

HP 95LX Developer's Guide

**Copyright Hewlett-Packard Company 1991
All rights reserved**

**1st Revision
August 30, 1991**

Notice

This manual and the software described herein are provided "as is" and are subject to change without notice. **Hewlett-Packard Company makes no warranty of any kind with regard to this manual or the software described herein, including, but not limited to, the implied merchantability and fitness for a particular purpose.** Hewlett-Packard Co. shall not be liable for any errors or for incidental or consequential damages in connection with the furnishing, performance, or use of this manual or the software described herein.

Copyright Hewlett-Packard Co. 1991. All rights reserved. Reproduction, adaptation, or translation of this manual, including any programs, is prohibited without prior written permission of Hewlett-Packard Company, except as allowed under copyright laws.

Corvallis Division
1000 NE Circle Blvd.
Corvallis, OR 97330, U.S.A.

Printing History

Preliminary Draft	May 1, 1991
1st Revision	August 30, 1991

Important Note to Software Developers

Hewlett-Packard is committed to making your current and future development efforts as easy as possible. In keeping with this commitment, Hewlett-Packard recommends that developers avoid using HP 95LX hardware specific features for I/O. Software written using HP 95LX hardware specific features is very likely not to run on future HP product.

HP 95LX Technical Information

Table of Contents:

	Pages
1. ISV Developer's Overview	1-1 to 1-4
2. Off-the-Shelf Development Tools	2-1
3. HP 95LX DOS	3-1
4. HP 95LX BIOS ERS	1 to 104
5. File Specifications for HP 95LX Built-in Applications	4-1 to 4-9
6. HP 95LX Memory Management	6-1 to 6-2
7. HP 95LX Low-Level Graphics Support	7-1 to 7-21
8. HP 95LX System Manager Operation and Programmer's Guide	7-1 to 7-18
9. HP 95LX System Manager Services Reference	8-1 to 8-60
10. From Software Design to Ordering ROM Cards	10-1 to 10-7
11. 'Hopper' HP 95LX System Controller ERS	1 to 80
12. HP 95LX Wired Serial and Infrared I/O ERS	1 to 15
13. Custom Artwork	14-1
14. PC Card Standard (PCMCIA 1.0)	1 to 102

ISV Developer's Overview

The information contained in this notebook is intended for Independent Software Vendor (ISV) use in planning of software development or adaptation tasks. It is hoped that this information will provide you with enough details about the Jaguar platform in order to formulate a strategy for adapting your product to the HP 95LX.

Levels of Adaptation

Most programs intended to run on The HP 95LX will require or at least benefit from being modified to conform to The HP 95LX hardware. Assuming that this has been considered, the next consideration is how the software will run on The HP 95LX. The The HP 95LX platform provides three options in this area:

1. Run from RAM as an independent DOS program. Such a program would be compiled and linked with standard DOS tools and would be loaded and executed by DOS exactly as on a standard PC.

There are several distribution options for these programs. All that is required for execution is that they reside on a The HP 95LX disk. Hence they could be down loaded to The HP 95LX's internal disk from a PC or a modem connection, or they could be distributed on a plug-in ROM card which has been formatted as a disk.

This is the simplest execution option. However it may be the least RAM efficient and it will not be integrated with The HP 95LX's built-in applications.

2. Run as an independent ROM executable XIP (eXecute In Place) program. Such a program would be distributed on a ROM card and at execution time would bank switch its code into the CPU address space using The HP 95LX's bank switching capabilities.

The advantage of this over option 1 is greater RAM efficiency. There is still no integration with the built-in applications.

This level of adaptation will require the code to be ROMable, will require the use of special software tools to prepare the ROM image and will require the program to use The HP 95LX's bank switching services. The necessary tools and services are not completely described in this document, but will be defined in the developer's kit. See the "Memory Management" chapter for more information on XIP.

3. Run under the System Manager. Such a program will be called System-Manager-compliant and can be either RAM executable or XIP.

Under this option, the program has access to the same set of services that are used by the built-in applications. For example, this enables the program to share a non-preemptive multitasking environment with the built-in applications so that the user can conveniently switch between tasks.

The “System Manager Guide” and “System Manager Reference” sections deal with writing System-Manager-compliant applications.

The HP 95LX Hardware Overview

The HP 95LX is a Palmtop PC which is very PC compatible except in areas which have been customized to obtain its small size or to support large amounts of memory. These areas are highlighted here.

Display

The HP 95LX’s physical display is 40 characters by 16 lines in text mode. This display is a window into an MDA standard 80 character by 25 line display RAM. While there is provision for windowing the physical display around the larger display RAM, it is expected that most software development for The HP 95LX will include customization to a 40 by 16 screen.

The HP 95LX’s display also has a (non-standard) graphics mode which has pixel dimensions of 240 columns by 128 rows.

Character Font

The HP 95LX character fonts are stored in ROM. Codepage 850 was used since it contains international characters. This was chosen to facilitate the localization of the product. Developers should be aware that the standard for PC’s in codepage 437 which contains line drawing characters.

Keyboard

The HP 95LX’s keyboard is shown in figure 1. See the BIOS Int 9h documentation for the scan/ASCII codes corresponding to each key.

Memory Structure

The HP 95LX has more “logical” memory (i.e., memory residing on memory chips) than can be accommodated in the 1 megabyte “physical” address space of the CPU. Bank switching is used to access the additional memory. For example, Figure 2 shows that bank switching is used to access the code for the built-in applications and to access the memory on a plug-in card. See the “Memory Management” chapter for more information.

Plug-in Cards

The HP 95LX has one plug-in card slot which accommodates a PCMCIA/JEIDA standard memory card. This card slot is somewhat analogous to a floppy disk drive on a standard PC. There are two main types of cards.

One is a battery-backed RAM card which is formatted as a RAM disk. This type of card is analogous to a floppy disk. RAM cards will be available in 128K and 512K byte sizes.

The other is a ROM card which contains application software. This card could be formatted as a disk in which case it would be analogous to a read-only disk. A ROM card can also

contain XIP software designed to be bank switched into CPU address space rather than accessed as a disk file. ROM cards will be available 1 and 2 megabyte sizes.

The HP 95LX Software Overview

Built-in Applications

The HP 95LX features eight built-in applications: Lotus 1-2-3, HP calculator, Memo, Phone Book, Appointment Book, Terminal Emulator, Filer, and Setup. These applications are accessed by pressing their corresponding key, see figure 1.

System Manager

The System Manager is the control program which runs the built-in and other System-Manager-compliant applications.

