PC Engines

tinyBIOS™

Version 1.3c

PC Engines
pdornier@pcengines.com
www.pcengines.com

Table of contents

Introduction

Thank you for considering tinyBIOS. I hope it will provide you with a fast and low cost solution for your embedded PC firmware needs.

License

The tinyBIOS core is published under the Common Public License.

Chipset support for some chipsets is open source. Others are supplied under a separate license. Please review the files and license agreement for details.

Technical support

"Freedom isn't free". Neither is my time. Many customers were able to adapt prior versions of tinyBIOS by themselves. Chances are, so can you. If you need specific support or customizations, please contact me for a quotation.

Design philosophy

tinyBIOS was written from the ground up to meet the requirements of embedded PC systems, based on my experience in the PC industry and with embedded PC applications. tinyBIOS is designed for use with open source operating systems, not Microsoft Windows.

tinyBIOS is streamlined. Conventional BIOSes include features that embedded applications neither need nor want. Most operating systems use only a small fraction of the BIOS functions available. After start-up, FreeBSD and Linux replace all BIOS functions with their own drivers for best performance.

A setup screen is not desirable - it gives the end user opportunity to break the system, takes time to initialize in manufacturing, and makes the system vulnerable to CMOS RAM corruption. tinyBIOS automatically detects the hard disk type and other parameters, and uses hardcoded values for performance settings.

The source code for tinyBIOS is a fraction of the size of other BIOSes (around 10K lines including typical chipset support). This makes it easier to understand and modify. The resulting binary image is also small (normally less than 16KB), which leaves more ROM space for your operating system, flash file system and application.

Hardware / software requirements

tinyBIOS includes 32 bit code, and requires a 386SX or better CPU. Intel's 386EX CPU would require significant work as its peripherals are not fully PC compatible.

tinyBIOS - a guided tour

tinyBIOS consists of the main file **bios.8** and include files for the different function sets. If in doubt, a *grep* type utility (e.g. included with Borland tools) will help you find things.

Directory structure

The BIOS core is stored at the base of the directory tree. Chipset specific files and the main file are stored in subdirectories. For example:

`\tb`	BIOS core
`\tb\chipset`	Chipset specific code, main file
`\tb\tool`	Build utilities

The chipset directory will include a **make.bat** or similar file for building the BIOS, and assembling the ROM image.

bios.8 main file

You may want to make a separate copy of this file for each project, or use *#if def* options. This file includes user options, includes for all other files, and configuration tables. Most options are set in this file.

General options:

STARTOFS determines the base address of the BIOS. This should be 0c000h for a 16KB image, or 8000h for a 32KB image. Additional padding to match the flash size is added by the **make.bat** file. Caution: version 4.05 of A386 will cause problems if *STARTOFS=0*.

BOOTBEEP enables the beep on bootup. Disable for faster startup.

DEBUG enables display of Int13 trace information. Disable for normal use.

NO_NMI disable NMI handler.

NO_RTCFAIL skip hang on RTC failure

QUICKMEM skip memory test (zero fill only) – enable this for faster startup.

Hard disk, boot options:

BOOT_AC If defined, the BIOS will try to boot from the A: drive first, then from C: hard disk. Comment out to attempt hard disk boot first.

HD_WAIT determines the time in seconds that the BIOS will wait for a hard disk to become active. 20 seconds is typical.

HD_ENA determines the time in seconds that the BIOS will wait before first trying to check hard disk status. Some hard disk drives get terminally confused when accessed before they are ready. 3 seconds is typical.

HDD_LBA enables LBA hard disk support.

HD_EDD enables extended disk drive (EDD) support.

HDD_NOSLAVE disable slave drive detection (select for faster startup)

HD_TIME If defined, the BIOS initializes the hard disk standby mode. The hard disk drive will spin down if there is no activity for this period. Comment out to disable hard disk power management.

Keyboard options:

NO_KBC enable operation even if keyboard controller is not present.

LED_UPDATE If defined, the keyboard LEDs (Num lock, caps lock and scroll lock are updated. Comment out for reduced interrupt latency.

KEY_RATE value for keyboard repeat rate.

Serial port console:

CONSOLE I/O port for console serial port, enables serial console.

CONINT Interrupt for console serial port.

COM_INIT Initialization value for serial ports. Overrides value in **com.8** if set.

PCI options:

INTA, INTB PCI interrupt assignment, also see *PCI_TAB*. 1..15 = interrupt number, 0FFh =
INTC, INTD no interrupt assigned

Platform specific options:

Depends on the chipset.

You may add your own sign-on message at *copyrt*. Terminate with a 0 if you do not wish the standard tinyBIOS message to be displayed.

The routine *decide* can verify error flags, and make a user-specific decision whether to boot the operating system, or display error messages and halt. This is also a good place to insert user code that needs to be executed just before boot.

The table *PCI_TAB* defines the mapping of primary PCI bus slots to PCI interrupts. The first entry is for device 0, etc. The first byte indicates which interrupt INTA# is connected to, the second byte INTB#, etc. The last entry should define no interrupts (all *INT0*), and is copied for all remaining device numbers. Interrupts on secondary PCI buses are mapped according to the PCI bridge specification (simple interrupt rotation).

Warning: Be careful about changing the sequence of *include* files. For example, the chipset support file has to come before the **hdd.8** module to make *#if def* statements work right.

message.8 BIOS messages

All standard messages are stored in this file for easier change and translation.

This file is normally included directly after the *copyrt* message in **bios.8**. If the *copyrt* message is not terminated by a 0 byte, the PC Engines copyright message will be displayed.

The PC Engines copyright message does not need to be displayed, but must remain readable in the binary image.

equ.8 equates

This file defines industry standard I/O and lower memory addresses.

tinyBIOS uses segment 0 instead of 0040h. This allows for easier access to interrupt vectors. The only place where this causes problems is the keyboard BIOS. The keyboard buffer pointers are offset by 0400h to account for the segment shift.

Variables starting at 0504h are for temporary POST use only, and are cleared just before boot. tinyBIOS does not use an extended BIOS data area (normally located just below 640 KB). Instead, rarely written data such as hard disk parameters is kept in shadow RAM, in the **data.8** module.

data.8 run-time data area

This module holds BIOS runtime configuration data. To enable writes to it, call *cs_shadrw*. To protect this area, call *cs_shadro*. The BIOS calls *d_dosum* before checksum test to update the checksum for this block.

hd_prm0 and *hd_prm1* contain the hard disk parameters. *d_lastbus* indicates the number of the highest PCI bus in the system (0 when no bridges are present).

d_dosum uses the *wbinvd* instruction to ensure that the cache has been written back to memory (including shadow RAM) before recalculating the checksum.

tables.8 ISA initialization tables

The table *clrtab* defines initial values for several system board devices. *tdmatab* defines operational settings.

rs_baud defines the mapping from INT14 parameter to baud rate. *rs_ports* defines the I/O ports tinyBIOS looks at for serial ports. *lp_ports* is the same for parallel ports.

rtc_tab defines the default settings for the RTC/CMOS RAM. It is used when invalid time / date or a battery failure are detected.

reset.8 Reset vector

The reset vector is at FFF0h: *jmp far 0F000:reset*.

