	DMA arbitration out of range
10h	Unrecoverable read error
11h	Recoverable data area (ECC corrected)
20h	Floppy disk controller failure
40h	Seek to invalid track
80h	Timeout
aah	Hard drive not ready
bbh	Unknown hard disk drive error
cch	Hard drive write error
e0h	Hard drive status register error
ffh	Hard drive sense operation failed

Most disk operations are sector, track, and head-based. When specifying these parameters in INT 13h functions, the sector number is usually specified in the CL CPU register, and the low eight bits of the cylinder number is specified in the CH CPU register. An additional two high bits of cylinder number are stored in the top two bits of the CL CPU register, limiting the sector number stored in the CL CPU register to six bits (values 0-63).

21.4.1 Reset (00h)

The Reset disk BIOS function is called to reset the disk subsystem (ROM, RAM, RFD, floppy, and IDE). This function is used during POST and also whenever an error occurs as a result of a disk operation.

Input Parameters:

AH - 00h, indicating the Reset Function.
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Status code if failure (00h if success).

21.4.2 Read Status (01h)

The Read Status disk BIOS function is called to return the status of the last operation on the specified drive. This status is invalidated when an intervening INT 13h function is invoked (except for this function).

Input Parameters:

AH - 01h, indicating the Read Status Function.
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - 00h.
AL - Disk status code of last operation (00h if success).

21.4.3 Read Sectors (02h)

The Read Sectors disk BIOS function is called to read a sector run from the specified drive into a user-defined buffer. The read must not span a track or head boundary, and the buffer must not cross a 64KB DMA boundary in the physical address space.

Input Parameters:

AH - 02h, indicating the Read Sectors Function.
AL - Number of sectors.
CH - Bottom 8 bits of track number (0-based).
CL - ttssssss, as follows:
tt = top two bits of 10-bit track number,
ssssss = 6-bit sector number (1-based).
DH - Head number (0-based).
DL - Drive number.
ES:BX - Address of user buffer.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).
AL - Number of sectors actually read.

21.4.4 Write Sectors (03h)

The Write Sectors disk BIOS function is called to write a sector run to the specified drive from a user-defined buffer. The write must not span a track or head boundary, and the buffer must not cross a 64KB DMA boundary in the physical address space.

This function returns an error when accessing the ROM disk.

Input Parameters:

AH - 03h, indicating the Write Sectors Function.
AL - Number of sectors.
CH - Bottom 8 bits of track number (0-based).
CL - ttssssss, as follows:

tt = top two bits of 10-bit track number,
ssssss = 6-bit sector number (1-based).
DH - Head number (0-based).
DL - Drive number.
ES:BX - Address of user buffer.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).
AL - Number of sectors actually written.

21.4.5 Verify Sectors (04h)

The Verify Sectors disk BIOS function is called to verify that the address marks on a specified track can be read. It does not verify data integrity.

Input Parameters:

AH - 04h, indicating the Verify Sectors Function.
AL - Number of sectors.
CH - Bottom 8 bits of track number (0-based).
CL - ttssssss, as follows:
tt = top two bits of 10-bit track number,
ssssss = 6-bit sector number (1-based).
DH - Head number (0-based).
DL - Drive number.
ES:BX - Address of buffer containing field data.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).

21.4.6 Format Track (05h)

The Format Track disk BIOS function is called to format the specified track of a drive with specific address marks.

This function returns an error when accessing the ROM disk.

Input Parameters:

AH - 05h, indicating the Format Sectors Function.
AL - Number of sectors/this track.
CH - Bottom 8 bits of track number (0-based).
CL - ttssssss, as follows:
tt = top two bits of 10-bit track number,
ssssss = 6-bit sector number (1-based).
DH - Head number (0-based).

DL - Drive number.
ES:BX - Address of buffer containing field data.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).

21.4.7 Read Drive Parameters (08h)

The Read Drive Parameters disk BIOS function is called to return the geometry and disk type information for the specified drive. Additionally, the number of drives like the one specified is returned; i.e., if a floppy drive number is supplied, then the number of floppy drives is returned in the DL CPU register, and so on for hard drives. The ROM, RAM, and RFD drives count as floppy drives.

Input Parameters:

AH - 08h, indicating the Read Drive Parameters Function.
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).
BH - 00h (floppy drives only).
BL - Drive type, as follows (floppy drives only):

01h - 5.25", 360KB, 40 tracks.
02h - 5.25", 1.2MB, 80 tracks.
03h - 3.5", 720KB, 80 tracks.
04h - 3.5", 1.44MB, 80 tracks.

CH - Bottom 8 bits of maximum track number.
CL - ttssssss, as follows:
tt = top two bits of 10-bit maximum track number,
ssssss = 6-bit maximum sector number.

DH - Maximum head number.
DL - Number of drives installed.
ES:DI - Pointer to the diskette parameter table entry for a floppy drive.

21.4.8 Initialize Hard Disk Controller (09h)

The Initialize Hard Disk Controller disk BIOS function is called to initialize the disk controller with the values in the BIOS hard disk parameter tables pointed to by IVT entries 41h and 46h. See Chapter 3 for a description of the data structures pointed to by these tables.

This function returns an error when accessing a floppy disk or its emulator.

Input Parameters:

AH - 09h, indicating the Initialize Hard Disk Controller Function.
DL - Drive number (80h=C:, 81h=D:).

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).

21.4.9 Read Long Sectors (0ah)

The Read Long Sectors disk BIOS function is called to read a sector run from the specified drive into a user-defined buffer with a 4-byte error correction code (ECC) for each sector. The read must not span a track or head boundary, and the buffer must not cross a 64KB DMA boundary in the physical address space.

This function is only valid for hard disk drives.

Input Parameters:

AH - 0ah, indicating the Read Long Sectors Function.
AL - Number of sectors.
CH - Bottom 8 bits of track number (0-based).
CL - ttssssss, as follows:
tt = top two bits of 10-bit track number,
ssssss = 6-bit sector number (1-based).
DH - Head number (0-based).
DL - Drive number.
ES:BX - Address of user buffer.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).
AL - Number of sectors actually read.

21.4.10 Write Long Sectors (0bh)

The Write Long Sectors disk BIOS function is called to write a sector run from the specified drive from a user-defined buffer with a 4-byte error correction code (ECC) for each sector. The write must not span a track or head boundary, and the buffer must not cross a 64KB DMA boundary in the physical address space.

This function is only valid for hard disk drives.

Input Parameters:

AH - 0bh, indicating the Write Long Sectors Function.
AL - Number of sectors.

CH - Bottom 8 bits of track number (0-based).
CL - ttssssss, as follows:
tt = top two bits of 10-bit track number,
ssssss = 6-bit sector number (1-based).
DH - Head number (0-based).
DL - Drive number.
ES:BX - Address of user buffer.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).
AL - Number of sectors actually written.

21.4.11 Seek to Cylinder (0ch)

The Seek to Cylinder disk BIOS function is called to position a read/write head on a hard drive over a specified track. No data is transferred during this request.

This function is only valid for hard disk drives.

Input Parameters:

AH - 0ch, indicating the Seek to Cylinder Function.
CH - Bottom 8 bits of track number (0-based).
CL - tt000000, as follows:
tt = top two bits of 10-bit track number.
DH - Head number (0-based).
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).

21.4.12 Reset Hard Disk Controller (0dh)

The Reset Hard Disk Controller disk BIOS function is called to initialize the IDE controller. While function 00h resets both the floppy and hard disk controllers, this function only resets the hard drive controller.

This function is only valid for hard disk drives.

Input Parameters:

AH - 0dh, indicating the Reset Hard Disk Controller Function.
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).

21.4.13 Test Drive Ready (10h)

The Test Drive Ready disk BIOS function is called to verify that the hard drive specified in the DL CPU register is ready to perform additional functions.

This function is only valid for hard disk drives.

Input Parameters:

AH - 10h, indicating the Test Drive Ready Function.
DL - Drive number.

Output Parameters:

CY - Set if failure (not ready), else clear if success (ready).
AH - Disk status code (00h if success).

21.4.14 Recalibrate Drive (11h)

The Recalibrate Drive disk BIOS function is called to recalibrate a hard drive. Externally, this involves restoring all of the read/write heads to the track 0 position. Internally, this also involves rezeroing the feedback loop that determines the head position inside the drive. If read and write requests start encountering frequent correctable errors, this function should be called to recalibrate the heads.

This function is only valid for hard disk drives.

Input Parameters:

AH - 11h, indicating the Recalibrate Drive Function.
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).

21.4.15 Controller Diagnostic (14h)

The Controller Diagnostic disk BIOS function is called to initiate a diagnostic routine on the hard disk controller. The outcome of this diagnostic is returned in the status code.

This function is only valid for hard disk drives.

Input Parameters:

AH - 14h, indicating the Controller Diagnostic Function.
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).

21.4.16 Read Drive Type (15h)

The Read Drive Type disk BIOS function is called to return the disk type information for a disk unit.

Input Parameters:

AH - 15h, indicating the Read Drive Type Function.
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Drive type, as follows:

00h - Drive number is invalid.
01h - Diskette drive with no change line.
02h - Diskette drive with a change line.
03h - Fixed disk.

CX:DX - for fixed disks, a 32-bit number of 512-byte sectors.

21.4.17 Detect Media Change (16h)

The Detect Media Change disk BIOS function is called to return the status of the disk change line for a disk unit.

Input Parameters:

AH - 16h, indicating the Detect Media Change Function.
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Change information, as follows:

00h - Diskette change line signal not active.
01h - Invalid drive number.
06h - Change may have occurred.
80h - Drive not ready, or invalid drive.

21.4.18 Set Diskette Type (17h)

The Set Diskette Type disk BIOS function is called to set the data transfer rate for the specified drive.

This function returns an error when accessing a fixed disk.

Input Parameters:

AH - 17h, indicating the Set Diskette Type Function.
AL - Diskette type, as follows:

00h - Reserved.
01h - 360KB diskette in 360KB drive.
02h - 360KB diskette in 1.2MB drive.
03h - 1.2MB diskette in 1.2MB drive.
04h - 720KB diskette in 720KB drive.

DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Disk status code (00h if success).

21.4.19 Set Media Type for Format (18h)

The Set Media Type for Format disk BIOS function is called to set the media type for the specified drive in order for a FORMAT operation to proceed.

This function returns an error when accessing a fixed disk.

Input Parameters:

AH - 18h, indicating the Set Media Type Function.
CH - Maximum track number (0-based).
CL - Maximum sectors per track (0-based).
DL - Drive number.

Output Parameters:

CY - Set if failure, else clear if success.
AH - Status code, as follows:

00h - Track/sector type supported.
0ch - Media type unknown.
80h - No diskette in drive.
xxh - Disk status code.

ES:DI - Address of diskette parameter table for specified track/sector combination.

21.5 INT 14h, Serial I/O Services

This section explains the serial BIOS application program interface (API). The serial BIOS is called through software interrupt 14H. Services are provided to initialize the serial ports, send characters, receive characters, and read the serial port status.

21.5.1 Initialize Serial Port (00h)

The Initialize Serial Port serial BIOS function is called to initialize the communications parameters for a specific serial port. For greater control over serial port initialization, use the extended serial port initialization function (04h).

Input Parameters:

AH - 00h, indicating the Initialize Serial Port Function.
AL - Serial port initialization parameters:

bbb00000b - Baud rate, as follows:

000b - 110 baud.
001b - 150 baud.
010b - 300 baud.
011b - 600 baud.
100b - 1200 baud.
101b - 2400 baud.
110b - 4800 baud.
111b - 9600 baud.