All the built-in applications execute from ROM under the direction of the System Manager. Briefly, when an application's key is pressed, the System Manager deactivates any current application, performs any necessary bank switching to access the code for the requested application and starts up the new application. This is termed non-preemptive multitasking since that application being deactivated gets a chance to "clean up" before losing control.

See "System Manager Guide" and "System Manager Reference" for more information.

DOS

The HP 95LX uses Microsoft DOS 3.22. This version of DOS was chosen because it executes from ROM leaving the maximum amount of RAM available for applications.

The DOS kernel functions are always available and provide access to the DOS command processor. However, since the emphasis of The HP 95LX is on applications, The HP 95LX does not contain the full set of DOS external commands. See the "DOS" chapter for notes on how DOS has been customized for The HP 95LX.

BIOS

The HP 95LX contains a ROM BIOS layer which provides the standard interface as well as many extensions to support The HP 95LX specific hardware. See the "BIOS" chapter for details.

Disks

The HP 95LX's disk support is patterned after that of a PC which has an internal hard disk drive and one floppy disk drive.

The HP 95LX's internal disk, which is named C:, is a combination RAM/ROM disk. The ROM disk portion contains a variety of files such as help files for the built-in applications and utility programs such as FORMAT. The RAM disk portion contains user files and resides in system RAM. The size of the RAM disk is user settable using the "Setup" application.

The HP 95LX uses plug-in battery-backed RAM cards formatted as RAM disks for its “floppy” disks. The plug-in port is the A: (or B:) drive, analogous to a single floppy drive on a PC.

Off-the-Shelf Development Tools

Since the HP 95LX runs Microsoft DOS version 3.22, compilers, assemblers, and debuggers compatible with this version of DOS can be used to develop software for the HP 95LX. Some development packages such as Turbo-C from Borland support remote debugging over a serial interface. This configuration works on the HP 95LX and is very useful for certain types of software development.

Since the HP 95LX is a relatively small PC, users of off-the-shelf tools should make certain they understand what overhead is built into the *use* of their tools. Two examples follow:

- Run time support libraries sometimes include modules that are not needed by the main program.
- Users building HP 95LX system-manager-compliant programs need to be aware of any library initialization code their package automatically includes in programs they build. Much of this sort of initialization code is incompatible with the HP 95LX System Manager. Steps can usually be taken to use library routines that do not require initialization.

HP 95LX DOS

Introduction

The HP 95LX contains the primary components of ROM-executable MS-DOS 3.22.

More specifically, the HP 95LX contains the DOS Kernel and the command processor, but only a few external commands.

Generally speaking, DOS works the same on the HP 95LX as it would on a standard PC (except for the missing external commands). For this reason, these notes discuss only the *differences* between the operation of MS-DOS 3.22 on the HP 95LX and DOS's operation on a standard PC.

DOS Boot Sequence

On a standard PC, DOS boots off disk and runs from RAM. On the HP 95LX, DOS boots out of ROM and runs from ROM. Thus on the HP 95LX DOS, RAM requirements are greatly reduced since only DOS data is stored in RAM.

There is no provision on the HP 95LX to boot from disk. However, a disk can contain CONFIG.SYS and AUTOEXEC.BAT files—see below.

DOS Initialization

On a standard PC, DOS makes the boot disk drive the default drive and searches the root directory of that drive for a CONFIG.SYS file.

On the HP 95LX, DOS initialization has been changed to first search for CONFIG.SYS on the A: drive and, if not found, to search on the C: drive. If CONFIG.SYS is found on the A: drive, then that CONFIG.SYS is processed and A: is made the default drive. In this case, any CONFIG.SYS that is on the C: drive will not be processed. If CONFIG.SYS is not found on the A: drive, then C: is made the default drive and any CONFIG.SYS found there will be processed.

The DOS EXEC Function

In ROM-executable MS-DOS 3.22, the EXEC function has been extended to first search a table of ROM-executable programs before searching disk in case the program name has no drive path specified.

On the HP95LX, the table contains the two names COMMAND and \$SYSMGR. The COMMAND program is the DOS command processor and the \$SYSMGR program is the System Manager.

The Default Shell

On standard DOS systems, the default shell is COMMAND.COM. On the HP 95LX, COMMAND.COM is the System Manager.

The System Manager is the program that directs execution of the built-in and special add-on applications. The HP 95LX System Manager does not process any AUTOEXEC.BAT files.

The DOS Command Processor

COMMAND.COM is available in the HP 95LX, but is not normally run at initialization time since DOS goes directly to the System Manager shell. There are two ways to access the command processor:

1. Invoke the command processor from within the System Manager program.
2. Change the shell to the command processor by including the line

SHELL=COMMAND /P

in the CONFIG.SYS file and reboot.

This causes DOS to go directly to COMMAND.COM — the System Manager is not invoked and the built-in application keys will not be active. The commands that are available are the internal command processor commands, any external programs on disk, and the two ROMed program commands listed above.

One use of this method is to get access to DOS in order to perform some custom initialization and then run the System Manager using the \$SYSMGR command.

Disk-Resident External Commands

The HP 95LX contains two external commands, COMMAND.COM and CHKDSK.EXE, which reside in the root directory of the ROM disk.

The COMMAND.COM program is only a stub that invokes the actual command processor residing in ROM. This stub is provided in case some program accesses the command processor by its full name (including extension). As mentioned above, COMMAND (without path or extension) will be executed directly from ROM.

CHKDSK is the standard DOS check disk command.

The HP 95LX also contains DEBUG.EXE in a hidden directory called _SYS. This directory is off the root of the ROM disk. This is the standard DOD DEBUG program.

International Support

MS-DOS 3.22 supports the COUNTRY= configuration parameter and DOS calls that allow tailoring DOS and applications to local language characteristics. Some of this information depends on the the PC's character set. DOS 3.22 supports only code page 437 (switchable code pages start with DOS 3.3). Since the HP 95LX uses code page 850, the country-specific information in HP 95LX DOS has been updated to match code page 850.

Note however that the international support that is built into the System Manager does not use these DOS capabilities.

JAGUAR BIOS

External Reference Specification

Version 2.02

May 1, 1991

Hewlett-Packard Company

Overview

This is the specification of the Jaguar ROM Basic Input/Output System (BIOS). The BIOS provides the lowest level of software support for applications running on Jaguar. This specification describes in detail the implementation of the Jaguar BIOS. The Jaguar BIOS is designed to be completely compatible with IBM's new version of the PC-XT. In addition to the PC-XT BIOS functions, the Jaguar BIOS includes a small number of BIOS functions which are compatible with the IBM-AT

The Jaguar BIOS is based upon an XT level BIOS source code obtained from Phoenix Software Associates (PSA). The code HP purchased from PSA was written to be compatible with the IBM PC-XT of pre-April 1986 vintage, i.e. before the introduction of a new version of the XT with the enhanced keyboard. To our knowledge, the PSA XT code received by HP was written by Phoenix without infringing on any of IBM's copyrights to the XT BIOS code. In the same spirit, HP has added and modified the code without copyright infringement.