The BIOS date (in mm/dd/yy format), AT model byte, and the BIOS checksum (the sum of all bytes, including the checksum byte, is zero) are filled in by the **biossum.exe** utility.

chipset.8 chipset support (actual name varies)

System-specific code for chipset and CPU support is kept in this file to minimize user changes to the BIOS core. This file has a different name for each chipset, for example **m1487.8** for the ALI M1487 chipset.

Keep in mind that many of these routines are called early in POST, before DRAM is initialized.

This file has the following entry points:

cs_init initializes chipset registers after reset. If shadow RAM is enabled, it returns with carry set to prevent modification of the DRAM settings.

cs_det detects what capacity and type of DRAMs is installed in the SIMM sockets. This test is highly chipset dependent. Ideally, the chipset will allow you to enable each bank separately. The detection algorithm can then enable one bank, set it to the maximum possible size (e.g. 16Mx32), and determine how many column addresses are really working: write a known pattern to address 0, then write another pattern to address 512. If the pattern at address 0 wasn't destroyed, the column address bit worked, keep shifting the address left until this test fails or all bits have been tested.

cs_shad copies the system BIOS to shadow RAM. POST will skip this if shadow is already enabled.

cs_cpu configures CPU specific settings such as write back cache, and enables the level 1 cache.

cs_vshad copies the video BIOS from the BIOS EPROM (at C000) to shadow RAM. This routine should skip if video BIOS shadow is already enabled.

cs_vshad2 copies the video BIOS from a PCI expansion ROM (address [esi]) to shadow RAM. This routine is entered in unreal mode to allow access to extended memory.

cs_spd determines the CPU bus speed (indirect, through measurement of ISA bus speed), and sets the DRAM and IDE timing registers accordingly.

cs_shadrw enables read/write access to the shadow RAM. This is required for access to the *data* module, which stores e.g. hard disk parameters in shadow memory.

cs_shadro disables write access to the shadow RAM.

cs_a20on turns on the A20 gate. This is normally done through the fast A20 gate facility of the chipset, rather than through the keyboard controller. Note that tinyBIOS never turns off A20 gate.

cs_waitbx uses the refresh bit of port 61 to provide consistent delay timing. This assumes a refresh rate of 16µs.

cs_npx tests and enables the floating point coprocessor.

cs_pciint programs the mapping from PCI to ISA interrupts.

cs_ide (optional) is called by the HDD initialization code to set the IDE timing. DL is the drive number, DS:DI points to the identification block returned by the IDE device.

superio.8 super I/O support (actual name varies)

This file has a different name depending on the super I/O device, e.g. *smc669* for the SMC37C669 super I/O controller.

This module has two entry points:

sio_init 0 early initialization

sio_init normal initialization

Serial port initialization for serial port console should also be done here.

post.8 Power-on Self Test

The reset vector at F000:FFF0 points to the beginning of POST, *reset*. At this point, DRAM has not been initialized. Procedure calls are simulated by the *ret_sp* macro: point SP at the return address, and the stack segment at the code segment, then jump to the procedure. At each step POST calls *postcode* to write a status code to port 80h, which can be read out by a POST card, or optionally copied to a serial port.

POST first initializes several I/O registers (*post_clr*), including refresh, and then calls the chipset initialization (*cs_init*). *cs_init* initializes the chipset. If the system is already running out of shadow memory (shadow reboot, or Ctrl-Alt-Del), the following steps are skipped: *cs_det* detects and configures DRAM. *cs_shad* copies the system BIOS to shadow RAM.

vid_init initializes a monochrome video card. To prevent conflicts with the memory test, *pci_rst* must disable PCI bus masters. The BIOS then enters "unreal mode" (*getunreal*, see **post2.8**), and tests the first 64KB of memory (*post_t64k*).

At this point the BIOS can finally set up a normal stack. *cs_cpu* initializes CPU registers and enables the level 1 cache. *d_dosum* updates the checksum for the data area in shadow memory. Then *post_sum* verifies the BIOS checksum. If an error is detected (carry flag set), *post_err* will hang.

sio_init initializes the super I/O controller. *rtc_test* tests the RTC/CMOS. *post_ref* verifies that the refresh bit at port 61h is toggling, and indirectly tests the 8254 timer. *cs_spd* detects the CPU bus speed and sets speed-dependent chipset registers. *post_dma* tests the 8237 DMA controller registers. *post_page* tests the DMA page registers. *post_tim* tests the 8254 timer registers. *kb_ini* does the first initialization of the 8042 keyboard controller. *post_tdma* configures timer, DMA, interrupt and port 92h (A20 gate) registers. *post_irq* tests 8259 interrupt mask registers.

post_base performs the base memory test (64KB to 640KB). To minimize the number of special cases, full memory test is performed even in the case of a Ctrl-Alt-Del restart (rare for embedded systems). Then *post_vec* initializes interrupt vectors, and *vid_vars* sets up video BIOS variables. *pci_pnp* enumerates and enables all PCI buses and devices. *cs_pciint* maps the PCI interrupt channels to ISA interrupts. If not already done by PCI plug & play, *cs_vshad* copies the video BIOS at C000 from ROM to shadow RAM. *post_scan* then calls the video BIOS if present.

At this point, POST displays the signon message and base memory size. *kb_inb* continues the keyboard test and initialization. *post_ext* tests extended memory. *kb_inc* does more keyboard test, and *tim_init* initializes the 8254 timer tick. *rtc_ini* initializes the RTC. *fd_init* detects and initializes the floppy disk controller. *post_scan* looks for options ROMs above C800. *lp_test* detects and initializes printer ports. *rs_test* detects and initializes serial ports.

cs_npx detects and enables the numeric coprocessor. *fd_inb* continues floppy initialization. *hd_init* detects and initializes hard disk drives. *tim_test* makes sure that the timer tick and RTC are working.

At this point it is up to user specific code in *decide* to inspect error flags, and decide whether to boot the operating system or not. Finally, POST calls Int19 to boot the operating system.

post2.8 Power On Self Test subroutines

This file includes routines used during POST. Most POST routines have been mentioned in the *bios* module, and will not be described again.

Most of the device initialization is table-driven, *clrtab* for the first settings, and *tdmatab* for operational settings. These tables are stored in the file **tables.8**.

tinyBIOS uses “unreal mode” to allow access to the full 4GB address space from real mode. Unreal mode is based on a quirk of the x86 segment architecture: all segment registers include an invisible descriptor register for the base and limit. These cache registers are *not* cleared when returning from protected to real mode. The routine *getunreal* temporarily enters protected mode, loads the DS and ES segment registers with selectors that allow full access, then immediately returns to real mode. As long as DS and ES are not loaded again, the full address space can be accessed by using 32 bit addressing instructions.