000pp000b - Parity, as follows:

00b - No parity.
01b - Odd parity.
10b - No parity.
11b - Even parity.

00000s00b - Stop bits, as follows:

0b - One stop bit.
1b - Two stop bits.

00000011b - Data bits, as follows:

10b - 7 data bits.
11b - 8 data bits.

DX - Serial port number (0=COM1, 1=COM2, 2=COM3, 3=COM4).

Output Parameters:

AH - Line status register, as follows:

10000000b - Timeout error occurred.
01000000b - Transmitter shift & holding register empty.
00100000b - Transmitter holding register empty.
00010000b - Break interrupt occurred.
00001000b - Framing error occurred.
00000100b - Parity error occurred.
00000010b - Data overrun error occurred.
00000001b - Data ready.

AL - Modem status register, as follows:

10000000b - Data carrier detect.
01000000b - Ring indicator.
00100000b - Data set ready.
00010000b - Clear to send.
00001000b - Delta data carrier select.
00000100b - Trailing edge ring indicator.
00000010b - Delta data set ready.
00000001b - Delta clear to send.

21.5.2 Send Character (01h)

The Send Character serial BIOS function is called to send a byte over the specified serial communications channel.

Input Parameters:

AH - 01h, indicating the Send Character Function.
AL - Character to send.
DX - Serial port number (0=COM1, 1=COM2, 2=COM3, 3=COM4).

Output Parameters:

AH - Line status register, as follows:

10000000b - Timeout error occurred.
01000000b - Transmitter shift & holding register empty.
00100000b - Transmitter holding register empty.
00010000b - Break interrupt occurred.
00001000b - Framing error occurred.
00000100b - Parity error occurred.
00000010b - Data overrun error occurred.
00000001b - Data ready.

AL - Character sent.

21.5.3 Receive Character (02h)

The Receive Character serial BIOS function is called to receive a byte over the specified serial communications channel.

Input Parameters:

AH - 02h, indicating the Receive Character Function.
DX - Serial port number (0=COM1, 1=COM2, 2=COM3, 3=COM4).

Output Parameters:

AH - Line status register, as follows:

10000000b - Timeout error occurred.
01000000b - Transmitter shift & holding register empty.
00100000b - Transmitter holding register empty.
00010000b - Break interrupt occurred.
00001000b - Framing error occurred.
00000100b - Parity error occurred.
00000010b - Data overrun error occurred.
00000001b - Data ready.

AL - Character received.

21.5.4 Read Serial Port Status (03h)

The Read Serial Port Status serial BIOS function is called to read the modem status register and the line status register for the specified serial port.

Input Parameters:

AH - 03h, indicating the Read Serial Port Status Function.
DX - Serial port number (0=COM1, 1=COM2, 2=COM3, 3=COM4).

Output Parameters:

AH - Line status register, as follows:

10000000b - Timeout error occurred.
01000000b - Transmitter shift & holding register empty.
00100000b - Transmitter holding register empty.
00010000b - Break interrupt occurred.
00001000b - Framing error occurred.
00000100b - Parity error occurred.
00000010b - Data overrun error occurred.
00000001b - Data ready.

AL - Modem status register, as follows:

10000000b - Data carrier detect.
01000000b - Ring indicator.
00100000b - Data set ready.
00010000b - Clear to send.
00001000b - Delta data carrier select.
00000100b - Trailing edge ring indicator.
00000010b - Delta data set ready.
00000001b - Delta clear to send.

21.5.5 Extended Initialize Serial Port (04h)

The Extended Initialize Serial Port serial BIOS function is called to initialize the communications parameters for a specific serial port, with more architectural room for handling faster ports, up to 115 kbaud.

Note that not all serial ports can be programmed to accommodate all baud rates or protocols. See the CPU Personality Module for your target's CPU class to determine if there are restrictions when initializing on-board CPU serial ports.

Input Parameters:

AH - 04h, indicating the Extended Initialize Serial Port Function.
AL - 00h if no break signal, 01h if break signal.
BH - Parity, as follows:

00h - no parity.
01h - odd parity.
02h - even parity.
03h - stick parity odd.
04h - stick parity even.

BL - Stop bits, as follows:

00h - 1 stop bit.
01h - 2 stop bits if data length is 6, 7, or 8 bits.
02h - 1.5 stop bits if data length is 5 bits.

CH - Data length, as follows:

00h - 5 bits.
01h - 6 bits.
02h - 7 bits.
03h - 8 bits.

CL - Baud rate, as follows:

00h - 110 baud.
01h - 150 baud.
02h - 300 baud.
03h - 600 baud.
04h - 1200 baud.

05h - 2400 baud.
06h - 4800 baud.
07h - 9600 baud.
08h - 19.2 kbaud.
09h - 38.4 kbaud.
0ah - 56 kbaud.
0bh - 115 kbaud.

DX - Serial port number (0=COM1, 1=COM2, 2=COM3, 3=COM4).

Output Parameters:

AH - Line status register, as follows:

10000000b - Timeout error occurred.
01000000b - Transmitter shift & holding register empty.
00100000b - Transmitter holding register empty.
00010000b - Break interrupt occurred.
00001000b - Framing error occurred.
00000100b - Parity error occurred.
00000010b - Data overrun error occurred.
00000001b - Data ready.

AL - Modem status register, as follows:

10000000b - Data carrier detect.
01000000b - Ring indicator.
00100000b - Data set ready.
00010000b - Clear to send.
00001000b - Delta data carrier select.
00000100b - Trailing edge ring indicator.
00000010b - Delta data set ready.
00000001b - Delta clear to send.

21.6 INT 15h, General Services

This section explains the general services BIOS application program interface (API). The general services BIOS is called through software interrupt 15H. Services are provided to support multitasking for device waits, protected mode functions, access to system configuration information, and access the Advanced Power Management services.

Additional services are supported to handle CMOS RAM reading and writing, setting the BIOS **CurrIo** variable to affect console redirection, Flash programming, and returning the EMBEDDED BIOS version number.

21.6.1 Query Port 92h A20 Gate Capability (24h)

The Query Port 92h A20 Gate Capability BIOS function provides information to the caller about whether the application or operating system can switch the A20 gate with port 92h. This is a legacy function used by HIMEM.SYS.

Input Parameters:

AH - 24h, indicating Query Port 92h A20 Gate Capability Function.
AL - subfunction, as follows:
    01h - Enable A20 gate.
    02h - Disable A20 gate.
    03h - Determine if port 92h support is available.

Output Parameters:

CY - set if failure (no port 92h support), else clear if success.
AH - if failure, 86h.

BX - if subfunction 03h, returns the value 2, indicating support available.

21.6.2 Keyboard Intercept Up-Call (4fh)

The Keyboard Intercept system services BIOS up-call is called by the keyboard BIOS interrupt service routine to allow the operating system or application (client) to receive notice of incoming scan codes (both make and break). If no client is available, then the default handler for this routine returns with the CY flag set.

If the client desires to process the incoming scan code, then it must clear the CY flag after processing the data passed in the AL CPU register. This causes the keyboard BIOS to abort further processing of the scan code other than issuing an EOI to the interrupt controller.

If the client does not wish to process the incoming scan code, then it must set the CY flag before returning. At the client's option, it may elect to modify the scan code passed in the AL CPU register so that the keyboard BIOS processes the input differently. The client is cautioned that this technique can lead to keyboard BIOS failure if non-scan codes are processed; other data, such as status codes, are also passed to this routine.

Input Parameters:

AH - 4fh, indicating the Keyboard Intercept Up-Call.
AL - scan code.

Output Parameters:

CY - set if keyboard BIOS should process scan code in AL, else clear if keyboard BIOS should discard the scan code.
AL - scan code as updated by client.

21.6.3 APM Installation Check (5300h)

The Advanced Power Management Installation Check BIOS function is called to determine if Advanced Power Management services are enabled, and if so, which version of the specification it supports.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 00h, indicating Installation Check Subfunction.
BX - 0000h, indicating system BIOS.

Output Parameters:

CY - set if failure, else clear if success.
AH - if failure, 86h.
AH - if success, major version number in BCD.
AL - minor version number in BCD.
BH - ASCII "P" character.
BL - ASCII "M" character.
CX - capabilities flags, as follows:

bit 0 = 1 if 16-bit protected mode interface supported.
bit 1 = 1 if 32-bit protected mode interface supported.
bit 2 = 1 if CPU Idle call slows processor clock speed.
bit 3 = 1 if BIOS Power Management is disabled.

21.6.4 APM Interface Connect (5301h)

The Advanced Power Management Interface Connect BIOS function is called to establish the cooperative interface between the caller and the system BIOS. Before the interface is established, the system BIOS will provide its own power management functionality as implemented by the OEM. Once the interface is established (connected), the system BIOS and the caller will coordinate power management activities together.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 01h, indicating Interface Connect Subfunction.
BX - 0000h, indicating system BIOS.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

02h - interface connection already in effect.
09h - unrecognized device ID.
86h - APM not supported.

21.6.5 APM Protected Mode 16-Bit Interface Connect (5302h)

The Advanced Power Management Protected Mode 16-Bit Interface Connect BIOS function is called to initialize an optional 16-bit protected mode interface between the caller and the system BIOS. This interface allows a protected mode caller to invoke the system BIOS functions

without the need to first switch into real or virtual-86 mode. A caller that does not operate in protected mode may not need to use this call. This function establishes a 16-bit protected mode interface, but this function must be invoked in either real or virtual-86 mode using the INT 15h interface.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 02h, indicating 16-Bit P/M Interface Connect Subfunction.
BX - 0000h, indicating system BIOS.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

02h - interface connection already in effect.
05h - 16-bit protected mode interface already established.
06h - 16-bit protected mode interface not supported.
09h - unrecognized device ID.
86h - APM not supported.

AX:BX - if success, 16:16 protected mode entrypoint supporting APM requests for the system BIOS.
CX - if success, 16-bit data selector used by APM entrypoint.

21.6.6 APM Protected Mode 32-Bit Interface Connect (5303h)

The Advanced Power Management Protected Mode 32-Bit Interface Connect BIOS function is called to initialize an optional 32-bit protected mode interface between the caller and the system BIOS. This interface allows a protected mode caller to invoke the system BIOS functions without the need to first switch into real or virtual-86 mode. A caller that does not operate in protected mode may not need to use this call. This function establishes a 32-bit protected mode interface, but this function must be invoked in either real or virtual-86 mode using the INT 15h interface.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 03h, indicating 32-Bit P/M Interface Connect Subfunction.
BX - 0000h, indicating system BIOS.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

02h - interface connection already in effect.
07h - 32-bit protected mode interface already established.
08h - 32-bit protected mode interface not supported.
09h - unrecognized device ID.

86h - APM not supported.

AX:EBX - if success, 16:32 protected mode entrypoint supporting APM requests for the system BIOS.
CX - if success, 16-bit code segment selector used by APM entrypoint.
DX - if success, 16-bit data selector used by APM entrypoint.

21.6.7 APM Interface Disconnect (5304h)

The Advanced Power Management Interface Disconnect BIOS function is called to break the cooperative interaction between the system BIOS and the caller, and in the process restores the system BIOS default functionality. Any protected mode connection set-up by the protected mode interface connection functions are invalidated by this call.

Even though this call returns control of power management to the system BIOS, the parameter values (timer values, enable/disable settings, etc.) in effect at the time of the disconnect will remain in effect.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 04h, indicating Interface Disconnect Subfunction.
BX - 0000h, indicating system BIOS.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

03h - interface not connected.
09h - unrecognized device ID.
86h - APM not supported.