Jaguar Hardware set

Jaguar is an Information Management Calculator. It features an 8088 processor and hardware set that is moderately compatible with an IBM-XT. Differences between Jaguar and XT hardware are listed below:

- Jaguar has a smaller display than a XT. The jaguar display size is 40 x 16 (text) or 240 x 128 dots (graphics) vs XT's monochrome display size of 80 x 25 (text) or 320 x 200 dots (CGA graphics). The Display RAM in Jaguar is the same size as that of an XT with a Monochrome Display Adapter(4K bytes). Also, there is provision to window around in the display RAM, so the user can see the contents of all 4K of RAM.
- Jaguar's display cursor size control is different from that of an IBM-XT.
- Jaguar has no mechanical disk. Instead there is a built-in RAM disk.
- Jaguar has a different keyboard layout from the IBM-XT.
- Jaguar's keyboard management is different from the IBM-XT. Keyboard Scans are implemented in software in Jaguar, while they are performed by an 8048 microcontroller in a XT. However, the Jaguar keyboard interrupt service routine will emulate the 8048.
- Jaguar supports plug-in ROMs.
- Jaguar supports plug-in RAMs. All memory in plug-in RAM will be used as RAM disk.
- Jaguar is switched ON or OFF under software control. This is compared to an XT which is switched ON or OFF by a hardware switch that controls power to the entire machine.
- Jaguar has LCD contrast adjustment under software control. This is fundamentally different from XT brightness control, which is done with a potentiometer adjustment.
- Jaguar does not support a parallel printer. However, it does support a serial printer which uses XON-XOFF flow control.
- Jaguar supports only one serial port UART. However the serial channel can be directed either to the IR or wired serial port.
- Jaguar RAM is 8 bits wide. There is no parity bit, as in the IBM-XT.
- Jaguar's hardware interrupt set is not identical in function to an XT.

Changes to PSA Code

The following changes were made to the PSA code. This list is just touches on the major changes. To obtain more information on the changes made, refer to the chapter on the BIOS Interrupts or go straight to the BIOS source code.

<Reset Vector>	Power On Code was changed so that turning the machine on causes the machine to return to the application that was running before the system was powered down.
Int 02	Nonmaskable interrupt (NMI). This is invoked in the IBM-XT when a RAM parity error occurs. It is invoked on Jaguar by either a Low Battery or Module Pulled event.
Int 05	Print Screen Interrupt. This prints the contents of the active display window only, not the contents of the entire display memory as in the IBM-XT.
Int 06	Low Power Hook This interrupt is called by the system: <ul style="list-style-type: none">– Just before going to light sleep.– Just after awakening from light sleep.– Just before going to deep sleep.– Just after awakening from deep sleep.
Int 08	Timer hardware service. The timer service was modified to add display window control and battery level checks.
Int 09	Keyboard interrupt. Int 09 was modified to support Char key translations, Mute key translations and the ALT-NUMPAD code was modified to work with top row number keys instead of number pad keys.
Int 0A	Miscellaneous interrupt. This is a reserved interrupt in the IBM-XT.
Int 0B	Keyboard and touch panel hardware interrupt. In Jaguar, hardware keyboard interrupt in Jaguar is INT Bh, not INT 09. INT 0Bh code debounces pressed keys and places key code in the keycode register (I/O address 60h). Then it invokes the INT 09 service routine. NOTE: INT 0Bh is the COM2 interrupt in the IBM-XT.
Int 0D	HOPPER IR Interrupt This is the fixed disk interrupt in the IBM-XT.
Int 0E	External XINT pin hardware interrupt routine. This is diskette interrupt in the IBM-XT.
Int 0F	HOPPER RTC interrupt. This is the LPT1 interrupt in the IBM-XT.
Int 10	Video Services Changed CGA functions to maintain a moderate degree of compatibility. Left MDA functions intact.

Overview

Int 13	Disk services. Modified to work with a RAM disk.
Int 14	Serial Port Services. Removed waits for DSR and CTS set when sending a character. Removed wait for DSR set when receiving a character. The receive character service changes the serial port interrupt vector to point to a dummy interrupt service routine (just an IRET).
Int 15	System services (Cassette control in now defunct XT) Just about all of Int 15 is new. Keyboard translation hook (Int 15 function 4F)
Int 16	Keyboard Services. Modified to trap [ON] key press while machine is running. Also invokes light sleep code.
Int 17	Printer services Modified to work with a serial printer. It implements the XON-XOFF handshake in Jaguar. These services change the serial port vector to point to a serial service routine that handles XON-XOFF handshakes.
Int 19	Boot service Now boots DOS from ROM.
Int 1A	Time of day services Added support of real time clock, including the capability of setting an alarm. The alarm is capable of turning on power to the unit.
Int 1E	Set to a dummy IRET. This is the disk parameter table in the IBM-XT.
Int 1F	Graphics character table pointer This points to a code page 850 font for characters 80h - FFh.

The following PSA interrupt handlers were not modified:

Int 11	Equipment check service
Int 12	Get memory size service
Int 14	Serial port service
Int 1B	Keyboard break default handler
Int 1C	Timer tick default handler (how can you change an IRET?)
Int 1D	Video parameter table

BIOS RAM Definition

BIOS RAM Definition

This chapter describes how memory is organized and used by the BIOS.

Interrupt Vectors	00000h
BIOS Data Area	00400h
DOS Data Area	00600h
Memory Mapped Display RAM	001000h
DOS Data Area	02000h
Disk Operating System (DOS)	(Variable)*
Application Program Area	(Variable)*
RAM Disk Portion of Drive C:	(Variable)*
Unused	80000h
OS Functions	A0000h
MDA RAM	B0000h
Unused	B1000h
Two 64KB Page Frames	C0000h
Four 16KB Page Frame	E0000h
BIOS ROM	F0000h
	FFFFFh

* Size of the Disk Operating System area varies since optional drivers and buffers may occupy variable amounts of RAM.

Interrupt Vector Table. The interrupt vector table is in the address range from 0:0000h through 0:3ffh. Vectors used by BIOS are initialized by the BIOS initialization code.

The table below lists the interrupt vector assignments and identifies each interrupt by function and type. The interrupts consist of four types: services, ISRs, hooks, and tables:

- A service is an application program callable interrupt. Such interrupts provide functions that an application can call by using the appropriate Int instruction.