The low memory sizing algorithm in *post_*szlo first writes the address to words in 64KB intervals (going down), then verifies the address (going up).

The high memory sizing algorithm in *post_szhi* first uses a binary search (address shift) to determine the approximate memory size, then uses the same algorithm as *post_szlo*.

All memory tests are performed in unreal mode, with cache enabled. *post_t64k* tests a 64KB block of memory. At the start of the block, a 64 bit sliding bit pattern (1 - 2 - 4 etc) is written and verified. Then, a 72 byte seed pattern is written. This ensures that data never enters the cache. The pattern is moved over the memory block such that the pattern is written to and read from DRAM repeatedly. Then the pattern is verified against the original. Finally, the memory block is cleared.

post_vec initializes the interrupt vectors based on the table *inttab*.

The routine *postcode* can be customized to display POST codes to a serial or parallel port. Please be aware that memory and the stack are *not* available, this routine is called from early POST. Also, you will have to initialize the super I/O controller first.

debug.8 Debug functions

Debug code should be placed in this file. This file is disabled by default to make sure that no debug code finds its way into your production BIOS. Support for these routines may change at any time.

To find additional debug hooks in the BIOS source, search for ;& .

diaghex displays a hex byte on the first line of the display, using direct video memory access. *diagword* displays a hex word using direct video memory access.

hexbyt displays a hex byte, going through Int10. *hex* displays a hex word, going through Int10.

scancode gets a key and displays its scan code.

v_dump and *v_dump2* display the registers in a Int13 stack frame, and allow easy tracing of disk activity.

diag_csip can be called from a timer interrupt to display the program counter at the time of the timer interrupt. This uses direct video memory access.

vid.8 Int10 video BIOS

The video BIOS included in tinyBIOS only supports monochrome (Hercules) video cards, and is intended mainly for debug purposes. VGA support is provided by the video BIOS, and overrides this code. tinyBIOS does not support the CGA card. The light pen, print screen (very printer dependent) and CGA font are not supported.

vid_init initializes the text mode, and is called both by POST and Int10. *vid_vars* initializes video variables.

vid_str displays a null-terminated (ASCIZ) string through direct video memory access. *v_msg* displays a null-terminated string through the Int10 TTY function. *beep* makes noise.

int10 is the entry point for the video BIOS. tinyBIOS supports the following functions:

AH=00 **Set video mode**. AL normally holds the mode number, and is ignored by tinyBIOS.

AH=01 **Set cursor type**. CH cursor start line, CL cursor end line.

AH=02 **Set cursor position** (0,0 = origin). DH row, DL column, BH page number.

AH=03 **Read cursor position**. BH page number → CL cursor end line, CH cursor start line, DL column, DH row.

AH=04 Read light pen position → AH 0 (not supported)

AH=05 **Select screen page.** AL page number

AH=06 **Scroll up active page**

AH=07 **Scroll down active page**. AL number of lines (0 to blank window), CH upper left row, CL upper left column, DH lower right row, DL lower right column. BH attribute for blank line.

AH=08 **Read from current position**. BH page number → AL character, AH attribute.

AH=09 **Write to current position**. AL character, BL attribute, BH page number, CX number of characters.

AH=0A **Write to current position** (attribute preserved). AL character, BH page number, CX number of characters.

AH=0B Set color palette - not supported.

AH=0C Write dot - not supported.

AH=0D Read dot - not supported.

AH=0E **Teletype write to active page**. AL character. Bell (7), backspace (8), line feed (10) and carriage return (13) are supported.

AH=0F **Current video state** → AL active mode, AH number of columns on screen, BH page number.

AH=13 Write string - not supported.

Int1D **Pointer to video parameters** *v_parm*.

Int1F **Pointer to extended font**. Initialized to 0:0.

int1x.8 Int11, Int12, Int15, Int18, Int19 miscellaneous interrupts

This file includes dummy interrupt handlers (iret) for the following interrupts:

int00	Divide by zero
int01	Single step
int03	Breakpoint
int04	Overflow
int06	Invalid opcode
int07	Coprocessor not available
int1b	User keyboard break handler
int1c	User timer tick handler

nmi handles non-maskable interrupts. Depending on the error, *nmi* will display a message and halt the system.

inteoi and *inteoi2* are default interrupt handlers that send the *eoi* command (20h) to the primary and (if needed) secondary interrupt controller.

int05 implements a dummy print screen function, and sets the error flag *m_prtsc*.

irq13 implements a numeric processor error handler. It clears the error by writing port F0h, then issues *eoi* to the interrupt controllers and transfers to the NMI handler for legacy support.

Int11: equipment flag

int11 returns the state of the equipment flag *m_devflg* in AX. Bits are defined as follows:

15:14	number of printer ports
13	0
12	0 = no game port present
11: 9	number of serial ports
8	0 = DMA present
7: 6	00 = 1 floppy disk drive (if bit 0 set)
5: 4	initial video mode, 11 for monochome, 10 for VGA
3	0
2	0 = no PS/2 mouse installed
1	1 if numeric coprocessor present
0	1 if floppy disk drive present

Int12: base memory size

int12 returns the base memory size *m_lomem* in 1KB units in AX.

Int15: miscellaneous functions

int15 only supports a very limited subset of the functions originally defined by IBM.

AH=80	Device open - not supported
AH=81	Device close - not supported
AH=82	Program termination - not supported
AH=83	Event wait - not supported
AH=84	Joystick - not supported
AH=85	**System request key pressed**. AL 00 make 01 break. Called by the keyboard handler.

AH=86 **Wait**. CX,DX wait time in µs (effective granularity is ~1ms) → carry clear.

AH=87 **Move block**. CX word count, ES:SI pointer to GDT → AH 00 ok. tinyBIOS uses unreal mode to implement this function, and assumes that the A20 gate is enabled through fast gate A20. Interrupts are disabled during block transfer.

AH=88 **Extended memory size** → AX extended memory size in 1KB units. Extended memory size is stored in CMOS locations 30h and 31h.

AH=89 Switch to protected mode - not supported.

AH=90 Device busy. Not called by tinyBIOS.

AH=91 Interrupt complete. Not called by tinyBIOS.

AH=C0 **Return system configuration** → ES:BX pointer to *conf_tab*, AH=0, C=0.

AH=C1 Return Extended BIOS Data Area Address - not supported.

AH=C2 **PS/2 mouse BIOS** – see file **ps2mous.8**.

AH=E8 **Big memory size** → AX,CX memory between 1MB and 16MB in 1KB blocks, BX,DX memory above 16MB in 64KB blocks.

Int18: Expansion ROM entry

This was originally intended for the BASIC expansion ROM. *int18* displays the message "no boot device" and waits for a keystroke, then returns.

Int19: Boot operating system

int19 attempts to boot the operating system. Depending on the user option *BOOT_AC*, the floppy drive or the hard drive will be tried first. To minimize the delay, the BIOS will give up on the floppy drive after one try if there was a time-out, which usually means that there was no diskette.