21.6.8 APM CPU Idle (5305h)

The Advanced Power Management CPU Idle BIOS function is called to inform the system BIOS that the system is currently idle, and that processing should be suspended until the next system event (typically an interrupt) occurs. This function allows the system BIOS to take some implementation specific power saving action, such as a CPU HLT instruction or stopping the CPU clock.

In cases where an interrupt causes the system to leave the idle state, the interrupt may or may not have been serviced when the BIOS returns from the CPU Idle request.

if interrupts are serviced from within the CPU Idle function, the interrupt handler must return to the BIOS when the interrupt processing is completed. The caller cannot use its knowledge of being in the idle state to retain control from an interrupt handler. For example, some system implementations may slow the processor CPU clock rate before waiting on an interrupt, and restore the normal clock rate after the interrupt is serviced but before returning from the idle call.

When the caller regains control from the system BIOS idle routine, it should determine if there is actually any processing to be performed, and reissue the CPU idle call if not. If the caller is a multitasking supervisor, it may be necessary for it to dispatch its applications, allowing them to check for activity that they should then perform.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 05h, indicating CPU Idle Subfunction.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

86h - APM not supported.

21.6.9 APM CPU Busy (5306h)

The Advanced Power Management CPU Busy BIOS function is called to inform the system BIOS that the system is now busy and processing should continue at full speed. Some system implementations may only be able to slow the CPU clock rate and return in response to the CPU Idle request (see function 5305h). It is expected that the system BIOS will restore the CPU clock reate to its normal rate when it recognizes increased system activity (typically interrupt-driven), but it may be unable to do so when interrupts are hooked by external software that does not invoke BIOS routines.

In cases where this is possible, the caller can ensure the system is running at full speed by invoking the CPU Busy function. Upon return from the APM Installation Check call, bit 2 of the CX CPU register indicates that the system BIOS slows the CPU clock rate during the CPU Idle call. The caller can use this bit to determine if it wishes to call CPU Busy before executing code that it wants to run at full speed.

Calling CPU Busy when the systme is already operating at full speed is discouraged due to the unnecessary call overhead, but the operation is allowed and iwll have no unexpected side effects.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 06h, indicating CPU Busy Subfunction.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

86h - APM not supported.

21.6.10 APM Set Power State (5307h)

The Advanced Power Management Set Power State BIOS function is called to place the system in the requested state. The system BIOS only responds to power device ID = 0001h (system BIOS).

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 07h, indicating Set Power State Subfunction.
BX - 0001h, indicating system BIOS.
CX - System State ID, as follows:

0000h - Ready (not supported for device ID 0001h).
0001h - Standby.
0002h - Suspend.
0003h - Off (not supported for device ID 0001h).

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

01h - power management functionality disabled.
09h - unrecognized device ID.
0ah - parameter valud in CX out of range.
60h - cannot enter requested state.
86h - APM not supported.

21.6.11 APM Enable/Disable APM Functionality (5308h)

The Advanced Power Management Enable/Disable APM Functionality BIOS function is called to enable or disable all APM automatic power down functionality. When disabled, the system BIOS will not automatically power down devices, enter the standby state, enter the suspended state, or take power saving steps in response to CPU Idle calls. In addition, many system BIOS functions will be disabled and will return error 01h, power management functionality disabled.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 08h, indicating Enable/Disable APM Functionality Subfunction.
BX - ffffh, "enable/disable all power management".
CX - 0 to disable, 1 to enable.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

01h - power management functionality disabled.
09h - unrecognized device ID.
0ah - parameter valud in CX out of range.
86h - APM not supported.

21.6.12 APM Restore APM Power-On Defaults (5309h)

The Advanced Power Management Restore APM Power-On Defaults BIOS function is called to instruct the BIOS to reinitialize all of its power-on APM defaults.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 09h, indicating Restore APM Power-On Defaults Subfunction.
BX - ffffh, "all power management".

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

09h - unrecognized device ID.
86h - APM not supported.

21.6.13 APM Get Power Status (530ah)

The Advanced Power Management Get Power Status BIOS function is called to return the system BIOS's current power management status.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 0ah, indicating Get Power Status Subfunction.
BX - 0001h.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

09h - unrecognized device ID.
86h - APM not supported.

BH - if success, A/C line status as follows:

00h - off-line.
01h - on-line.
ffh - unknown.

BL - if success, battery status as follows:

00h - high.
01h - low.

02h - critical.
03h - charging.
ffh - unknown.

CL - if success, remaining battery life, as follows:

0 - 100% = # of full charge
ffh - unknown.

21.6.14 APM Get APM Event (530bh)

The Advanced Power Management Get APM Event BIOS function is called to return the next pending PM event, or indicates if no PM events are pending.

Input Parameters:

AH - 53h, indicating an Advanced Power Management Function.
AL - 0bh, indicating Get APM Event Subfunction.

Output Parameters:

CY - set if failure, else clear if success.
AH - error code, as follows:

03h - interface connection not established.
86h - APM not supported.

BX - if success, PM event code, as follows:

01h - system standby request notification
02h - system suspend request notification
03h - normal resume system notification
04h - critical resume system notification
05h - battery low notification

21.6.15 System Request Key (58h)

The System Request Key BIOS Up-Call is called by the keyboard BIOS interrupt service routine to allow the operating system or application (client) to receive notice that the SysReq key has been pressed or released.

If no client intercepts the up-call, then it will be handled by the default system BIOS handler, which clears the CY flag and sets the AH CPU register to 00h.

Input Parameters:

AH - 85h, indicating a System Request Key Up-Call.
AL - 00h if key pressed, else 01h if key released.

Output Parameters:

CY - clear.
AH - 00h.

21.6.16 Wait Function (86h)

The Wait Function BIOS function is called to delay for a specified number of microseconds so that applications can perform fine timing.

This function must be carefully tuned by the OEM during the adaptation process; it should not be assumed to be accurate until the OEM has done this tuning. The tuning is handled in BPM routine **BoardDelayUsec**.

Input Parameters:

AH - 86h, indicating the Wait Function.
CX:DX - 32-bit number of microseconds to wait.

Output Parameters:

CY - set if failure, else clear if success.

21.6.17 Move Extended Memory Block (87h)

The Move Extended Memory Block BIOS function is called to perform a memory copy using the protected mode capabilities of targets that support protected mode operation.

The memory transfer is specified in the form of a GDT whose real-mode address is passed to the function. Also passed to the function is a number of 16-bit words to copy (beware, this function cannot transfer an odd number of bytes).

Input Parameters:

AH - 87h, indicating the Move Extended Memory Block Function.
CX - number of 16-bit words to copy.
ES:SI - 16:16 real-mode address of GDT describing source and destination addresses for the copy process, formatted as follows:

GDT entry #0 - dummy entry, should be all zeroes.
GDT entry #1 - pointer to GDT.
GDT entry #2 - source buffer.
GDT entry #3 - destination buffer.
GDT entry #4 - reserved by BIOS, do not initialize.
GDT entry #5 - reserved by BIOS, do not initialize.

The format of a GDT and its entries is specified in Intel documentation and is beyond the scope of this manual.

Output Parameters:

CY - set if failure, else clear if success.
AH - status code, as follows:

00h - no error.
01h - RAM parity error occurred during copy.
02h - CPU exception occurred during copy.
03h - gate A20 operation failed.
86h - protected mode services not available.

21.6.18 Extended Memory Size (88h)

The Extended Memory Size BIOS function is called to return the amount of extended memory (that RAM available to the application above the 1MB physical address boundary).

Input Parameters:

AH - 88h, indicating the Extended Memory Size Function.

Output Parameters:

CY - set if failure, else clear if success.
AH - status code, as follows:

86h - protected mode services not available.

AX - if success, extended memory size in 1KB units.

21.6.19 Switch To Protected Mode (89h)

The Switch To Protected Mode BIOS function is called to switch the mode of the processor and establish a GDT for addressability for the remainder of the system's operation.

Input Parameters:

AH - 89h, indicating the Switch To Protected Mode Function.
BH - index into the IDT specifying the base of the first 8 hardware interrupts.
BL - index into the IDT specifying the base of the second 8 hardware interrupts.
ES:SI - 16:16 real-mode address of GDT built by caller, as follows:

GDT entry #0 - dummy entry, should be all zeroes.
GDT entry #1 - pointer to GDT.
GDT entry #2 - pointer to IDT.
GDT entry #3 - pointer to data segment.
GDT entry #4 - pointer to extra segment.
GDT entry #5 - pointer to stack segment.
GDT entry #6 - pointer to code segment.
GDT entry #7 - additional descriptor for BIOS scratch.

The format of a GDT and its entries is specified in Intel documentation and is beyond the scope of this manual.

Output Parameters:

CY - set if failure, else clear if success.
AH - status code, as follows:

00h - no error.
01h - RAM parity error occurred during copy.
02h - CPU exception occurred during copy.
03h - gate A20 operation failed.
86h - protected mode services not available.

21.6.20 Device Busy Up-Call (90h)

The Device Busy BIOS up-call is called by various device management modules within the system BIOS to allow the operating system or application (client) to receive notice of an impending spin-loop within the BIOS to wait for a device to perform a mechanical function. If no client is available, then the default handler for this routine returns with the CY flag set.

If a client is available, it can decide to perform other activities and return when the Device Interrupt up-call is received. When it takes this option, it returns from the Device Busy BIOS up-call with the CY flag cleared. When the BIOS detects that the CY flag is cleared, it does not enter the anticipated spinloop, but instead assumes that the operation has completed or has timed-out.

If the client decides to ignore the Devicy Busy notification and let the BIOS enter its spin-loop, then it returns from the Device Busy BIOS up-call with the CY flag set. This causes the BIOS to continue as though the client had never hooked the INT 15h service in the first place.

Input Parameters:

AH - 90h, indicating a Device Busy Up-Call.
AL - Device type code, as follows:

00h - hard disk drive.
01h - floppy disk drive.
02h - keyboard.
03h - PS/2 mouse.
80h - network.
fch - hard disk reset.
fdh - floppy disk drive motor.
feh - printer.

ES:BX - if AL=80h-ffh, then this register pair points to a request block that is device-dependent.

Output Parameters:

CY - set if caller should enter spinloop, else clear if caller should avoid spinloop and assume device operation has completed or has timed-out.

21.6.21 Device Interrupt Up-Call (91h)

The Device Interrupt BIOS up-call is called by various device management modules within the system BIOS to allow the operating system or application (client) to receive notice of a peripheral device's completion of some event, such as a seek of a disk drive. This allows the client to return control to the BIOS if it transferred control to another task in response to the Device Busy up-call associated with this Device Interrupt up-call. If no client is available, then the default handler for this routine returns with the CY flag set.

If a client is available, it can decide to reschedule the task that was blocked waiting for the completion of the device operation associated with the previous Device Busy up-call. Regardless of the client's decision to perform this action, the BIOS will continue execution of its code upon return from this function without inspecting the CY flag.

Input Parameters:

AH - 91h, indicating a Device Interrupt Up-Call.
AL - Device type code, as follows:

00h - hard disk drive.
01h - floppy disk drive.
02h - keyboard.
03h - PS/2 mouse.
80h - network.
fch - hard disk reset.
fdh - floppy disk drive motor.
feh - printer.

ES:BX - if AL=80h-ffh, then this register pair points to a request block that is device-dependent.

Output Parameters:

none.