BIOS RAM Definition

- An ISR is a hardware interrupt service routine. These routines should not be called from applications since unpredictable results may occur.
- A hook is an interrupt service routine provided for applications to optionally take over.
- A table is a pointer to a table of data bytes.

Int	Address Range (Hex)	Function	Type
00h	000-003	Divide by Zero	Hook
01h	004-007	Single Step	Hook
02h	008-00B	NMI Interrupt	Hook
03h	00C-00F	Breakpoint	Hook
04h	010-013	Arithmetic Overflow	Hook
05h	014-017	Print Screen	Service
06h	018-01B	Low Power Hook	Hook
07h	01C-01F	Reserved	Hook
08h	020-023	IRQ0, Timer0 Hardware Interrupt	ISR
09h	024-027	IRQ1, PC Compatible Keyboard Interrupt	ISR
0Ah	028-02B	IRQ2, HOPPER Miscellaneous Interrupt	ISR
0Bh	02C-02F	IRQ3, HOPPER keyboard & touch panel interrupt	Hook
0Ch	030-033	IRQ4, HOPPER UART interrupt	Hook
0Dh	034-037	IRQ5, HOPPER IR input interrupt	ISR
0Eh	038-03B	IRQ6, HOPPER XINT pin interrupt	ISR
0Fh	03C-03F	IRQ7, HOPPER RTC interrupt	ISR
10h	040-043	Video Services	Service
11h	044-047	Equipment Check	Service
12h	048-04B	Memory Size	Service
13h	04C-04F	Flexible Disk Services	Service
14h	050-053	Serial Port Services	Service
15h	054-057	System Functions	Service
16h	058-05B	Keyboard Services	Service
17h	05C-05F	Dummy Return	-
18h	060-063	Reserved	-
19h	064-067	Boot	Service
1Ah	068-06B	Time-of-Day Services	Service
1Bh	06C-06F	Keyboard Break	Hook
1Ch	070-073	Timer Tick	Hook
1Dh	074-077	Video Parameter Table Pointer	Table
1Eh	078-07B	Flexible Disk Parameter Table Pointer	Table
1Fh	07C-07F	Graphics Character Table Pointer	Table
20h-3Fh	080-0FF	Reserved for DOS	-
40h-49h	100-127	Reserved	-
4Ah	128-12B	Alarm Interrupt	Hook
4Bh-5Fh	12C-17F	Reserved	-
60h-61h	180-187	System Manager Interrupts	
62h	188-18B	Reserved	
63h	18C-18F	XIP Services	
64h-6Fh	190-1BF	Reserved	-
70h	1C0-1C3	Real-Time Clock Interrupt	ISR
71h-F0h	1C4-3C3	Reserved	-
F1h-FFh	3C4-3FF	Not Used	-

Most Jaguar interrupts have the same function as the corresponding PSA interrupt. (p 28 of Phoenix

BIOS RAM Definition

manual). Interrupts with different functions from the PSA BIOS definition are shown in the following table:

Int (Hex)	Jaguar Function	IBM-XT Function	Type	IRQ
02h	Low Batt & Module Pulled	RAM Parity Errors	Non-Maskable	NMI
06h	Low Power Hook	Reserved	Hook	
09h	Keyboard	Keyboard	Hardware*	1
0Ah	Miscellaneous	Reserved	Hardware	2
0Bh	Kbd & Touch Panel	COM2	Hardware	3
0Dh	IR Input	Hard Disk	Hardware	5
0Eh	XINT pin	Floppy Disk	Hardware	6
0Fh	RTC	LPT	Hardware	7

*INT 9h is a hardware interrupt in the IBM XT, but it is invoked by software in Jaguar. After an INT 0Bh, software scans and debounces the keyboard and writes the keycode to the keycode register (60h). Then it invokes INT 09h.

BIOS Data Area. The BIOS data area is also set up by the BIOS initialization code. Jaguar RAM definitions are similar to PSA XT BIOS definitions, except as noted below.

BIOS Data Area Definitions

Address	Length (Bytes)	Description
40:00h	8	I/O address of up to 4 serial communications ports
40h:08h	6	I/O address of up to 3 parallel ports. Set to all 00s in Jaguar.
40h:0Eh	2	Not used
40h:10h	2	Equipment variable, where: bit definition 15-14 Number of printer adapters 13-12 Reserved 11-9 Number of RS-232 Adapters 8 Reserved 7-6 Number of disk drives where 00b=1 drive 01b=2 drives 5-4 Initial video mode (11b in Jaguar) 3-2 Installed Memory Size (11 = at least 256k installed) 1 1 if Math coprocessor installed 0 1 if disk installed
40h:12h	1	Reserved
40h:13h	2	Installed memory in Kilobytes
40h:15h	2	Reserved
40h:17h	1	Keyboard flag 1, where: bit definition 7 1=Insert active 6 1=Caps Lock active 5 1=Num Lock active 4 1=Scroll Lock active 3 1=Alt pressed 2 1=Ctrl pressed 1 1=Left shift pressed 0 1=Right shift pressed
40h:18h	1	Keyboard flag 2, where: bit definition 7 1=Insert pressed 6 1=Caps Lock pressed 5 1=Num Lock pressed 4 1=Scroll Lock Pressed 3 1=Ctrl-Num Lock state active 2 1=Sys Req pressed 1 1=Left Alt pressed 0 1=Left Ctrl pressed
40h:19h	1	Alt-key, keypad buffer
40h:1Ah	2	Key buffer read pointer
40h:1Ch	2	Key buffer write pointer

BIOS RAM Definition

BIOS Data Area Definitions, continued

Address	Length (Bytes)	Description
40h:1Eh	32	Key buffer (16 words)
40h:3Eh	1	Floppy recalibrate status (not used in Jaguar)
40h:3Fh	1	Floppy motor status (not used in Jaguar)
40h:40h	1	Floppy motor time-out count (not used in Jaguar)
40h:41h	1	Disk status return code where: bit definition 7 1=Drive not ready 6 1=seek error occurred 5 1=disk ctrlr failed 4-0 Error codes, where: 00h=No error 01h=Illegal function was requested 02h=Address mark not found 03h=Write protect error 04h=Sector not found 06h=Drive door was opened 08h=DMA overrun error (not used in Jaguar) 09h=DMA boundary error (not used in Jaguar) 0Ch=Media type unknown 10h=CRC failed on disk read
40h:42h	7	Floppy controller status and command bytes (not used in Jaguar)
40h:49h	1	Video mode setting
40h:4Ah	2	Number of columns on screen
40h:4Ch	2	Video buffer length (bytes)
40h:4Eh	1	Offset address of current display page
40h:50h	16	Cursor coordinates for 8 pages. Two bytes each page. First byte of each pair is column, second byte is row. (0,0) is upper left corner of screen.
40h:60h	2	Cursor size. 1st byte=end scan line, 2nd byte=start scan line
40h:62h	1	Current display page number
40h:63h	2	Base I/O address of video controller
40h:65h	1	Display controller mode select register copy
40h:66h	1	Display controller palette register copy
40h:67h	4	Reserved
40h:6Bh	4	Reserved
40h:6Ch	4	Timer count - number of ticks since midnight
40h:70h	1	24 hour rollover flag
40h:71h	1	Ctrl-Brk flag (bit 7 = 1 : <Ctrl> <Break> pressed)