The routine *bootdrv* reads one sector from cylinder 0, head 0, sector 1 to the address 0:7C00h. If the read is successful, and the signature AA55h is found at the end of the boot sector, control is transferred to 0:7c00h.

fdd.8 Int13 floppy BIOS

To simplify the code, tinyBIOS only supports one 1.4MB floppy drive, and does not support 720KB media. This limitation should not cause problems in an embedded environment.

int13 handles the floppy BIOS functions. All commands return error codes in AH, and set the carry flag if there was an error. The status is also stored in *m_fdstat*. Error codes are:

00	ok
01	bad command
02	bad address mark
03	write protected
04	record not found
06	media change
08	DMA overrun
09	DMA across 64KB boundary
0C	Media type not found
10	Bad CRC
20	Floppy controller failed
40	Seek error
80	Time-out

AH=00 **Reset diskette system**. DL drive number (if 80h or above, will also reset hard disk drives). This function initializes the floppy controller, the floppy drive will be recalibrated before the next access.

AH=01 **Return diskette status**. DL drive number → AH status of last operation (from *m_fdstat*).

AH=02 **Read sectors**. AL number sectors, ES:BX buffer pointer, CL sector 1..18, CH track 0..79, DL drive number, DH head 0..1 (values are not checked) → AL number of sectors transferred, AH status code.

AH=03 **Write sectors**. Same parameters as read.

AH=04 **Verify sectors**. Same parameters as read, ES:BX not needed.

AH=05 **Format track**. Same parameters as read. ES:BX points to a buffer of 4 bytes per sector: Track number, head number, sector number, sector size (0 = 128, 1 = 256, 2 = 512, 3 = 1024 bytes).

AH=08 **Read drive parameters**. DL drive number → AX 0, BL 4 = 1.44MB, BH 0, DL 1 number of drives, DH 1 maximum head number, CL 18 sectors per track, CH 79 maximum track number, ES:DI points to *fd_ptab*. If no drive is present, 0 is returned in all registers.

AH=15 **Read drive type**. DL drive number → AH 00 if no drive present, 02 disk change line available if present.

AH=16 **Diskette change line status**. DL drive number → AH 00 not active 01 invalid parameter 06 disk change active 80 not ready.

AH=17 **Set disk type for format**. DL drive number, AL disk type → same return values as AH=16. Since tinyBIOS only supports 1.44MB media, the disk type is ignored.

AH=18 **Set media type for format**. DL drive number, CH 79 number of tracks, CL 18 sectors per track → ES:DI pointer to *fd_ptab*, AH status. This function returns 0C (invalid format) unless the number of tracks and sectors match.

Int1E: pointer to disk parameters

This vector points to the diskette parameter table *fd_ptab*.

0	bits 7..4 step rate (F= 1 ms, E= 2 ms), 3..0 head unload time (0=8 ms, 1=16ms)
1	bits 7..1 head load time (1= 1 ms, 2= 2 ms), bit 0 DMA mode
2	motor wait time in timer ticks
3	bytes / sector (0=128, 1=256, 2=512, 3=1024)
4	end of track
5	gap length
6	DTL (data length low)
7	gap length for format
8	fill byte for format
9	head settle time (ms)
10	motor start time (1/8 seconds)

hdd.8 Int13 hard disk BIOS

int13hd handles the interrupt 13 functions for hard disk access. Hard disks are identified by DL=80h (IDE master) or DL=81h (IDE slave). Floppy disk accesses are forwarded to the floppy BIOS through interrupt vector 40h.

Important: Hard disk access will not work if the BIOS isn't running from shadow RAM !

AH=00 **Reset disk system**. DL drive number - if 80h or above, will reset the hard disk, otherwise floppy drives only.

AH=01 **Return disk status**. DL drive number → AH status of last operation (from *m_hdstat*).

AH=02 **Read sectors**. AL number sectors, ES:BX buffer pointer, CL bits 5..0 sector, CL bits 7..6 cylinder high, CH cylinder low, DL drive number, DH head → AH status code.

AH=03 **Write sectors**. Same parameters as read.

AH=04 **Verify sectors**. Same parameters as read, ES:BX not needed.

AH=05 Format track - not supported by most IDE drives, hardware dependent.

AH=08 **Read drive parameters**. DL drive number → CL maximum sector, cylinder high, CH maximum cylinder low, DL number of consecutive drives, DH maximum head number.

AH=09 **Set drive parameters**. DL drive number → AH status. This function initializes the disk geometry according to the drive table *hd_prm0* or *hd_prm1*.

AH=0A Read long - not supported.

AH=0B Write long - not supported.

AH=0C **Seek**. CL bits 7..6 cylinder high, CH cylinder low, DL drive number, DH head number → AH status.

AH=0D **Alternate disk reset**. DL drive number → AH status. This function resets the hard disk only, not the floppy drives.

AH=10 **Test drive ready**. DL drive number → AH status.

AH=11 **Recalibrate**. DL drive number → AH status.

AH=14 **Controller diagnostics**. This issues the diagnostic command to the drive.

AH=15 **Read drive type**. DL drive number → AH 00 if no drive present, 03 (fixed disk) if present. CX,DX number of sectors.

AH=23 **Set drive powerdown (non-standard function)**. DL drive number → AH status. This uses the value *HD_TIME* as a drive time-out.

AH=24 **Set multiple mode**. AL number of sectors, DL drive number → AH status. Note that the BIOS itself currently doesn't use multiple mode.

AH=25 **Identify drive**. ES:BX destination buffer (512 bytes), DL drive number → AH status. This function uses the 0ECh identify drive function.

The following functions are only supported if option *HD_EDD* is enabled. See the Phoenix Enhanced Disk Drive Specification version 1.1 for details. Please note that the current EDD code expects *all* drives in the system to support LBA addressing.

AH=41 **Check extensions present**. BX=55AA, DL drive number → AH major version, BX=AA55, CX=1 support packet structure, no lock/eject.

AH=42 **Extended read**. DL drive number, DS:SI address packet → AH status

Address packet buffer is a 16 byte structure:

0	packet size in bytes (should be 16)
1	reserved, must be 0
2	number of blocks, max. 127
3	reserved, must be 0
4	transfer buffer offset
6	transfer buffer segment
8	block number (64 bit LBA)

AH=43 **Extended write**. AL verify on/off (ignored), DL drive number, DS:SI address packet → AH status

AH=44 **Extended verify**. DL drive number, DS:SI address packet → AH status

AH=45 Lock/unlock drive. Not implemented.

AH=46 Eject removable media. Not implemented.

AH=47 **Extended seek**. DL drive number, DS:SI address packet → AH status

AH=48 **Get drive parameters**. DL drive number, DS:SI address result buffer → AH status.