21.6.22 Read/Write CMOS RAM Cell (A0h)

The Read/Write CMOS RAM Cell BIOS function is called by application software to access CMOS RAM cells in a hardware-independent manner. This is useful when the application must run on hardware that may not use the ISA-standard ports 70h and 71h for accessing this hardware.

Input Parameters:

AH - A0h, indicating the Read/Write CMOS RAM Cell Function.
AL - 00h for a read operation, or 01h for a write operation.
BL - specifies the CMOS RAM index to read or write.

BH - for writes only, specifies the value to be written.

Output Parameters:

CY - clear if success, else set if failure.
AL - for reads only, contains the value that was read.
AH - status code, as follows:

00h - no error.
86h - not supported by BIOS configuration.

21.6.23 Set Console I/O Redirection (A1h)

The Set Console I/O Redirection BIOS function is called by application software to specify the device that will be used by the BIOS to redirect console input (INT 16h) and console output (INT 10h). This feature is only available in BIOS adaptations that provide for console redirection.

Console I/O can be redirected to the standard keyboard and screen with the device value, 0. Other values, such as 1, 2, 3, and so on, specify a COM port number that specifies the serial port that will be used. For example, the value 2 specifies that output will be redirected over the serial line attached to COM2.

Input Parameters:

AH - A1h, indicating the Set Console I/O Redirection Function.
BX - specifies the new console device. The value 0 indicates the standard keyboard and screen, and nonzero values indicate the COM port number (starting with 1 for COM1) to be used as a console redirection device.

Output Parameters:

CY - clear if success, else set if failure.
AH - status code, as follows:
00h - no error.
86h - not supported by BIOS configuration.

21.6.24 Get Embedded BIOS Version (A3h)

The Get Embedded BIOS Version BIOS function is called by application software to return the major and minor version codes of the underlying implementation of Embedded BIOS. This allows application software to determine if it can use specific features supported by the BIOS.

Input Parameters:

AH - A3h, indicating the Get Embedded BIOS Version Function.

Output Parameters:

CY - clear if success, else set if failure.
AL - for successful returns, the minor BIOS version (i.e., 3 for version 4.3).

AH - for successful returns, the major BIOS version (i.e., 4 for version 4.3).
AH - if failure, status code, as follows:

86h - not supported by BIOS configuration.

21.6.27 Return System Configuration (C0h)

The Return System Configuration BIOS function is called to return the address of the System Configuration Table (SCT), as defined in Chapter 3. This table reveals the BIOS's support for various hardware features.

Input Parameters:

AH - C0h, indicating the Return System Configuration Function.

Output Parameters:

CY - clear if success, else set if failure.
AH - status code, as follows:

00h - no error.
86h - SCT not supported by BIOS configuration.

ES:BX - if success, 16:16 address of SCT data structure.

21.6.28 Return Extended BIOS Data Area (C1h)

The Return Extended BIOS Data Area BIOS function is called to return the 16-bit segment address of the EMBEDDED BIOS Extended BIOS Data Area, located in the top 1KB of low memory. The format of this area is General Software-proprietary.

Input Parameters:

AH - C1h, indicating the Return Extended BIOS Data Area Function.

Output Parameters:

CY - clear if success, else set if failure.
AH - status code, as follows:

00h - no error.
86h - function not supported.

ES - if success, segment address of Extended BIOS Data Area.

21.6.29 PS/2 Mouse Request (C2h)

The PS/2 Mouse Request BIOS function is called to process an application or operating system request to read the status of, or control the operation of, the PS/2 mouse.

Input Parameters:

AH - C2h, indicating the PS/2 Mouse Request Function.
AL - subfunction, as follows:

00h - Enable/Disable mouse.
01h - Reset mouse.
02h - Set sample rate.
03h - Set resolution.
04h - Get mouse type.
05h - Initialize mouse interface.
06h - Get mouse status/set scaling factor.
07h - Register callout address.

BH - sub-subfunction code or parameter for subfunction.

Output Parameters:

CY - clear if success, else set if failure.
AH - status code, as follows:

00h - no error.
01h - invalid subfunction code (code in AL).
02h - invalid input value (value in BH).
03h - I/O communications error.
04h - resend status received from mouse.
05h - no callout address registered.
86h - function not supported (if **OPTION_SUPPORT_PS2MOUSE** disabled).

21.6.30 Watchdog Timer Control (C3h)

The Watchdog Timer Control BIOS function is called to enable or disable the watchdog timer, if available. When the watchdog timer is enabled by this function, the value passed in the BX CPU register is used as a countdown value for the timer. When the timer reaches 0, it resets the system with a warm boot.

If the timer is running and the disable subfunction is called, then the timer stops without resetting the system.

If the timer is running and the enable subfunction is called, then the timer restarts with the specified value without resetting the system.

If the timer is stopped and the disable function is called, no operation is performed.

If the timer is stopped and the enable function is called, then the timer restarts with the specified value without resetting the system.

Input Parameters:

AH - C3h, indicating the Watchdog Timer Control Function.

AL - 00h to disable timer, 01h to enable timer.
BX - if enabling, fail-safe timer value (units unspecified).

Output Parameters:

CY - clear if success, else set if failure.
AH - status code, as follows:

00h - no error.
86h - function not supported.

21.6.31 Debugger Breakpoint (D0h)

The Debugger Breakpoint BIOS function is called to perform a breakpoint into the debugger on systems where an operating system is loaded, and that operating system has revectored INT 3 to its own private dummy routine, otherwise making the BIOS debugger inaccessible.

Input Parameters:

AH - D0h, indicating the Debugger Breakpoint Function.

Output Parameters:

CY - clear if success, else set if failure.
AH - status code (if error), as follows:

86h - function not supported.

21.6.32 Flash Programming (E0h)

The Flash Programming BIOS function is called to perform any of a number of operations on the Flash array supported by the underlying BIOS. This function is General Software-proprietary.

There are four subfunctions, all of which require that the (DI:SI) register pair contain the 32-bit media address of Flash memory being manipulated. If the operation is lock or unlock, then (DI:SI) can point to any byte within the block to be locked or unlocked.
If the operation is a read or write, then this register pair points to the first byte in a contiguous area of Flash to be read or written.

For read and write operations, the (ES:BX) register pair points to a user buffer where the data read from Flash will be stored for read operations, or that will contain data to be written to Flash for write operations. The (CX) register is used to specify the number of bytes to transfer.

This function requires that Flash programming support (**OPTION_SUPPORT_MCL**) be enabled by the OEM adaptation. If this support is not enabled, this function will return CY set and AH=86h.

Input Parameters:

AH - E0h, indicating the Flash Programming Function.

AL - Subfunction, as follows:

00h - Lock block function.
01h - Erase block function.
02h - Read block function.
03h - Write block function.

DI:SI - 32-bit media address of Flash memory area.
CX - bytes to read or write (unused for lock or erase).

ES:BX - 16:16 real-mode address of a user buffer where information is transferred *from* on a write operation, or where it is transferred *to* on a read operation.

Output Parameters:

CY - clear if success, else set if failure.
AH - status code, as follows:

00h - no error.
86h - function not supported.

21.7 INT 16h, Keyboard Services

This section explains the keyboard BIOS application program interface (API). The keyboard BIOS is called through software interrupt 16H. Services are provided to read keystrokes from the keyboard typeahead buffer, peek at the next keystroke in the typeahead buffer, and get the status of the shift keys on the keyboard.

Additional services are provided by the INT 16h service module to set the CPU speed and manipulate the system's cache. These services are historically bound to the INT 16h service because they were first managed by the 8042 keyboard controller.

21.7.1 Read Keyboard Input (00h)

The Read Keyboard Input keyboard BIOS function is called to read a keystroke from the keyboard device, waiting until a keystroke arrives if one is not present. The scan code of the keystroke is returned in (AH), and the ASCII code is returned in (AL). An exception exists for function keys and ALT keys; in this case, the ASCII code returned is zero, and the scan code in (AH) is used to determine which function or ALT key was read from the keyboard.

Input Parameters:

AH - 00h, indicating the Read Keyboard Input Function.

Output Parameters:

AH - Scan code of the returned keystroke.
AL - ASCII code for the returned keystroke.

21.7.2 Return Keyboard Status (01h)

The Return Keyboard Status keyboard BIOS function is called to peek at the status of the typeahead buffer, to determine if a keystroke is waiting to be read. If not, the zero flag (ZF) is cleared, so that a JZ instruction after the INT 16H instruction would not be taken. If a keystroke is waiting to be read, then ZF is set, so that a JZ instruction after the INT 16H instruction would be taken. In the latter case, the scan code and character code are returned in (AH) and (AL), respectively. If a character is found, this function returns a copy of it but does not remove it from the typeahead buffer. The application can call function 00H to remove the character from the typeahead buffer properly.

Input Parameters:

AH - 01h, indicating the Return Keyboard Status Function.

Output Parameters:

ZF - Clear if character ready, else set if not.
AH - Scan code of the returned keystroke.
AL - ASCII code for the returned keystroke.

21.7.3 Return Shift Flag Status (02h)

The Return Shift Flag Status keyboard BIOS function is called to return the status of the shift keys, including the INS key, CAPS LOCK key, NUM LOCK key, SCROLL LOCK key, ALT key, CTRL key, and LEFT and RIGHT SHIFT keys.

Input Parameters:

AH - 02h, indicating the Return Shift Flag Status.

Output Parameters:

AL - Current shift status, in the form of a bit mask, one bit per shift key.

10000000b - INS key is active.
01000000b - CAPS LOCK key is active.
00100000b - NUM LOCK key is active.
00010000b - SCROLL LOCK key is active.
00001000b - ALT key is pressed down.
00000100b - CTRL key is pressed down.
00000010b - LEFT SHIFT key is pressed down.
00000001b - RIGHT SHIFT key is pressed down.

21.7.4 Set Typematic Rate (03h)

The Set Typematic Rate keyboard BIOS function is called to program the typematic delay and rate associated with holding down a key on the keyboard. This keyboard service is not redirectable over serial links.

Input Parameters:

AH - 03h, indicating the Set Typematic Rate Function.
AL - 05h
BH - typematic delay before repeat starts, as follows:

00h - 250 milliseconds.
01h - 500 milliseconds.
02h - 750 milliseconds.
03h - 1,000 milliseconds (1 second).

BL - typematic rate in characters per second, as follows:

00h - 30.0 CPS.	01h - 26.7 CPS.
02h - 24.0 CPS.	03h - 21.8 CPS.
04h - 20.0 CPS.	05h - 18.5 CPS.
06h - 17.1 CPS.	07h - 16.0 CPS.
08h - 23.1 CPS.	09h - 13.3 CPS.
0ah - 12.0 CPS.	0bh - 10.9 CPS.
0ch - 10.0 CPS.	0dh - 9.2 CPS.
0eh - 8.6 CPS.	0fh - 8.0 CPS.
10h - 7.5 CPS.	11h - 6.7 CPS.
12h - 6.0 CPS.	13h - 5.5 CPS.
14h - 5.0 CPS.	15h - 4.6 CPS.
16h - 4.3 CPS.	17h - 4.0 CPS.
18h - 3.7 CPS.	19h - 3.3 CPS.
1ah - 3.1 CPS.	1bh - 2.7 CPS.
1ch - 2.5 CPS.	1dh - 2.3 CPS.
1eh - 2.1 CPS.	1fh - 2.0 CPS.

Output Parameters:

none.

21.7.5 Push Data to Keyboard (05h)

The Push Data to Keyboard keyboard BIOS function is called to push data (a character and a scan code) into the keyboard typeahead buffer.