BIOS Data Area Definitions, continued

Address	Length (Bytes)	Description
40h:72h	2	Warm start flag: 1234h means warmstart
40h:74h	1	Hard disk status (not used in Jaguar)
40h:75h	1	Number of hard drives (set to 0 in Jaguar)
40h:76h	1	Hard disk control byte copy (not used in Jaguar)
40h:77h	1	Hard disk controller port offset (not used in Jaguar)
40h:78h	3	Parallel printer time-out table. Ports 0-2.
40h:7Bh	1	Halt Value. Used for return time out count of Int 15h service 41h
40h:7Ch	4	Serial port time-out table. Ports 0-3. (Only port 1 used in Jaguar)
40h:80h	2	Offset of Key buffer
40h:82h	2	Offset of first byte after key buffer
40h:84h	1	Number of video rows -1 EGA mode. (not used in Jaguar)
40h:85h	2	Character height EGA mode. (not used in Jaguar)
40h:87h	1	Video control bits EGA mode (not used in Jaguar)
40h:88h	1	EGA/VGA switch data(not used in Jaguar)
40h:89h	1	EGA/VGA control bits(not used in Jaguar)
40h:8Ah	1	Index into DCC table VGA mode (not used in Jaguar)
40h:8Bh	1	Last floppy data rate selected (not used in Jaguar)
40h:8Ch	1	Hard disk controller status copy (not used in Jaguar)
40h:8Dh	1	Hard disk error status copy (not used in Jaguar)
40h:8Eh	1	Hard disk interrupt flag (not used in Jaguar)
40h:8Fh	1	Hard disk controller flag (not used in Jaguar)
40h:90h	2	Floppy drive 0/1 media state (not used in Jaguar)
40h:92h	2	Floppy drive 0/1 operation state (not used in Jaguar)
40h:94h	2	Floppy drive 0/1 track number (not used in Jaguar)
40h:96h	1	Keyboard flag 3 (not used in Jaguar)

BIOS RAM Definition

BIOS Data Area Definitions, continued

Address	Length (Bytes)	Description
40h:97h	1	Keyboard LED flag (not used in Jaguar)
40h:98h	4	Vector to user wait flag (not used in Jaguar)
40h:9Ch	4	User wait count (low word, high word order) (not used in Jaguar)
40h:A0h	1	Wait active flag (not used in Jaguar)
40h:A1h	2	Number of timer ticks until display timeout
40h:A3h	2	Display time out reset value.
40h:A5h	1	Printer status flag
40h:A6h	1	Cursor movement flag
40h:A7h	1	Card Detect Register Copy
40h:A8h	4	Pointer to table of EGA pointers (not used in Jaguar)
40h:ACH	2	CPU register checksum
40h:AEh	2	User RAM checksum
40h:B0h	2	Hopper register checksum
40h:B2h	2	Stack Segment register save location
40h:B4h	2	Stack Pointer register save location
40h:B6h	2	Day Counter. This word contains the count of times the software clock at 40h:06Ch has been set to zero.
40h:B8h	1	RTC century value in bcd.
40h:B9h	1	RTC year value in bcd
40h:BAh	1	RTC month value in bcd
40h:BBh	1	RTC day of month value in bcd
40h:BCh	1	RTC hour value in bcd
40h:BDh	1	RTC minute value in bcd
40h:BEh	1	RTC second value in bcd
40h:BFh	1	RTC daylight savings time flag
40h:C0h	1	RTC alarm hour value in bcd
40h:C1h	1	RTC alarm minute value value in bcd
40h:C2h	1	RTC alarm second value value in bcd
40h:C3h	1	RTC alarm status
40h:C6h	1	NCE[1] RAM FLAG where: 32 Indicates 2048 kbytes 16 indicates 1024 kbytes 08 indicates 512 kbytes 04 Indicates 256 kbytes 02 indicates 128 kbytes 01 indicates 64 kbytes 00 indicates 0 kbytes

BIOS RAM Definition

BIOS Data Area Definitions, continued

Address	Length (Bytes)	Description
40h:C4h	2	RTC timer value. Last value written to RTC register.
40h:C7h	1	NCE[2] RAM FLAG where: 32 indicates 2048 kbytes 16 indicates 1024 kbytes 08 indicates 512 kbytes 04 indicates 256 kbytes 02 indicates 128 kbytes 01 indicates 64 kbytes 00 indicates 0 kbytes ff indicates NCE[2] is ROM
40h:C8h	1	Low Battery Flags
40h:C9h	1	Voltage Reference value
40h:CAh	2	Keyboard output register copy
40h:CCh	1	Port Locked & System Manager Media Changed flags where: bit 7 set indicates port 1 locked bit 6 set indicates port 0 locked bits 2-5 unused bit 1 set indicates port 1 media changed bit 0 set indicates port 0 media changed
40h:CDh	1	Shift annunciator flag
40h:CEh	8	Shift annunciator save location
40h:D6h	1	Unused
40h:D7h	11	OLD BIT MAP - last bit map collected by INT 0bh
40h:E2h	22	Keyboard work area
40h:F8h	1	Last key pressed
40h:F9h	1	Key repeat counter.
40h:FAh	1	Miscellaneous Key flags.
40h:FBh	1	Mute key flags
40h:FCh	1	Number of 33 msec ticks after key press before key repeat starts.
40h:FDh	1	Number of 33 msec ticks between 'keys' during typematic key repeat.
40h:FEh	1	Unused.
40h:100h	1	Print screen status byte where: 00h = No Print Screen activity 01h = Print Screen operation in progress ffh = Previous Print Screen operation failed.

Jaguar BIOS ID Block

Jaguar BIOS ID Block

The BIOS ROM contains a block of information which encodes the identification of the machine, and indicates the date the BIOS was created. The BIOS ID Block begins at FFFF:5.

FFFF:5	Date of bios release formatted as MM/DD/YY (8 bytes)
FFFF:D	Unused
FFFF:E	System model id (FEh for jaguar)
FFFF:F	Unused

Battery Check

There are two batteries in Jaguar: the Main battery and the Backup Battery. In addition, each RAM card has a battery.