The result buffer has the following structure:

0	buffer length, must be at least 26 at entry
2	information flags: 2 = valid geometry, not removable, no write with verify, no change line support, not lockable
4	number of cylinders
8	number of heads
12	number of sectors
16	number of physical sectors (64 bit)
24	number of bytes per sector = 512

Int41, 42: **Pointers to drive 0 and drive 1 parameters** *hd_prm0* and *hd_prm1*. Hard disk parameters are stored in shadow RAM (part of the data module). They have the following format:

0	Logical cylinders
2	Logical heads
3	Signature, 0A0h
4	Physical sectors
5	(precompensation cylinder) Multiple count
7	(reserved) Shift count for CHS translation
8	Drive control byte
9	Physical cylinders
11	Physical heads
12	(landing zone)
14	Logical sectors
15	(reserved)

tinyBIOS does not have a setup screen, and requires drives to be detected automatically. This means that ancient drives that don't support the identify drive command will not work.

The BIOS interface is limited to 1023 cylinders, but can handle up to 255 heads. The IDE task file supports up to 65535 cylinders, but only 16 heads. CHS translation (*hd_chs)* is used for large capacity disk drives (1024 cylinders or more). The translation works by shifting the cylinder number right, and moving these bits to the head number.

CHS translation fails for drives with more than about 6GB capacity. If option *HDD_LBA* is enabled, tinyBIOS will use LBA mode when CHS translation is not possible.

Caution: There is no standard for when to use CHS or LBA, there may be problems if tinyBIOS tries to run a disk formatted in LBA mode in CHS mode.

com.8 Int14 serial port BIOS

int14 is the entry point for the serial port BIOS.

AH=00 **Initialize serial port**. DX port 0..3, AL configuration → AL modem status, AH line status.

AL bits 7..5	000: 110 baud	001: 150 baud
	010: 300 baud	011: 600 baud
	100: 1200 baud	101: 2400 baud
	110: 4800 baud	111: 9600 baud
AL bits 4..3	00: no parity	01: odd parity
	10: no parity	11: even parity
AL bit 2	0: 1 stop bit	1: 2 stop bits
AL bits 1..0	10: 7 data bits	11: 8 data bits

AH bit 7	Time out
AH bit 6	Transmit shift register empty
AH bit 5	Transmit holding register empty
AH bit 4	Break detect
AH bit 3	Framing error
AH bit 2	Parity error
AH bit 1	Overrun error
AH bit 0	Data ready

AL bit 7	Received line signal detect
AL bit 6	Ring indicator
AL bit 5	Data set ready
AL bit 4	Clear to send
AL bit 3	Delta receive line signal detect
AL bit 2	Trailing edge ring detect
AL bit 1	Delta data set ready
AL bit 0	Delta clear to send

AH=01 **Send character**. AL character to send, DX port 0..3 → AH line status.

AH=02 **Receive character**. DX port 0..3 → AL character received, AH line status.

AH=03 **Return communication port status**. DX port 0..3 → AL modem status, AH line status.

If the option *CONSOLE* is set, support for a serial console is included. All Int 10 TTY output is copied to the serial port. Input from the serial port is converted slightly (ANSI → scan codes for cursor keys), and inserted in the keyboard buffer. When *CONSOLE* is set, the Initialize function is inhibited to keep DOS from setting the serial port to 2400 baud on startup.

A VGA BIOS will take over Int 10, and disable output to the serial console.

ps2mous.8 Int15 PS/2 mouse BIOS

This file is provided on request only, needs more work. Define *PS2MOUSE* in the main file to include this optional module. *int15c2* is the entry point for the PS/2 mouse BIOS.

AX=C200 **Enable** (BH=1) **/ disable** (BH=0) **mouse** → AH status, C=1 if error

AH=00:	No error
AH=01:	Invalid function call
AH=02:	Invalid input
AH=03:	Interface error
AH=04:	Resend
AH=05:	No call-back installed

AX=C201 **Reset** mouse → AH status, BH device ID (usually 0), BL byte returned by mouse (usually AA), C=1 if error.

AX=C202 **Set sample rate** BH rate → AH status, C=1 if error.

BH=00:	10 samples per second
BH=01:	20 samples per second
BH=02:	40 samples per second
BH=03:	60 samples per second
BH=04:	80 samples per second
BH=05:	100 samples per second
BH=06:	200 samples per second

AX=C203 **Set resolution** BH resolution → AH status, C=1 if error.

BH=00:	1 count per mm
BH=01:	2 counts per mm
BH=02:	4 counts per mm
BH=03:	8 counts per mm

AX=C204 **Read device type** → AH, BH device ID (usually 0), C=1 if error.

AX=C205 **Mouse interface initialization** BH packet size → AH status, C=1 if error.

tinyBIOS only supports BH=3.

AX=C206 **Return status** BH=00 → AH status, C=1 if error, BL status byte 1, CL status byte 2, DL sample rate.

BL bit 7:	Reserved
BL bit 6:	0 = stream mode, 1 = remote mode
BL bit 5:	0 = disable, 1 = enable
BL bit 4:	0 = 1:1 scaling, 1 = 2:1 scaling
BL bit 3:	Reserved
BL bit 2:	Left button pressed
BL bit 1:	Reserved
BL bit 0:	Right button pressed.

CL = 00:	1 count per mm	CL = 01:	2 counts per mm
CL = 02:	4 counts per mm	CL = 03:	8 counts per mm

AX=C206 **Set 1:1 scaling** BH=01 → AH status, C=1 if error.

AX=C206 **Set 2:1 scaling** BH=02 → AH status, C=1 if error.

AX=C207 **Install call-back routine** ES segment BX offset → AH status, C=1 if error.

If ES and BX are both 0, the call-back is disabled. When a position update is available, the following data is pushed on the stack, and the user call-back routine is called by far call. The user routine should leave the sample data on the stack.

Word 1: Bit 7 = 1: Y overflow
Bit 6 = 1: X overflow
Bit 5 = 1: Y negative
Bit 4 = 1: X negative
Bit 3 = 1
Bit 2 = 0
Bit 1 = 1: right button pressed
Bit 0 = 1: left button pressed

Word 2: X data (8 bits)

Word 3: Y data (8 bits)

Word 4: 0

kbd.8 Int16 keyboard BIOS

int16 is the entry point for the keyboard BIOS.

AH=00 **Keystroke read** → AL ASCII code, AH scan code.

AH=01 **Keystroke status** → Z=1 no scan code available; Z=0 scan code available, AL ASCII code, AH scan code. The keystroke remains in the buffer.

AH=02 **Shift status** → AL shift status

AL bit 7: Insert
AL bit 6: Caps Lock
AL bit 5: Num Lock
AL bit 4: Scroll Lock
AL bit 3: Alt pressed
AL bit 2: Ctrl pressed
AL bit 1: Left shift
AL bit 0: Right shift

AH=03 **Set repeat rate**. AL=05, BL rate 0 (fastest) to 1F (slowest), BH initial delay (0 = 250ms, 3 = 1s).

AH=05 **Place keystroke in buffer**. CL ASCII code, CH scan code → AL 00 if ok, 01 if buffer full.

AH=10 **Extended keystroke read** → AL ASCII code, AH scan code. Extended functions allow differentiation between numeric keypad and cursor block keys.