Input Parameters:

AH - 05h, indicating the Push Data to Keyboard Function.
CH - scan code to be pushed.
CL - character to be pushed.

Output Parameters:

CY - set if failure, else clear if success.
AL - error code, as follows:

00h - no error.
01h - keyboard buffer full.

21.7.6 Enhanced Read Keyboard (10h)

The Enhanced Read Keyboard keyboard BIOS function is called to read a character and scan code from the keyboard buffer when the BIOS supports an enhanced (101-key) keyboard. There is no advantage to calling this routine over the standard read function; it is provided for compatibility with some versions of DOS that call it.

Input Parameters:

AH - 10h, indicating the Enhanced Read Keyboard Function.

Output Parameters:

AH - 00h scan code or character ID if special character.
AL - ASCII code.

21.7.7 Enhanced Read Keyboard Status (11h)

The Enhanced Read Keyboard Status keyboard BIOS function is called to determine if an enhanced keyboard has a character waiting in its buffer. There is no advantage to calling this routine over the standard read function; it is provided for compatibility with some versions of DOS that call it. This routine does not remove the data from the keyboard buffer; it is a "peek" operation.

Input Parameters:

AH - 11h, indicating the Enhanced Read Keyboard Status Function.

Output Parameters:

ZF - set if no character, else clear if character waiting.

Then, if a character is waiting:

AH - 00h scan code or character ID if special character.
AL - ASCII code.

21.7.8 Enhanced Read Keyboard Flags (12h)

The Enhanced Read Keyboard Flags keyboard BIOS function is called to return the state of the enhanced shift flags maintained by 101-key keyboards. There is limited advantage to calling this routine over the standard read function; it is provided for compatibility with some versions of DOS that call it.

Input Parameters:

AH - 12h, indicating the Enhanced Read Keyboard Flags Function.

Output Parameters:

AX - 16-bit bitmask containing keyboard flags, as follows:

00000000.00000001 - right shift key pressed.
00000000.00000010 - left shift key pressed.
00000000.00000100 - ctrl key pressed.
00000000.00001000 - alt key pressed.
00000000.00010000 - scroll lock is on.
00000000.00100000 - num lock is on.
00000000.01000000 - caps lock is on.
00000000.10000000 - insert mode is on.
00000001.00000000 - left ctrl key is pressed.
00000010.00000000 - left alt key is pressed.
00000100.00000000 - right ctrl key is pressed.
00001000.00000000 - right alt key is pressed.
00010000.00000000 - scroll lock key is pressed.
00100000.00000000 - num lock key is pressed.
01000000.00000000 - caps lock key is pressed.
10000000.00000000 - sysreq key is pressed.

21.7.9 Set CPU Speed (F0h)

The Set CPU Speed BIOS function is called to change the CPU's clocking to either the low or high states.

Not all BIOS adaptations can switch speeds; this is largely a function of the BPM, CPM, and CSPM implementations.

Input Parameters:

AH - F0h, indicating the Set CPU Speed Function.
AL - speed to set, as follows:

00h - slow.
01h - medium.
02h - fast.

Output Parameters:

none.

21.7.10 Get CPU Speed (F1h)

The Get CPU Speed BIOS function is called to read the CPU's speed as set with the Set CPU Speed function.

Not all BIOS adaptations can switch speeds; this is largely a function of the BPM, CPM, and CSPM implementations.

Input Parameters:

AH - F1h, indicating the Get CPU Speed Function.

Output Parameters:

AL - current CPU speed, as follows:

00h - slow.
01h - medium.
02h - fast.

21.7.11 Read Cache Status (F400h)

The Read Cache Status BIOS function is called to request the status of the external cache as supported by the BIOS.

Not all BIOS adaptations can manipulate the cache; this is largely a function of the BPM, CPM, and CSPM implementations.

Input Parameters:

AH - F4h, indicating the Cache Control Function.
AL - 00h, indicating the Read Cache Status Subfunction.

Output Parameters:

AH - cache status, as follows:

not modified - no cache status is available.
E2h - successful, information returned.

AL - cache controller status, as follows:

00h - cache controller not present.
01h - cache memory enabled.
02h - cache memory disabled.

CX - cache memory size, as follows:

Bit 15 - 1 if information invalid, else 0 if valid.
Bits 14 through 0 - cache memory size in KB.

DH - cache write strategy, as follows:

Bit 7 - 1 if information invalid, else 0 if valid.
Bits 6 through 1 - set to 0's.
Bit 0 - 0 if write-through, else 1 if write-back.

DL - cache type, as follows:

Bit 7 - 1 if information invalid, else 0 if valid.
Bits 6 through 1 - set to 0's.
Bit 0 - 0 if direct-mapped else 1 if two-way set associative.

21.7.12 Enable Cache (F401h)

The Enable Cache BIOS function is called to enable the external cache controller as supported by the BIOS.

Not all BIOS adaptations can manipulate the cache; this is largely a function of the BPM, CPM, and CSPM implementations.

Input Parameters:

AH - F4h, indicating the Cache Control Function.
AL - 01h, indicating the Enable Cache Subfunction.

Output Parameters:

AH - cache status, as follows:

not modified - no cache status is available.
E2h - successful, operation performed.

21.7.13 Disable Cache (F402h)

The Disable Cache BIOS function is called to disable the external cache controller as supported by the BIOS.

Not all BIOS adaptations can manipulate the cache; this is largely a function of the BPM, CPM, and CSPM implementations.

Input Parameters:

AH - F4h, indicating the Cache Control Function.
AL - 02h, indicating the Disable Cache Subfunction.

Output Parameters:

AH - cache status, as follows:

not modified - no cache status is available.

E2h - successful, operation performed.

21.8 INT 17h, Parallel I/O Services

This section explains the parallel BIOS application program interface (API). The parallel BIOS is called through software interrupt 17H. Services are provided to write a character to the parallel port, initialize the printer attached to a parallel port, and read the status of a printer attached to a parallel port.

21.8.1 Write Character (00h)

The Write Character parallel BIOS function is called to write a character over the parallel port to a printer or other parallel device.

Input Parameters:

AH - 00h, indicating the Write Character Function.
AL - Character to print.
DX - Parallel port number (0=LPT1, 1=LPT2, 2=LPT3).

Output Parameters:

AH - Printer status, as follows:

10000000b - Printer not busy.
01000000b - Acknowledgement.
00100000b - Out of paper.
00010000b - Printer selected.
00001000b - I/O error occurred.
00000100b - Reserved.
00000010b - Reserved.
00000001b - Timeout error occurred.

21.8.2 Initialize Printer (01h)

The Initialize Printer parallel BIOS function is called to initialize an attached print device. It does this by pulsing the reset line on the parallel interface.

Input Parameters:

AH - 01h, indicating the Initialize Printer Function.
DX - Parallel port number (0=LPT1, 1=LPT2, 2=LPT3).

Output Parameters:

AH - Printer status, as follows:

10000000b - Printer not busy.
01000000b - Acknowledgement.

00100000b - Out of paper.
00010000b - Printer selected.
00001000b - I/O error occurred.
00000100b - Reserved.
00000010b - Reserved.
00000001b - Timeout error occurred.

21.8.3 Read Printer Status (02h)

The Read Printer Status parallel BIOS function is called to read the status lines attached driven by a parallel-mode printer attached to the parallel port.

Input Parameters:

AH - 02h, indicating the Read Printer Status Function.
DX - Parallel port number (0=LPT1, 1=LPT2, 2=LPT3).

Output Parameters:

AH - Printer status, as follows:

10000000b - Printer not busy.
01000000b - Acknowledgement.
00100000b - Out of paper.
00010000b - Printer selected.
00001000b - I/O error occurred.
00000100b - Reserved.
00000010b - Reserved.
00000001b - Timeout error occurred.

21.9 INT 1ah, Time Services

This section explains the date/time BIOS application program interface (API). The date/time BIOS is called through software interrupt 1aH. Services are provided to read the system time counter, write the system time counter, read the real time clock, write the read time clock, read the real time clock date, and write the real time clock date.

21.9.1 Read System Timer Count (00h)

The Read System Timer Count date/time BIOS function is called to return the 32-bit number of ticks since last midnight as stored in the BIOS data area.

Input Parameters:

AH - 00h, indicating the Read System Timer Count Function.

Output Parameters:

CY - set if failure, else clear if success.

AH - 00h.
AL - Timer overflow flag:

00h - Time has not overflowed the field.
01h - Time has overflowed the field.

CX:DX - 32-bit number of ticks elapsed since midnight.

21.9.2 Write System Timer Count (01h)

The Write System Timer Count date/time BIOS function is called to store the 32-bit number of ticks since last midnight into the BIOS data area.

Input Parameters:

AH - 01h, indicating the Write System Timer Count Function.
CX:DX - 32-bit number of ticks elapsed since midnight.

Output Parameters:

CY - set if failure, else clear if success.
AH - 00h.

21.9.3 Read Real Time Clock Time (02h)

The Read Real Time Clock Time date/time BIOS function is called to return the time information from the battery-backed real time clock.

Input Parameters:

AH - 02h, indicating the Read RTC Time Function.

Output Parameters:

AH - 00h.
CY - set if RTC update in progress, else clear if success.
CH - Hours in BCD.
CL - Minutes in BCD.
DH - Seconds in BCD.
DL - Daylight savings option:
00h - No daylight savings supported.
01h - Daylight savings supported.

21.9.4 Write Real Time Clock Time (03h)

The Write Real Time Clock Time date/time BIOS function is called to store the time information into the battery-backed real time clock.

Input Parameters:

AH - 03h, indicating the Write RTC Time Function.
CH - Hours in BCD.
CL - Minutes in BCD.
DH - Seconds in BCD.
DL - Daylight savings option:
 00h - No daylight savings supported.
 01h - Daylight savings supported.

Output Parameters:

AH - 00h.
AL - Value written to CMOS 0bh register.
CY - set if RTC update in progress, else clear if success.

21.9.5 Read Real Time Clock Date (04h)

The Read Real Time Clock Date date/time BIOS function is called to return the date information from the battery-backed real time clock.

Input Parameters:

AH - 04h, indicating the Read RTC Date Function.

Output Parameters:

AH - 00h.
CY - set if RTC update in progress, else clear if success.
CH - Century in BCD (19h or 20h).
CL - Year in BCD.
DH - Month in BCD.
DL - Day in BCD.

21.9.6 Write Real Time Clock Date (05h)

The Write Real Time Clock Date date/time BIOS function is called to store the date information into the battery-backed real time clock.

Input Parameters:

AH - 05h, indicating the Write RTC Date Function.
CH - Century in BCD (19h or 20h).
CL - Year in BCD.
DH - Month in BCD.
DL - Day in BCD.

Output Parameters:

AH - 00h.
AL - Value written to CMOS 0bh register.

CY - set if RTC update in progress, else clear if success.

21.9.7 PCI Services (B1h)

The PCI Services BIOS function is called to make a PCI function request. The PCI API is well-documented by the PCI Consortium and its operation is beyond the scope of this section.

Input Parameters:

AH - b1h, indicating the PCI Services function.
Others - as defined by PCI Specification.

Output Parameters:

All - as defined by PCI Specification.

PART IV

TROUBLESHOOTING

This part of the EMBEDDED BIOS reference documentation provides guidance on the most frequently-asked technical support problems. You can save lots of time by scanning this section before the adaptation process begins, so that you can steer the adaptation process away from problems to begin with. You can also save time by getting quick solutions to basic problems by looking up the symptoms here.

Chapter 22

TROUBLESHOOTING

This Chapter can be used to find guidance to help you get through problem areas if they occur.