Checks are performed on each battery to detect low voltage. If a battery is has low voltage, the appropriate message is shown. The low battery messages are shown below:

- MAIN BATTERY LOW
- BACKUP BATTERY LOW
- CARD BATTERY LOW

Main Battery.

The main battery is checked at power on and once per minute while Jaguar is running. If the battery is low, the MAIN BATTERY LOW message is displayed each time jaguar is powered on. Also, if the battery drops below 2.0 volts while the machine is running, the LOW MAIN BATTERY message is displayed the first time the voltage is found below this threshold. The thresholds for the main battery are shown below.

2.5 Volts	LOW MAIN BATTERY message disabled.
2.0 Volts	LOW MAIN BATTERY message enabled.
1.8 Volts	System Shutdown to backup mode.

In other words, the LOW MAIN BATTERY message is disabled until the voltage drops below 2.0 volts. Once the message is enabled, it will be displayed each time jaguar is powered on. The message is disabled if the main battery voltage goes above 2.5 volts.

If the main battery voltage drops below 1.8 volts, the hardware causes a system shut down to backup mode.

Backup Battery.

The backup battery status is checked and displayed only at power on. The voltage from the backup battery is passed through a voltage divider, so the thresholds measured by the Hopper Chip are shifted downwards. The thresholds are shown below:

TERMINAL VOLTAGE	VOLTAGE DIVIDER OUTPUT	
3.05 Volts	2.77 Volts	LOW BACKUP BATTERY message disabled.
2.78 Volts	2.49 Volts	LOW BACKUP BATTERY message enabled.

The LOW BATTERY BATTERY message is disabled until the backup battery terminal voltage drops below 2.78 volts. Once the message is enabled, it will be displayed each time jaguar is powered on until the terminal voltage goes above 3.05 volts.

Battery Check

Card Battery.

The card battery status is tested and displayed only at power on. The thresholds are shown below:

2.60 Volts	LOW CARD BATTERY message disabled.
2.40 Volts	LOW CARD BATTERY message enabled.

Power Management

Jaguar is unique among HP CMOS calculators because power to the CPU is completely shut off when the machine is turned off. This means that the CPU registers are reset to default values whenever the machine is turned on. Furthermore the CPU starts executing instructions at a different address when it is powered on, compared to where it was running when it was powered off.

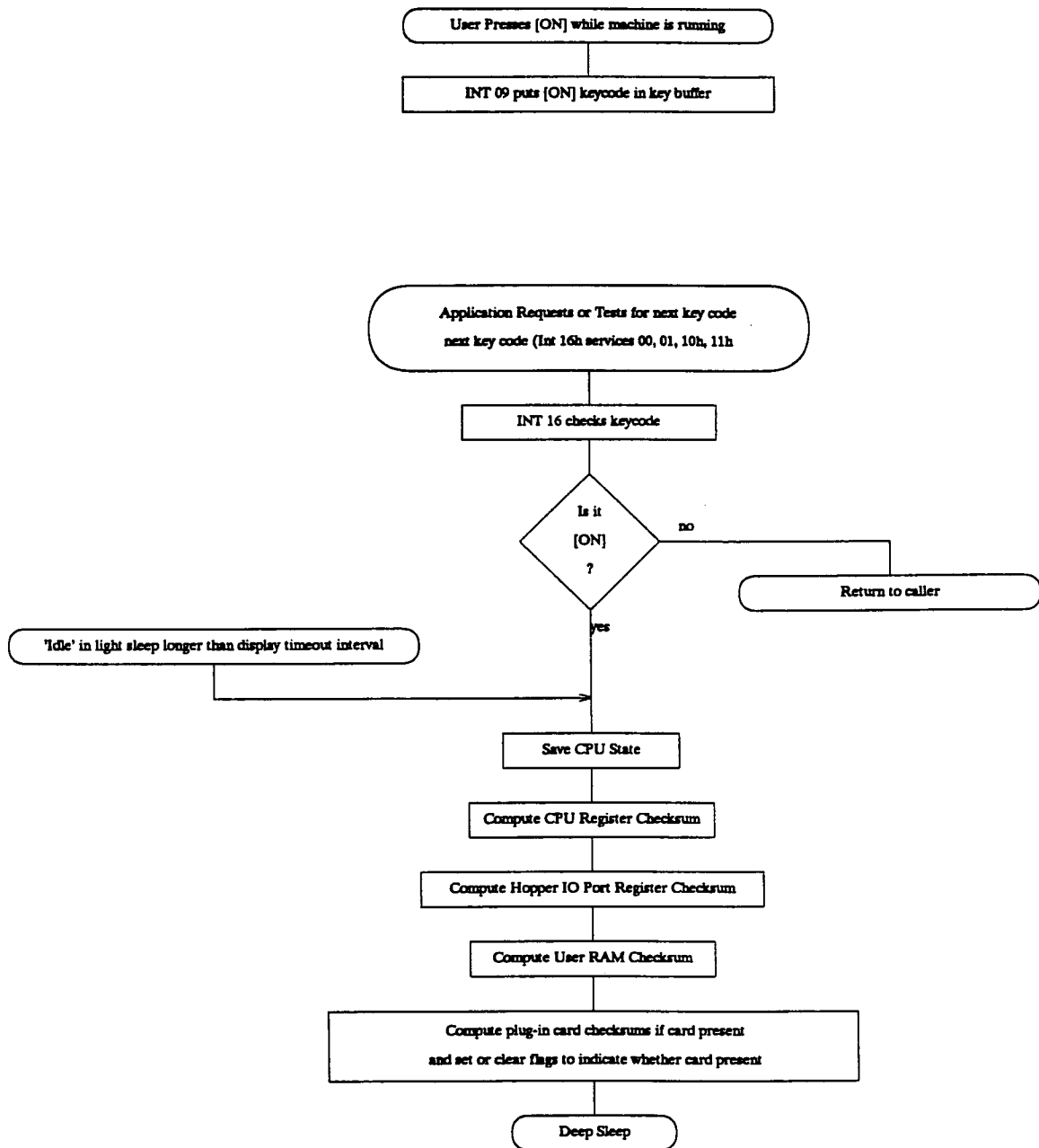
However, it is desirable from the user's viewpoint to be able to turn the machine off, then later turn it on and have it continue in the same application that was previously running. The purpose of the power management code is to perform that function. In addition, the power management code performs a number of quick checks to verify that the saved CPU registers, Hopper Memory Management registers, USER RAM and Built-In RAM DISK were not corrupted while power was off. If any of these were corrupted, the power management code will perform either a warm start or a cold start.