AH=11 **Extended keystroke status** → Z=1 no scan code available; Z=0 scan code available, AL ASCII code, AH scan code. The keystroke remains in the buffer.

AH=12 **Extended shift status** → AL shift status (same as AH=02), AH extended shift status

AL bit 7: SysRq pressed
AL bit 6: Caps Lock pressed
AL bit 5: Num Lock pressed
AL bit 4: Scroll Lock pressed
AL bit 3: Right Alt pressed
AL bit 2: Right Ctrl pressed
AL bit 1: Left Alt pressed
AL bit 0: Left Ctrl pressed

Int15 AH=4F **Scan code intercept**. Called by the keyboard handler. AL scan code → clear carry if keystroke has been intercepted.

Int1B **Keyboard break handler**

The keyboard handler calls this interrupt when Ctrl-Brk is pressed.

kbtab.8 Keyboard layout

To simplify localization, the keyboard layout is driven by the table *kb_tab*. There are 11 bytes for each key table entry: A control byte to decide whether the key is sensitive to Caps Lock or Num Lock, and scan codes for normal, shift, control, alt and control-alt states. Special keys such as shift are encoded through action codes (see *vectab* in **kbd.8**).

lpt.8 Int17 printer port BIOS

int17 is the entry point for the printer BIOS.

AH=00 **Print character**. AL character, DX port 0..2 → AH status

AH bit 7:	Not busy
AH bit 6:	Acknowledge
AH bit 5:	Out of paper
AH bit 4:	Printer selected
AH bit 3:	I/O error
AH bit 2:	reserved
AH bit 1:	reserved
AH bit 0:	Time-out

AH=01 **Initialize printer port**. DX port 0..2 → AH status (see above).

AH=02 **Get status**. DX port 0..2 → AH status (see above).

rtc.8 Int1A timer / RTC BIOS

int1a is the entry to the timer / RTC BIOS.

AH=00 **Read tick counter** → CX,DX tick count, AL 0 if no day overflow, 1 if we passed midnight. This function clears the overflow flag. The timer tick runs at the rate of 1193180/65536, about 18.2 ticks/s.

AH=01 **Set tick counter**. CX,DX new tick count. This function clears the overflow flag.

AH=02 **Read RTC time** → CL minutes, CH hours, DL 1 if daylight savings option enabled, 0 otherwise. DH seconds. C=0 if clock running, C=1 if not. All RTC data is in BCD format.

AH=03 **Set RTC time**. CL minutes, CH hours, DL 1 to enable daylight savings option, 0 otherwise, DH seconds.

AH=04 **Read RTC date** → CL year, CH century, DL day, DH month. C=0 if clock running, C=1 if not.

AH=05 **Set RTC date**. CL year, CH century, DL day, DH month.

AH=06 **Set RTC alarm**. CL minutes, CH hours, DH seconds → C=0 if ok, C=1 if error or alarm already set.

AH=07 **Disable RTC alarm**.

Int1C **Timer tick user hook**

The timer tick routine calls this interrupt on each timer interrupt.

Int4A **Alarm interrupt**

The BIOS will call this interrupt when the alarm goes off.

tinyBIOS uses the following CMOS RAM locations:

00..09	time / date / alarm
0E	power loss flag
0F	diagnostic byte
15	base memory size, low (option *CM_LEGACY*)
16	base memory size, high (option *CM_LEGACY*)
17	extended memory size low (option *CM_LEGACY*)
18	extended memory size high (option *CM_LEGACY*)
30	extended memory size (low byte)
31	extended memory size (high byte)
32	century

Whenever the RTC date is read, tinyBIOS looks for a year 2000 rollover. Specifically, when the century and year read 1900, the century is forced to 2000.

pci.8 Int1A PCI BIOS

int1a jumps to *int1a1* if AH=B1. *int1a1* handles all 16 bit PCI BIOS functions. Since it is explicitly mentioned in the PCI BIOS spec, tinyBIOS also supports the direct entry point at F000:FE6E. The BIOS32 service directory is supported as well.

AX=B101 **PCI BIOS present** → AL hardware mechanism 11h (mechanism 1), AH 00, BX 0210h (version 2.10), CL number of last PCI bus in system, EDX "PCI ", C=0.

AX=B102 **Find PCI device**. CX device ID (0..65535), DX vendor ID (0..65534), SI index (0..N) → AH status 00 ok 83 bad vendor ID 86 not found, BL device number / function number, BH bus number. C=1 if error.

AX=B103 **Find PCI class code**. ECX class code (lower three bytes), SI index (0..N) → AH status 00 ok 86 device not found, BL device number / function number, BH bus number, C=1 if error.

AX=B106 **Generate special cycle**. BH bus number, EDX special cycle data → AH status 00 ok 81 function not supported, C=1 if error.

AX=B108 **Read configuration byte**. BL device number / function number, BH bus number, DI register number → AH status 00 ok, CL read data, C=1 if error.

AX=B109 **Read configuration word**. BL device number / function number, BH bus number, DI register number (multiple of 2) → AH status 00 ok 87 bad register number, CX read data, C=1 if error.

AX=B10A **Read configuration dword**. BL device number / function number, BH bus number, DI register number (multiple of 4) → AH status 00 ok 87 bad register number, ECX read data, C=1 if error.

AX=B10B **Write configuration byte**. BL device number / function number, BH bus number, CL data, DI register number → AH status 00 ok, C=1 if error.

AX=B10C **Write configuration word**. BL device number / function number, BH bus number, CX data, DI register number (multiple of 2) → AH status 00 ok 87 bad register number, C=1 if error.

AX=B10D **Write configuration dword**. BL device number / function number, BH bus number, ECX data, DI register number (multiple of 4) → AH status 00 ok 87 bad register number, C=1 if error.

AX=B10E Get PCI IRQ routing options. Not supported.

AX=B10F Set PCI hardware interrupt. Not supported.

pcipnp.8 PCI plug & play

tinyBIOS includes a PCI plug & play algorithm. Note: Bridge support currently doesn't work.

Very early during startup, POST calls *pci_rst* to disable all bus masters / bridges. This is necessary to prevent bus masters from causing conflicts with the memory test. Later, POST calls *pci_pnp* to enumerate the bus and allocate resources.

pci_pnp searches all PCI devices, and allocates memory / prefetchable memory / low memory (below 1MB) and I/O space as requested by devices. Interrupt lines are assigned according to the table *PCI_TAB*, or according to the standard rotation scheme for devices connected to PCI bridges.

Except for the video BIOS, tinyBIOS currently doesn't support PCI option ROMs. For most embedded systems, a lower cost approach is to integrate boot ROMs for SCSI and network devices with the system BIOS.

The traditional approach to resource allocation would be to pack resources as tightly as possible. For simplicity, and to better support future developments such as PCI hot plug, tinyBIOS uses user-defined minimum allocation sizes to improve the chances of successful reconfiguration.

PCI resource allocation is controlled by the following options. These are usually defined in the chipset module.