General Software can only help you if you do not modify core EMBEDDED BIOS files. Technical support for OEMs who have modified any of the core files is strictly at General Software's option, and is provided only on a fee basis. If you feel a need to modify core BIOS files, contact your General Software representative to learn about solutions you may not be aware of for handling your specific situation, before you do this. Any warranties, express or implied, associated with the BIOS Adaptation Kit and licensed software are voided by modification of the core files.

If you're having trouble getting the EMBEDDED BIOS system or auxilliary components to build properly, consult the section on "Compiling, Assembling, & Linking" in this chapter.

If you need help deciphering a run-time error message printed by EMBEDDED BIOS, see the "Error Messages" section in this chapter.

For information on how to call EMBEDDED BIOS functions, consult Chapter 21 for technical programming information.

For information on how to diagnose hardware that won't boot, see the "Diagnosing POST" section in this chapter. You'll need to have used procedures similar to these first before contacting General Software for assistance, since our technical support people will be asking you to do these things anyway to get the information necessary to find out why your target doesn't boot. You'll find that, once you have started the diagnosis process yourself, you won't need to make the call.

For 3rd party BIOS specifications, such as Plug-n-Play, El Torito, and so on, search the web. Due to copyright restrictions, General Software cannot supply copies of 3rd party BIOS specifications, even if they are rightfully defacto industry standards.

For information on the PCI portion of your EMBEDDED BIOS adaptation, consult Chapter 12. There are adaptation issues that arise in the project file as well as the Board Personality Module. Also, it is recommended that you have a copy of the PCI Specification 2.1.

For information about how to build graphical images for display in the EMBEDDED BIOS graphical POST system, consult Chapter 11. General Software does not have graphical design services, but does provide some suggestions for obtaining best results when producing graphics for use with this system.

For other problems, we have maintained a list of technical support problems and their solutions, and have listed them under "Advanced Troubleshooting" in this chapter. Be sure to review this section before calling General Software, since you'll likely be referred to this text for those common situations.

If you are able to identify a specific problem setup and behavior that cannot be diagnosed either from the Adaptation Kit materials or by your tool vendor (Borland, Microsoft, or PharLap, for example), call General Software Technical Support. See the section on "Technical Support From General Software" in this chapter for details. Please remember that we are not able to provide technical support for Borland or Microsoft on the installation or use of their tools.

22.1 Compiling, Assembling, & Linking

If you get errors during a build of EMBEDDED BIOS, this section has a checklist to make sure you have the right tools set-up.

If you have MASM 5.x or lower, MSC 7.0 or lower, or BCC 4.0 or lower, you need to upgrade your tools. General Software does not support these early versions of the Borland or Microsoft tools. BCC 3.1's assembler works but the C compiler incorrectly manages loading of DS in some cases. BCC 4.0's assembler and C compiler work, but the linker has a problem linking the BIOS due to its complexity; this is why Borland issued BCC 4.02, containing the linker fix.

If you're using Borland tools such as BCC 4.5, 5.0, etc., including `BCC.EXE`, `TASM.EXE`, `TASMX.EXE`, and `TLINK.EXE`, you will need to set `BORLAND=YES` in your environment. Beware, setting `BORLAND=NO` or `BORLAND=MICROSOFT` has the same effect. If this variable is defined in your environment, the MAKEFILEs used in the EMBEDDED BIOS build will use Borland tools. If it is not present, you get Microsoft compilations.

Don't mix Borland and Microsoft OBJ files; they are incompatible. For example, it is not okay to build part-way with the Borland assembler, and the rest of the way with Microsoft's MASM, because the object file formats are incompatible. Stick with one tool set or the other.

Make sure that you are using the correct tools, and not pieces of one with pieces of another. If you have both Borland and Microsoft tools, make sure that you aren't using Borland libraries with Microsoft compilers or linkers, for example.

The best route to use when building the BIOS is to use BIOStart under Windows. It works under Windows 3.1, Windows 95, and Windows-NT. It should work with future versions of Windows to the extent that those versions are upward compatible with these operating systems.

If you are building EMBEDDED BIOS in a DOS box in Windows, or under real DOS, you *must* use the General Software GSMAKE utility with the MAKEFILEs in the EMBEDDED BIOS builds; make sure that `EBIOS43\TOOLS` is the *first* directory in your path, or at least that it comes *before* another directory with GSMAKE in it. You may use General Software's GSMAKE utility

to read Microsoft MAKEFILEs, and basic Borland MAKEFILEs, although both Borland and Microsoft have various dialects that are not supported by General Software's GSMAKE utility.

If you encounter an "out of compiler space" or "out of heap space" during compilation or assembly, you have just encountered an architectural limitation of your DOS compiler or assembler. This is a common thing to happen when running Microsoft tools with TSRs and network software loaded, because the compiler and assembler need quite a bit of memory to build the EMBEDDED BIOS components.

If you have MASM 6.1 or better (including MASM 6.1, 6.11c, 6.11d, and 6.14), you should set `MASM61=YES` in your environment. If you don't have MASM 6.1x, make sure that the keyword MASM61 is <u>not</u> defined at all in your environment. The MASM61 keyword, when defined, instructs General Software's MAKEFILEs to use special syntax for invoking the assembler that can eliminate some of the out-of-memory problems that MASM encounters when assembling this large code base.

If you have MASM 5.x, you need to upgrade to at least MASM 6.0 or 6.1, which will solve the problem. If you have MSC 5.1 or C 6.0, you need to move to at least MSVC (Microsoft Visual C++), commonly called C 8.0.

MASM 6.10 does not operate properly in the Windows NT 4.0 environment, and may not operate in future versions of Windows. Users experiencing problems with MASM 6.10 are encouraged to use MASM 6.11c, 6.11d, 6.14, or a later version when it becomes available.

If MASM starts emitting errors that seem to cascade upon one another, you have encountered a bug in MASM that causes it to not recognize that it is out of memory, so that it trashes its own tables. You need to reduce the number of `FILES` or `BUFFERS` in `CONFIG.SYS`, remove TSRs, upgrade your DOS and use `HIDOS=YES`, or upgrade your MASM to at least 6.0 or 6.1.

If MSC 5.x-7.x runs out of memory during compilation, you can replace C1.EXE with `C1L.EXE`, by renaming `C1.EXE` to `C1SAVE.EXE`, then renaming `C1L.EXE` to `C1.EXE`. The `C1L.EXE` file is actually a large model compiler front-end module that is a fully-plug-and-play replacement for C1.EXE, except that it handles larger programs. See your `Cxxx\BIN` directory for the various EXE files that come with MSC.

If you encounter "jump out of range" errors in MASM, you need to upgrade to a later (supported) version of MASM, or modify the source code line that causes the error. MASM does not properly generate the jump-around-jump sequence to handle conditional jumps that can't span more than 128 bytes. As EMBEDDED BIOS configurations in OPTIONS.INC and CONFIG.INC can cause these spans to exceed 128 bytes, these errors may occur. If you are unable to upgrade MASM, you can note the line number on which the error occurs in a specific module, and then edit that line in the file by up-casing the jump instruction, and prefixing it with a capital 'L'. Thus:

```
je      becomes     LJE
jne     becomes     LJNE
ja      becomes     LJA
jb      becomes     JB
jae     becomes     JAE
jbe     becomes     JBE
```

22.2 3rd-Party Technical Support

If you need assistance using 3rd-party development tools or libraries, please call the tool vendor before calling General Software. You are apt to receive more current and accurate information about tools directly from the vendors that make them.

For your convenience, we have supplied the most current contact information at the time this manual was printed.

Calling Intel Corporation

Call Intel Corporation if you need documentation for Intel components. General Software does not have copies of Intel literature for redistribution.

PHONES:	(800) 548-4725	(USA & Canada)
	61-2-975-3300	(Sydney, Australia)
	55-11-287-5899	(Sao Paulo, Brazil)
	1-500-4850	(Beijing, PRC)
	(852) 844-4555	(Hong Kong, China)
	91-812-215773	(Bangalore, India)
	0426-48-8770	(Tokyo, Japan)
	(2) 784-8186	(Seoul, Korea)
	(65) 250-7811	(Singapore)
	886-2-5144202	(Taipei, Taiwan)

Calling Microsoft Corporation

For support of Microsoft tools such as MSC, MSVC, MASM, LINK, NMAKE, or Codeview; for technical support associated with Windows NT, Windows CE, or Windows NT Embedded; or for help in determining which library functions are reentrant or which have stack probes, contact Microsoft. The products change enough from release to release that it is not possible for General Software to be aware of the internals of every version of the compiler and C libraries, or the build or installation procedures or device drivers associated with Microsoft's operating systems.

PHONE: (425) 882-8089

COMPUSERVE: GO MICROSOFT

Calling Paradigm Systems

If you need assistance with Paradigm Debug or other Paradigm products, would like to learn about Paradigm's debuggers, or need help in getting Paradigm products to work in the EMBEDDED BIOS environment, call Paradigm for support. Paradigm may be able to help with Borland products, but they are not obliged to offer help for them.

PHONE: (607) 748-5966

EMAIL: support@devtools.com

WEB: http://www.devtools.com

Calling PharLap Software

If you need assistance with the design, development, or debugging of protected mode applications using the PharLap 286|DOS Extender or 386|DOS Extender products, or need assistance using Phar-ASM or LinkLoc, then call PharLap directly for technical assistance.

PHONE: (617) 661-1510

EMAIL: tech-support@pharlap.com

22.3 Technical Support From General Software

If you have purchased your EMBEDDED BIOS Adaptation Kit from an Authorized General Software Distributor, they are responsible for providing first-level support of the product, including installation and general high-level operation such as using BIOStart. A second, more in-depth level of support is obtainable from General Software's Support Centers. These organizations are trained to solve in-depth technical problems with EMBEDDED BIOS, enabing you to receive the fastest response. Your distributor will be able to provide you with contact information for the Support Center in your area. In cases where there is none, 2nd level support is provided directly by General Software in the USA.

The support center for all General Software Embedded BIOS customers in Europe and the Middle East should be contacted via email at eurosupport@gensw.com. This support center does not provide support for products other than General Software Embedded BIOS.

If you purchased the Adaptation Kit directly from General Software, or if your area is not serviced by a General Software Support Center, follow the procedures below.

If you have technical questions concerning the installation or building processes for EMBEDDED BIOS, contact us after you have verified that your environment is set-up properly, and that you have reviewed the "Compiling, Assembling, & Linking" tips in this chapter. If you are not able to resolve the problem, then contact General Software.

If you have technical questions about how to use any of the EMBEDDED BIOS APIs, you can refer to Chapter 21 of this manual. If the chapter contains an error that you can't see how to correct, contact General Software by FAX, jotting down the exact details you need so that it can be properly addressed by an engineer.

If you have a compatibility problem with a desktop program, you should contact General Software by FAX, giving as much detail as possible about the run-time environment (hardware and software) and the program you are running, including if possible, any interaction with the integrated debugger. If you are able to narrow-down an incompatibility to a specific system call or programming method, then we will be able to review the BIOS code to see if the program can be accommodated. We cannot review code without looking for a specific system-oriented problem. We cannot accept third-party software for compatiblity studies as it violates your license with the third-party software vendor.

If you need help getting your embedded application to work with EMBEDDED BIOS, please E-mail, FAX or call General Software directly. Faxed inquiries have been shown to be resolved sooner on the average than telephone inquiries. Premium technical support customers receive

priority over other technical support, and non-premium technical support is processed on a first-come, first-served basis. Keep in mind that many OEMs are producing hardware in the same cycle that you may be; therefore, at the times when you desire the fastest response, others may be requesting the same service. Use premium support to gain access to specific response times.