Power OFF. Deep Sleep is invoked when any of the following events occur:

- [ON] pressed when machine is ON.
- System Timer timed out because machine was idle during timeout interval.
- Very Low Battery Interrupt occurred.
- Application program invoked INT 15h function 42h.

The Power Down Code behavior is described by the following flowcharts:

Power Management



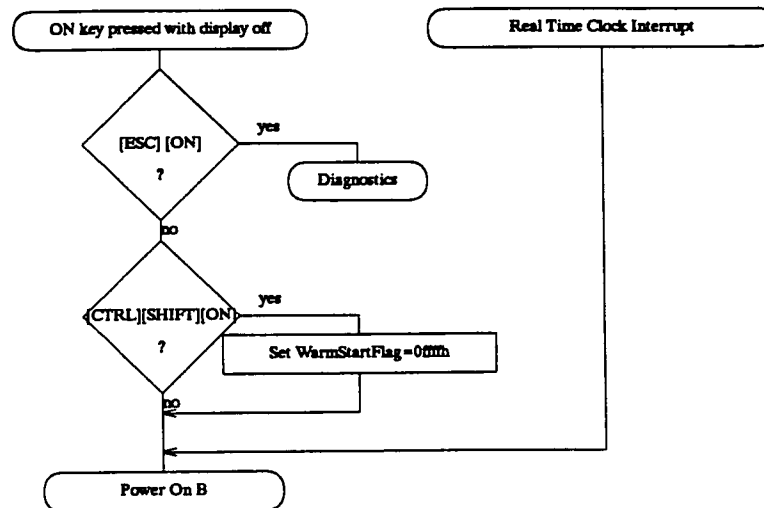
This is the normal power down sequence. All CPU registers except SS and SP are saved on the user's stack. SS and SP are saved in the BIOS data. Checksums are computed for stack area containing the CPU registers, the Hopper memory configuration registers and user RAM. These are saved for use when the machine is powered back on. The checksums of built-in RAM disk and plug-in RAM disk are computed each time the disk is written to.

Normal Power Up Behavior. The code that handles power on is accessed by the reset vector (0FFFFh:0000h). This is invoked when the ON key is pressed while the machine is off.

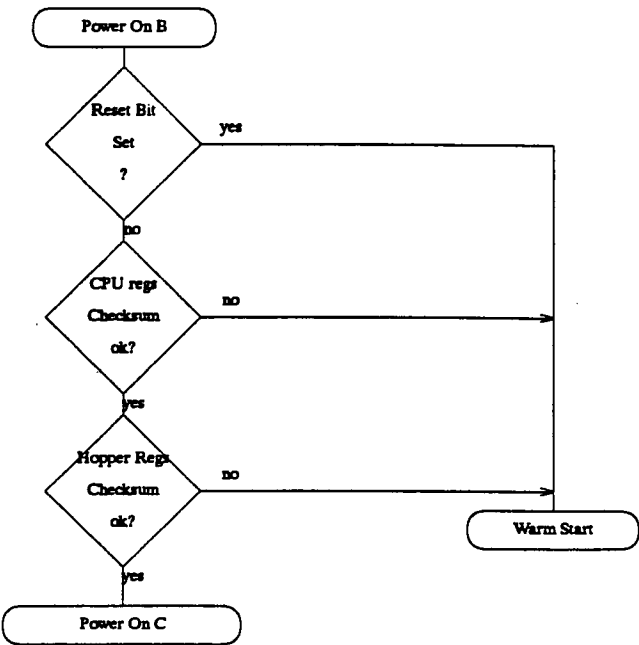
Warm Start Behavior. Warm Start is normally invoked by [CTRL][ALT][DEL]. It is also invoked if the user ram is found to be corrupted during normal power on initialization. It initializes User RAM, then invokes int 19h bootstrap loader.

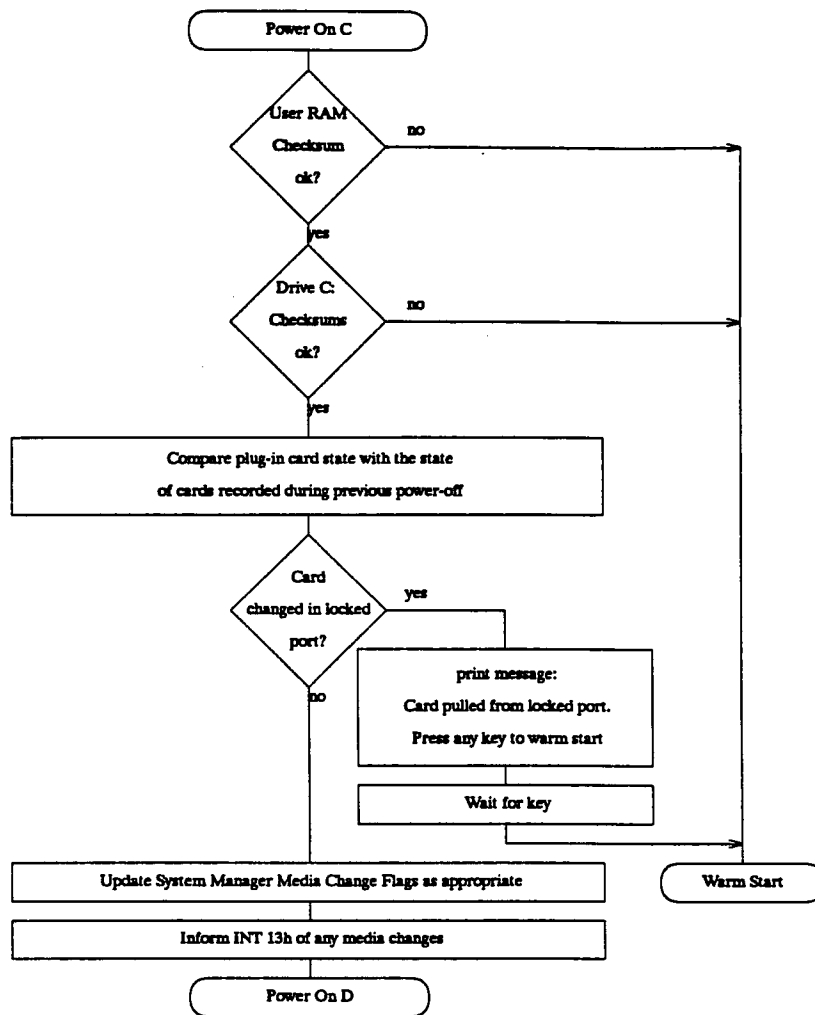
Cold Start Behavior. Cold Start is invoked by [Shift][CTRL][ON] or, if the Built-In RAM disk is found to be corrupted during normal power on. It initializes the User RAM and built in RAM DISK, but not the plug-in RAM DISK. The user is prompted to specify whether or not to blow away the in the built-in RAM disk. After all initializations are done, cold start invokes int 19h bootstrap loader.