PCI	If defined, will include the PCI BIOS and PCI plug & play modules.
P_MEM0	Start address (64K multiple) for PCI memory space.
P_MEM9	End address (64K multiple) for PCI memory space.
P_ROM0	Temporary address for PCI expansion ROMs. This is used during plug & play only, disabled during operation.
P_MEMP0	Start address (64K multiple) for prefetchable PCI memory space. This space must be separate from *P_MEM0*.
P_MEMP9	End address (64K multiple) for prefetchable PCI memory space.
P_MEMINC	Minimum memory allocation (64K multiple) for memory devices. Must be 2^n.
P_MEMINC1	Minimum memory allocation (64K multiple) for primary bus bridges. Must be 2^n.
P_MEMINC2	Minimum memory allocation (64K multiple) for secondary bus bridges. Must be less than *P_MEMINC1*, and 2^n.
P_MEMR0	Start address (16 byte multiple) for low 1MB PCI memory.
P_MEMR9	End address (16 byte multiple) for low 1MB PCI memory.
P_MEMRINC	Minimum memory allocation (16 byte multiple) for low 1MB PCI memory. Must be 2^n.
P_IO0	Start address for PCI I/O space.
P_IO9	End address for PCI I/O space. Normally 64K.
P_IOINC	Minimum I/O allocation for I/O devices. Must be 2^n.
P_IOINC1	Minimum I/O allocation for primary bridge. Must be 2^n.
P_IOINC2	Minimum I/O allocation for secondary bridge. Must be 2^n.
P_PRILAT	Default value for primary latency timer

P_SECLAT Default value for bridge secondary latency timer

P_BRIDGE PCI bridge control register

P_COMMAND PCI device command register

Compatibility issues

tinyBIOS is optimized for embedded systems, not for maximum compatibility with legacy operating systems such as Microsoft Windows.

A20 gate

The A20 gate is a workaround dating back to the IBM PC/AT. When on, the address line A20 is passed through, giving access to the full address space of the CPU. When off, the address line A20 is masked to zero, causing accesses to e.g. FFF0:0100 to wrap around to 0000:0000. This was "needed" for compatibility with some existing applications.

tinyBIOS always leaves the A20 gate on.

This option tells **himem.sys** to use port 92, which eliminates the test:

```
DEVICE=HIMEM.SYS /m:2
```

BIOS functions

tinyBIOS does not implement all BIOS functions. Functions that were found to be of little value to embedded applications were left out. Refer to the documentation of the individual source files for details.

Fixed entry points

tinyBIOS only supports two fixed entry points:

`F065` Int 10 (required by some VGA BIOSes)

`FE6E` Int 1A (required by PCI specification)

Other legacy entry points are not supported - applications are not supposed to rely on specific BIOS locations.

Building tinyBIOS

tinyBIOS can be built on virtually any system that runs DOS (or provides a DOS shell) and supports or emulates direct video memory access (required by the **as.com** editor). It is often possible to develop directly on the target system. New BIOS code can be tested by loading it into shadow RAM. Once the new code looks good, the flash EPROM can be reprogrammed in system, or using a device programmer.

See the **make.bat** files in the chipset directories for examples of the complete process. You may need to modify this to include a video BIOS or other modules.

A386 assembler

Use the **A386** assembler to assemble tinyBIOS. It is published by Eric Isaacson (**www.eji.com**). Porting tinyBIOS to other assemblers such as MASM, NASM or TASM is strongly discouraged.

- **A386** is fast (seconds to assemble the entire BIOS).
- **A386** does not require the arcane overhead needed with most other x86 assemblers.
- **A386** directly generates a binary file, no linker required.
- No complex **make** file needed.

Some issues to keep in mind with A386:

- Numbers with leading zeroes are interpreted as hexadecimal unless ended with xb.
- Code overruns are currently not handled gracefully. If in doubt, configure a lower STARTOFS (e.g. 8000h).

biossum.exe checksum utility

biossum.exe does the following:

- Delete all bytes prior to *STARTOFS*.
- Insert checksum byte for 32 bit PCI BIOS header (*bios_32hd* in **pci.8**)
- Insert checksum byte for BIOS read / write data area (*d_beg* in **data.8**)
- Insert BIOS date (**reset.8**)
- Insert model byte (**reset.8**)
- Insert overall checksum byte (**reset.8**)

Use Borland Pascal 7.0 (earlier versions might also work) to recompile this program.

Generating a ROM image

The resulting BIOS image will usually be smaller than the flash EPROM. For a typical example, concatenate a 32KB video BIOS, 64KB of padding, and 32KB system BIOS:

```
cat video.bin ..\fill32.b  in ..\fill32.bin bios.abs >rom.rom
```

add.com will append binary files to the first file listed:

```
add destination source1 source2
```

cat.com will concatenate binary files to standard output:

```
cat source1 source2 > destination
```

Loading the ROM image

Use a device programmer or EPROM emulator to program an EPROM with the binary image. To save time, write a batch file or script for this. If working directly on the target system, you can reprogram the flash EPROM in circuit (somewhat risky, if the BIOS doesn't come up you will have to reprogram the EPROM in a device programmer), or load the BIOS into shadow RAM.

Shadow reboot is the fastest way to test a new BIOS versions, but has some limitations: Since the BIOS is only present in DRAM, DRAM autosizing cannot be tested from a shadow BIOS.

as.com source code editor

Use the **as.com** editor to edit the source files. The tinyBIOS source is formatted with tab stops at columns 11, 19 and 41. Please use this formatting convention for any code you contribute.

as.com includes a simple text editor. To open a file, enter:

```
as bios
```

To save and exit a file, press Escape. Source code for as.com is available on request.

The editor is similar to the WordStar and Borland Pascal editors. Tab stops are optimized for assembly language source. Most normal editing can be done with the usual function keys, you really don't need to learn many functions.

Some useful commands:

^Backspace	Delete to end of line
^J	Search a word, starting from beginning of file (use ^QF for more options)
^L	Repeat search
^QA	Search and replace
^U	Move display such that cursor line is in the middle. Cursor row will lock if Scroll Lock is pressed.
^Y	Delete line.

File management: This editor can handle a main file (selected by command line, or by ^KM), and an include file at the same time. Both files together must be less than 64KB.

Escape	Save and exit (or return to main file if editing include file)
^KI	Return to previous include file.
^KL	Load include file
^KM	Load main file, or return to main file
^KQ	Abandon changes, return to main file or exit to DOS.
^KS	Save current file

Block commands. This editor always selects entire lines.

^KA	Select all
^KB	Mark beginning of selection
^KC	Copy selection
^KK	Mark end of selection
^KR	Read file
^KV	Move selection
^KW	Write selection to file

^KY Delete selection

Keyboard Macros: Function and Alt key combinations can be programmed with macros.

^OL F1 Program macro for F1 - type in key sequence, then press ^Brk to complete.
Press F1 to execute macro.