22.3.1 Support by EMAIL

For technical support inquiries, you may send Email technical support inquiries to the following address:

EUROPEAN CUSTOMER SUPPORT: eurosupport@gensw.com
ALL OTHER CUSTOMER SERVICE EMAIL: support@gensw.com

Email inquiries are generally resolved sooner than FAXed or voice inquiries, because they are the most efficient. By stating the system behavior in writing, these inquiries are concise and are quick to process.

22.3.2 Support by FAX

For technical support inquiries, you may send a FAX, 24 hours/day, 365 days/year (subject to periodic maintenance and FAX repairs) to our customer service FAX line:

CUSTOMER SERVICE FAX: (425) 454-5744

Faxed inquiries are generally resolved sooner than voice inquiries, because they are more efficient. By stating the system behavior in writing, these inquiries are concise and are quick to process.

22.3.3 Support by Phone

For technical support inquiries involving programming to the EMBEDDED BIOS APIs or configuring EMBEDDED BIOS to a specific platform, or for information on General Software's Technical Seminars or BIOS Customization and Support services, contact General Software by phone.

GENERAL SOFTWARE
TECH SUPPORT PHONE: (425) 454-5755

If you have questions about how to use third-party products with EMBEDDED BIOS, consult the third-party vendor first (see the phone numbers earlier in this chapter), and then call General Software after you have talked with them. The third-party vendor will be able to offer more current information on their product than General Software.

22.3.4 Reproducing the Problem

Supporting operating system products is a difficult task that requires more data than application programs. Because we support APIs and not application program features, it is important to provide General Software with a distilled version of a problem at the API level, together with good behavioral data.

For example, if running your program causes the system to "hang", you will need to provide information about what you mean by the system being hung, and what the program does in concise detail. A hung system can mean that no expected output is displayed, or that it cannot be rebooted, etc. You will need to reduce the problem to a statement such as:

> *"My Borland C++ program calls fwrite on a handle I opened with fopen, and the fopen call never returns. Instead, I see the hard disk light remain on, and when I break into the debugger, I see the following display (go ahead and copy the display, including register contents and current instruction being executed)."*

When calling General Software, be prepared to provide more details about your problem, including possibly using debugger commands to aid the technician in learning more about the circumstances of the problem.

As another example, if you were expecting a message to be printed with a printf call, but garbage resulted, then it is important to know what the call looked like, what libraries you were using, whether stack probes are enabled, what compilation model you are using, etc.

22.3.5 Using Tech Support Requests (TSRs)

A Technical Support Request is provided in your Adaptation Kit as an 8 1/2 by 11 inch form that can be photocopied for multiple requests. Please fill-out the form in concise detail and FAX the form to General Software for FAX requests.

If you are an international customer, please set your fax to "fine" resolution before sending, as it is difficult to read many of the faxes we receive from overseas.

22.4 Advanced Troubleshooting

This section covers problems by application category. For example, all common RS-232-related problems have been grouped together so that you can make sure that you have addressed a set of potential hazards in your application.

22.4.1 Diagnosing POST

There are hundreds of reasons that could potentially account for a piece of hardware to not run through POST. If the target does not proceed all the way to its first attempt to boot the operating system, or to enter the SETUP system, then you need to diagnose the POST process. DO NOT CALL GENERAL SOFTWARE FIRST. Instead, use the techniques outlined here, or use similar techniques as the situation requires, to determine the reason why POST does not complete to the point where you can receive a message from the BIOS.

When building a new BIOS for new hardware for the first time, there are bound to be mismatches between the hardware and the software, and it is actually likely that, after applying power to your first BIOS, absolutely nothing will seem to happen.

If we were to provide an exhaustive list of the possible reasons, we could include all the combinations of 400+ configuration options that don't match the hardware. This is simply too large a task.

To bring-up new hardware, it is a far better approach to build a very minimal system BIOS that has almost no features enabled. Ignoring for the moment that there will be no DOS prompt, no video, no keyboard, or even a serial port in such a minimal system configuration, there are ways of isolating the cause of a hang on power-on.

In most cases, power-on hangs are the result of inappropriate selections for the reboot, toreal, or A20 gating functions. The actual source of the latter two functions can be removed by disabling **OPTION_SUPPORT_PROTECT_MODE**; this avoids the complicated code path involved in switching to protected mode during memory sizing, and switching back. It also avoids touching memory above 1MB, which is a likely problem if the chipset hasn't been correctly programmed to accommodate the types of SIMMs being used in the system.

By disabling nearly everything, and enabling something simple, such as video boards, you can try plugging a VGA card into your ISA system and try booting it. If the VGA BIOS gets control, chances are good it will write a message to the display.

If using a VGA BIOS extension to issue a message doesn't work, then it's time to take a look at the simpler types of indicators that can be used to debug the early POST process. Keeping in mind that no stack or RAM can be used during this time, it is still possible to send signals to the outside world via a parallel port, turning on a 7-segment LED, or even turning on the floppy drive motor (which incidently also turns on the LED on the floppy). Even the speaker can be a good binary indicator of progress during early POST.

If you have a POST code monitor board that can monitor I/O port 80h, then enable **OPTION_SUPPORT_POSTCODES** and set **CONFIG_PROGRESS_PORT** to 80h and watch the POST process proceed. When it stops, obtain the last POST code from the read-out on the I/O board, and look it up in the `INC\POST.INC` file. Search the files `POST*.ASM` for the associated symbol, and you'll know how far POST got. This is an excellent starting point to determining what has already transpired, and what piece of code is not returning to its caller.

If you have an 8250-compatible serial port (not necessarily one of the standard ports at 3f8h or 2f8h), then enable **OPTION_SUPPORT_POSTCODES_COM** and set **CONFIG_PROGRESS_COM** to the I/O address of the COM port (i.e., 3f8h). ASCII characters will be output to the COM port, which can be monitored with a terminal program on another computer to watch the entire POST process history. This is another excellent way to see what POST functions have been completed, and what function is stalled. These codes are only generated in module `POST.ASM`, so compare the **POSTCODECOM** macro calls in that file with your output to see where POST stalls.

If no POST code port 80h monitor or serial port is available, you should use one of the binary indicators previously described, such as an LED, speaker, or even floppy disk motor. Then, step through `POST.ASM`, in routine POST, and start at the beginning. Place your signal-producing code directly in-line in the POST code at the very top, and begin moving it down until it is no longer reached. The routine that was just skipped is the source of the problem.

By then inspecting the top-level routine called by POST, you'll be able to place the same types of signal-producing code in-line in the called function. This technique should be able to isolate a difficult problem in about an hour.

The speaker can actually work immediately in most systems. Just place the following instruction in-line in the POST routine, using the technique described above:

```
Rcall  Beep              ; beep the speaker.
```

You can also use the "Bop" function in the same manner; it produces a longer tone. Finally, there is a routine called "Click" that just makes a soft click using the speaker.

To use a parallel port for communication, you may wish to locate an oscilloscope or voltage meter, and connect it to one of the output pins on a parallel port in your system. Then, after determining the I/O port location of the parallel port hardware, use OUT instructions to alternately set and clear the pin. This mechanism can be used to determine if a code path has been reached in early POST.

If POST messages are starting to print, then you can add your own PRINTF statements in POST or other modules (after the point where there is a stack and INT 10h services) to show the contents of registers and memory. Consult Chapter 12, "Using the BIOS Debugger", for a detailed explanation of how to use the PRINTF macro, actually considered part of the debugger.

The integrated BIOS debugger is actually enormously helpful if your system is at the point where it can display messages on the screen. Simply press ALT and the left SHIFT key to enter the debugger. If no keyboard is available, then insert an INT 3 instruction in the code where you want the debugger to breakpoint. <u>Do not be afraid to use the debugger. Most OEMs use it to solve at least one technical problem in an adaptation. It is indispensible.</u>

Following the instructions in Chapter 9 on how to use the debugger, you can dump memory, show the vector table elements, disassemble instructions, and even use the E (enter bytes) command to hand-assemble simple INT requests. For example, to code an INT 13h instruction, followed by an INT 3 instruction, you could use the following set of debugger commands:

```
DEBUG: R CS 4000
DEBUG: R IP 0
DEBUG: E CS:IP CD 13 CC
DEBUG: R
<register display verifies next instruction is INT 13h>
```

Once a simple instruction sequence such as this is set-up in the middle of RAM, it becomes easy to load values in the general registers and see what happens when a sample function is executed, such as a read sector function from the boot record of the boot device.

22.4.2 PCI Issues

If you are having trouble with interrupts in a PCI system, consider the possibility that the chipset may be incorrectly programmed to edge-sensitivity instead of level-sensitivity for the PCI bus.

If your PCI devices are not being detected properly, your PciIrqTbl table in the Board Personality Module may be defined incorrectly. Specifically, the bus, device, and function numbers in the BPM's tables may be specified incorrectly. Compare the tabular values with those on your schematics.

If you have a VGA or SCSI option ROM for an embedded PCI device, make sure it isn't just visible in the ISA region of the address space. PCI option ROMs for embedded devices must be specified with the PCI_ROM statement in the project file, and the binary copy of their option ROM must be visible somewhere in the physical address space of the target without being scannable by the ISA ROM scan, or it will be detected as an ISA ROM and will not be passed the correct information about the device it controls as it should be.

Multiple PCI VGA devices are supported. See the documentation in Chapter 7 about **BoardPciControl** for managing the policy for which devices are selected as the primary VGA device.

22.4.3 Booting Issues

There are several reasons why the BIOS may not boot DOS or the application. If your operating system (DOS) and applications are stored on a ROM disk, and you have trouble getting DOS to initialize, look for the following things:

If you are using a ROM disk image of a floppy, or a floppy disk to boot with, then it is important that the ROM disk software (the ROM extension code itself) reports the same geometry through the INT 13h interface as that of the actual floppy disk used to create the image. If the INT 13h function 08h geometry information does not match the image's actual geometry, then DOS has no way of knowing how to properly read file system data through the INT 13h service. The INT 13h function 08h service is handled automatically in the core EMBEDDED BIOS's ROM disk module, as this information is dynamically retrieved from the boot record in the floppy image.

The third-party ROM BIOS extensions must support INT 13h, function 08h. If it does not, then some versions of DOS cannot boot from the device as-is. Either INT 13h function 08h must be provided (it can be added to the BIOS extension or core BIOS), or the internal DOS device drivers would need to be modified to not use this function, and instead read the information from the BPB. This method does not work in cases where floppy disks do not have BPBs (there are such floppies). General Software's Embedded DOS product allows these modifications as it comes with source code.

Third-party utilities to create BPBs on media may not create them properly. For example, a media descriptor byte in the BPB may not match the first byte of each FAT (the first FAT starts at logical sector 1, whereas the boot record containing the BPB starts at sector 0). The geometry given in the BPB must match the media descriptor byte, and the descriptor byte must be valid.

When making boot disks for input to a ROM disk transfer utility (a program that copies the raw sectors of the boot disk into a file suitable for burning into ROM), it is important that the files on the disk be moved to the beginning of the floppy disk. Because DOS optimizes the storage of files by sweeping forward on the disk, you will leave allocation holes on the disk if you delete a file and replace it with another `COPY` command. Thus, it is important to freshly-format a floppy disk, then issue a sequence of `COPY` commands that copy the files you need to the disk without re-copying the same files or using intervening deletes. This forces all of the information to be stored to the front of the disk, so that 256KB of information stored on a 720KB or 1.44MB floppy disk would fit in a 256KB ROM.

22.4.4 RS-232 Communications Issues

Most versions of DOS initialize the serial ports in their AUX device driver (see `AUXDEV.ASM`). You may wish to disable this if you have special initialization requirements; i.e., you have Remote disk going or you are communicating with the debugger over an RS-232 link. The AUX device driver makes BIOS calls (through INT 14h) to program the serial ports.

If you are polling for RS-232 activities, you may wish to avoid calling the *kbhit* C library function directly. Some versions of this function as provided by C compiler vendors block-out

interrupts for extended periods, causing RS-232 interrupts to be missed. Use *_intdos* instead to make an INT 21h call directly to have the same effect without the overhead. If needed, you can further reduce latency by turning `BREAK OFF` with an INT 21h function, reducing calls to the CON device driver on every INT 21h call to check for ^C.

22.4.5 Console I/O Issues

If while running Microsoft Windows you encounter slow updating of the screen or slow response to the keyboard (seconds for a typed character to appear on the screen), then you need to adjust the 8042 delay parameters in `INC\CONFIG.INC`. Reduce these delays to increase rapid response.

If you are having trouble getting HyperTerminal to operate with Console Redirection, there are two common problems. First, the properties of the current session must be configured, saved, and then HyperTerminal restarted and the session reloaded, before those properties actually take effect. Second, the flow control mechanism being used may be wrong. Embedded BIOS can support hardware flow control or no flow control. If one isn't working, switch to the other method.

If you are seeing the **POSTCODECOM** strings disappear shortly after the cache is enabled, or part way through POST, then there are two probable causes. The first cause is that the **POSTCODECOM** feature has a delay feature enabled (**CONFIG_WAIT_POSTCODE_COM** set to a nonzero value). When the cache is enabled, or shadowing is enabled, the delay becomes much shorter and is not significant enough to pace the characters. This can be switched by setting the parameter to 0, to enable hardware flow control. The second cause is that the UART may be reprogrammed in a Board or Chipset module routine called by POST, such as **BoardInit1**, **BoardInit4**, **BoardInit6**, or **BoardInit8**. It is easy for these routines to reprogram a chipset or Super I/O part and accidently reassign the UART's I/O address or disable it, even though the **POSTCODECOM** feature thinks it is still there.

22.4.6 Hard Disk Issues

If you have a hard disk formatted under another BIOS, and then are having trouble using the hard disk on a target running EMBEDDED BIOS, be aware that other BIOSes may not support or be properly configured to support the same industry standard cylinder/head/sector to 32-bit block number transfer function that EMBEDDED BIOS supports. Actually, EMBEDDED BIOS supports three standards: Raw physical, Phoenix CHS, and LBA. Raw physical means that the cylinders, sectors, and heads are sent to the drive AS-IS, without any conversion to the ATA 32-bit block number protocol. Drives up to 528MB are supported with the raw physical transfer function. The Phoenix CHS function is not industry standard (not accepted by other desktop BIOS vendors) but is accepted by EMBEDDED BIOS by setting the hard drive type in the Setup screen. The details of this transfer function are beyond the scope of this section. The LBA function is industry standard, and EMBEDDED BIOS follows the industry guidelines for converting the INT 13h heads, cylinders, and tracks into 32-bit LBA values which are then passed to the drive in ATAPI protocol. Thus, you may need to configure your hard drive in the EMBEDDED BIOS Setup screen to match the type under which it was formatted in the foreign BIOS. Also, you may need to reformat it for use with EMBEDDED BIOS if the BIOS under which it was formatted was not using an industry standard at the time the disk was formatted.

If you are trying to boot an operating system from a second hard drive, be aware that the Master Boot Record (MBR, the very first sector on the hard disk) contains a partition table, which contains an "Active" flag for the one entry in the partition table that is bootable. This active flag is according to definition 80h to mean "active" and 00h to mean "nonactive". By industry

convention, the MBR **must** use this as a BIOS unit number when reading in the Partition Boot Record (PBR, the 1st sector within the partition to be booted). If the disk was FDISKed using a tool that recorded the unit number instead of just 80h for the active flag, and if the drive was attached to the system as a second drive when the FDISK was done, then it is conceivable that this active flag is set to 81h and not 80h. This would cause the MBR to read the PBR off of the wrong disk device! To check this, use the EMBEDDED BIOS debugger to read the MBR into memory, and visually inspect the partition table to verify that the active flag is set to 80h and not 81h.

If you are trying to boot an operating system from a hard drive that was formatted as a secondary drive, but is now your primary drive in the target, it is possible that the Partition Boot Record (PBR) contains references to drive unit 81h instead of 80h, since the FORMAT utility may have picked up the unit number at the time when it was formatted. If this happened, then your hard drive's PBR is trying to load its operating system from the wrong disk device! You could patch your PBR with a debugger, or choose a different FORMAT program that doesn't have this problem. This is not a BIOS issue, but comes up often in embedded system development because it is common to load-up a drive's contents from a workstation and then use it as a primary device on an embedded evaluation platform.

22.4.7 V20, V25, V30, and 80186 Issues

These processors do not have standard I/O devices. In particular, there is no 8250 UART for serial I/O, no 8259 interrupt controller, no 8237A DMA controller, and no 8254 programmable interrupt timer.

If you are using a high-integration CPU, make sure you have a properly-configured CPU Personality Module (CPM) for the target.

These processors may not have the same set of hardware interrupts normally used on a PC/AT; i.e., your keyboard may not be mapped to IRQ 1, and your timer may not be mapped to IRQ 0 (it most likely is not). Therefore, you will need to modify the operating system running on top of the BIOS to remap these interrupts.

Appendix A

PRODUCT CHANGE NOTES

The following is a list of changes to the documentation by release level for the EMBEDDED BIOS Adaptation Kit components.

EMBEDDED BIOS Documentation

1.0 Initial version; basic configuration of EMBEDDED BIOS and licensing (first release and 17 upgrades).

2.0/2.1 Revised to cover new architecture, Borland/MS tool sets, automated build procedures, new configuration, supporting the hardware, bringing up new hardware, using ROM/Flash disks, using the remote disk, using the BIOS debugger, using CPU personality modules, using chipset personality modules, using advanced power management, expanded BIOS services, troubleshooting, this change notes section (first release).

2.2 Minor typographic corrections and clarifications.

3.0 Major reorganization of topic presentation; revised to cover AMD Elan, NEC V41/51, Flash devices, Resident Flash Disk, RAM disk, Embedded DOS-ROM, SETUP screens, manufacturing mode, PCMCIA socket services, internationalization, and more how-to descriptions of the BIOS adaptation process.

3.1 Support for IDE autodetection, LBA and CHS geometry translation, Manufacturing Mode INT 13h redirection, new INT 15h services.

3.2 Support for bulk erase Flash devices, revised CPM DMA services, miscellaneous typographic corrections, renamed Mini-DOS to Embedded DOS-ROM.

4.0 Reorganized chapters, documented new MCL, revised configuration options and parameters, new BPM, revised CPM and CSPM functions, documented new build

procedure, documented Project files, documented new BIOStart utility, revised BIOS service reference chapter to include new functions.

4.1 Separated DOS section into its own book; documented PCI table macros for project file; documented new options; documented expanded INT 13h architecture to support multiple ROM, RAM, Flash, and OEM-defined file systems; documented Windows CE bootstrap option (CE Ready); documented Atmel Flash MTD; documented consolidation of Basic and Advanced Setup screens into one screen and added drive mapping; documented support for four real IDE drives and four real floppy drives; and documented simplified procedures for formatting RAM and RFD disks supported by the BIOS.

4.2 Documented Pentium II, Pentium III, Celeron, MediaGXm, STPC, and K6 support; added additional callouts for Board and Chipset Personality Modules; added footprint scalability documentation; documented new Flash drivers; revised setup screen description.

4.3 Added PCI chapter; added POST user interface chapter; documented 32-bit BIOS Directory Services, 32-bit PCI services, 32-bit BIOS build, GSMERGE; updated Board, Chipset, and CPU Personality Modules to add new entrypoints and update others; documented NAND Flash file system and Toshiba NAND MTD; removed obsolete parameter documentation; documented new parameters.

EMBEDDED BIOS Software

1.0 MASM/MSC 5.1-compatible BIOS, full source, IDE extra, SCSI extra. (first release and 17 field support upgrades).

2.0 New modular architecture, improved build automation, Borland toolset support, support for MS Visual C 8.0, new GSMAKE utility, file system performance analyzer, CPU personality modules, chipset personality modules, desktop PC-compatible API and data structure compatiblity (first release).

2.1 Changes to support Intel 80386-EX, minor bug fixes.

2.2 Support for 80386-EX CPU Personality Module.

3.1 Support for AMD Elan CPU, V41/V51 CPU, raw Flash I/O, Flash file system, RAM disk, Embedded DOS-ROM, SETUP diagnostics, manufacturing mode, PCMCIA socket services; software changes to support extended memory in a wide variety of targets with nonstandard 8042s and port 92h implementations, enhancements to CPU Personality Module and Chipset Personality Module interface.

3.2 Renamed Mini-DOS to Embedded DOS-ROM, added automatic ROM disk detection and improved compatibility in Embedded DOS-ROM, added bulk erase Flash support for AMD and Intel 28F series parts, corrected LBA/CHS IDE translations, supported DMA-based floppy I/O for 386-EX, added PicoPower and 486 SX/GX support, added AMD Elan PCMCIA ATA autodetection, added PCODE interpreter to reduce size of BIOS and Embedded DOS-ROM.

4.0	Added Board Personality Module (BPM); redefined BPM/CPM/CSPM architecture; reworked configuration parameters to support BPM; combined basic and advanced SETUP screens; replaced old Flash drivers with MCL/MTD system; added Project files and new build procedure; added BIOStart expert configuration system; added special INT 15h functions for use by Embedded DOS-ROM's file system for RFD optimizations; strengthened RFD data integrity checks; added CHKRFD utility; removed obsolete remote disk and PCMCIA socket services; added PCI bus services.
4.1	Expanded INT 13h architecture to support multiple ROM, RAM, Flash, and OEM-defined file systems; added Windows CE bootstrap option (CE Ready); added 430HX/TX PCI adaptations; added Atmel Flash MTD; consolidated Basic and Advanced Setup screens into one screen and added drive mapping; supported four real IDE drives and four real floppy drives; and simplified procedures for formatting RAM and RFD disks supported by the BIOS.
4.2	Added Pentium II, Pentium III, Celeron, MediaGXm, STPC, and K6 support; added additional callouts for Board and Chipset Personality Modules; added 16KB-256KB footprint scalability; new Flash drivers; and PCI bridging.
4.3	Added 32-bit BIOS Directory Services, 32-bit PCI services, and capability of supporting other 32-bit BIOS components with a 32-bit build; added graphical POST with spash screens, POST progress icons, and animation sequences; added CDROM disk emulation (and boot); reorganized PCI support to be split into 16-bit and 32-bit components; added BoardPciControl callout for PCI; updated Board, Chipset, and CPU Personality Module APIs to support additional callouts; removed obsolete PCI Board Personality Module callouts; added SDRAM and SPD support; added STPC Industrial support; added GSMERGE build tool; added more configuration options for finer build control; added FAT snooping to RFD for hard disk emulation; added NAND Flash file system and Toshiba NAND Media Technology Driver; and improved host-side Manufacturing Mode utilities, including HOST.EXE and MFGDRV.SYS to run at 115kbaud with minimized timeouts.