For additional details, please refer to **ed.txt. ed.com** includes the same editor as **as.com**, but with conventional tab stops.

alist.exe source code print utility

alist.exe will print assembly source files on any HP Laserjet compatible printer. Output is in a two up landscape format, including tab stop handling.

Use Borland Pascal 7.0 (earlier versions might also work) to recompile this program.

Adapting tinyBIOS

Embedded systems are much less homogeneous than desktop PC's. This means that BIOS modifications are almost always needed. tinyBIOS is designed to simplify the adaptation as far as possible.

See the section *Building tinyBIOS* for the mechanics on assembling and loading tinyBIOS.

Once things are working reasonably well, I prefer to work directly on the target system, using a CompactFlash card (less fragile than an actual hard disk !) for the source code.

Debug tools and techniques

The most important troubleshooting tool in BIOS development is a POST card. During POST, tinyBIOS writes status codes (POST codes) to port 80h. When something goes wrong, the last POST code will remain on the display, and allow quick identification of the offending code. Insert additional POST codes to narrow down the crash location if necessary. See the file **postcode.txt** for a listing.

A logic analyzer is helpful for low level debug. Where available, I prefer to monitor the PCI bus, as it doesn't require as many samples as the ISA bus (32 bit vs. 8 bit code fetches), and includes PCI configuration cycles. ISA and PCI test adapters are available from PC Engines.

x86 code is difficult to disassemble in logic analyzers. To follow BIOS execution, convert the BIOS binary file to a .COM file (start offset 100h, end offset FFFEh), and load this file into Borland Turbo Debug or a similar debugger on the development system. Use the disassembler screen to follow execution. Enter ^G to enter a start address, or ^F to follow a jump or call instruction. Or change the A386 options to generate a listing (-l).

An in-circuit emulator (ICE) is not required.

A monochrome graphics card (Hercules / MGP) can be very helpful for low level debugging, and only requires a 8 bit ISA interface to your target system. It can be initialized by a few lines of code, and can be used even when DRAM isn't working yet. VGA BIOSes will not work without system DRAM.

When debugging video cards, it can be helpful to redirect console output to a serial port. Set the serial port parameters with the MODE command, then type CTTY >COM1 to redirect output to the serial port. Input still comes from the keyboard.

When inserting debug or tentative code, always include a comment with an easy to find signature, e.g. &&&. That way, you can easily find and remove debug code once everything is working properly.

Other methods to monitor BIOS status include sending results to a serial port, or displaying them through the regular Int 10 interface. For example, see the debug hooks in *int13* that can display all register values on entry and exit.

Chipset adaptation

Most PC chipsets have a large number of configuration registers that must be initialized by the BIOS. Incorrect settings either won't function at all, work slowly, or cause unstable operation. As a starting point, use the register settings recommended by the chipset supplier, or dump register settings from a working board. To get results more quickly, initially use hardcoded settings for memory size and cache configuration. Add autodetection later.

Memory size detection tends to be tricky, and is very hardware-dependent. A typical method is to set a DRAM bank to the maximum size supported by the chipset. Write a test pattern to address 0, then write another pattern to the address corresponding to the highest column bit (e.g. 1000h). When the pattern at address 0 is modified, the tested address bit didn't work, and we can deduce the DRAM type. Once a bank has been determined, disable it, and detect the next bank.

Cache detection is similar, but is complicated by the presence of the CPU cache, and potential cache write-back.

When doing performance tuning, verify the actual timing with an oscilloscope or logic analyzer. Chipset documentation is not always very clear, and timing violations can cause unstable operation.

Which files to modify

The tinyBIOS source code has been structured such that you should only have to modify a few of the files. Specifically:

bios.8 Main file. Create a copy with a new name for your project. Set options and select include files.

chipset.8 (actual name varies) Chipset specific code. This often needs to be modifed for your application. Use *#if def* statements for any changes you make.

superio.8 (actual name varies) Super I/O specific code. This often needs to be modifed for your application. Create a copy with a new name for your project.

Resources

Tools

POST cards, CompactFlash to IDE adapters, PCI/ISA logic analyzer adapters	PC Engines	www.pcengines.com/test.htm
ROM emulators:	EE Tools	www.eetools.com
	Tech-Tools	www.tech-tools.com
Device programmers:	Needhams Electronics	www.needhams.com

Literature

Please also visit www.pcengines.com/embres.htm for pointers to literature and standards. The following are documents I used, you can find similar information in many other documents.

BIOS interrupts, memory locations

IBM PS/2 and PC BIOS Interface Technical Reference

IBM no. S68X-2341-00.

IBM Surepath BIOS PC Standard

Download at www.surepath.ibm.com/documents/pubsdocs.html (no longer available)

Interrupt listing, Ralf Brown

Look for interxxx.zip at www.simtel.net/simtel.net/msdos/info-pre.html .

PC Interrupts : A Programmer's Reference to BIOS, DOS, and Third-Party Calls

Ralf Brown, Jim Kyle, Addison-Wesley, ISBN 0-201-62485-0.

System BIOS for IBM PCs, Compatibles, and EISA Computers

Phoenix Technologies, Addison-Wesley, ISBN 0-201-57760-7.

DOS + 386 = 4 Gigabytes ! Article on unreal mode.

Dr. Dobbs Journal 7/1990.

8237 DMA controller

Intel 8237 data sheet, Intel 82430FX PCIset ISA bridge data sheet (#290519-001)
c't 9/88 pg. 178ff (German magazine)

8254 Timer

Intel 8254 data sheet, Intel 82430FX PCIset ISA bridge data sheet (#290519-001)
c't 4/88 pg. 196ff

8259 Interrupt Controller

Intel 8259 data sheet, Intel 82430FX PCIset ISA bridge data sheet (#290519-001)
c't 8/88 pg. 174ff

RTC / CMOS

IBM PS/2 technical reference
Dallas DS12887 data sheet
c't 12/88 pg. 196ff

Port 61, port 92

IBM PS/2 technical reference
c't 7/88 pg. 164ff

Numeric coprocessor initialization

Intel 486 Programmer's Reference Manual

Monochrome video

IBM PC technical reference library
Documentation of a video card (manufacturer unknown)

Serial port, parallel port

National PC87332VLJ data sheet
IBM PC technical reference library
c't 6/88, pg. 166ff
c't 5/88, pg. 204ff
PC Magazine 3/14/89, pg. 315ff

8042 keyboard interface

IBM PC technical reference library
Compaq SystemPro technical reference manual, Chapter 8

Floppy controller

IBM PC technical reference library
National PC87332VLJ data sheet

IDE hard disk

ATA spec (find links at www.bswd.com/cornucop.htm)
Phoenix (www.phoenix.com/techs/specs.html)

PCI BIOS and plug & play

PCI specification, PCI BIOS specification (http://www.pcisig.com)
Microsoft (www.microsoft.com/hwdev)

Chipset

Data books for ALI M1487 and other chipsets.

Super I/O

Data sheets for SMC FDC37C669 and other super I/O devices.

PS/2 mouse

USAR UR7HCMTBMousecoder data sheet.