Initialization Flow Charts. The following flow charts describe the behavior of Jaguar during Normal Power On, Warm Start and Cold Start:

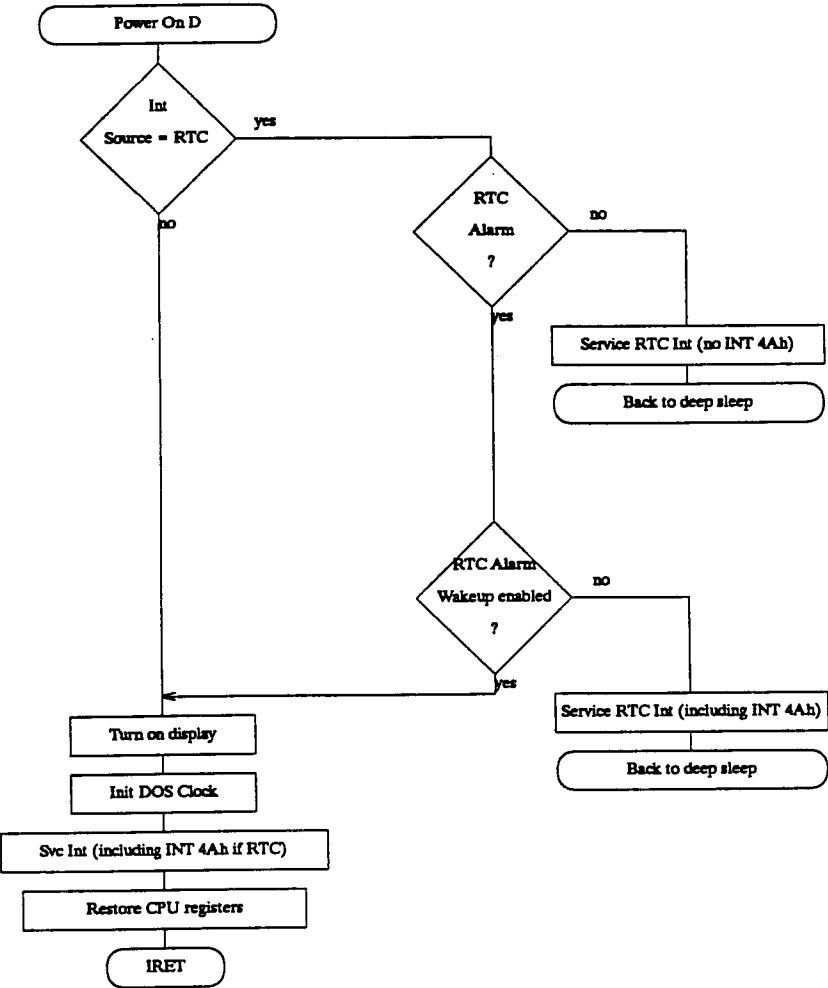


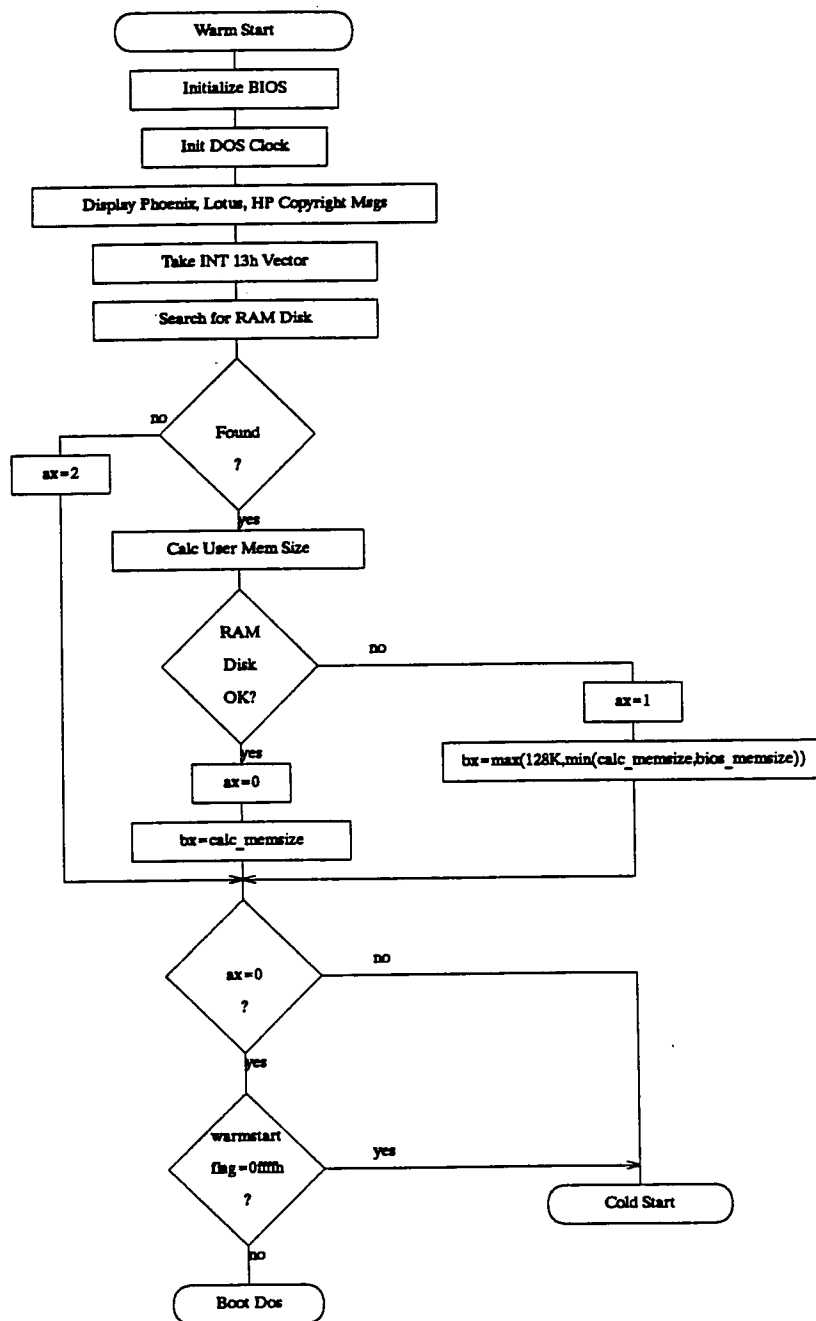
Power Management



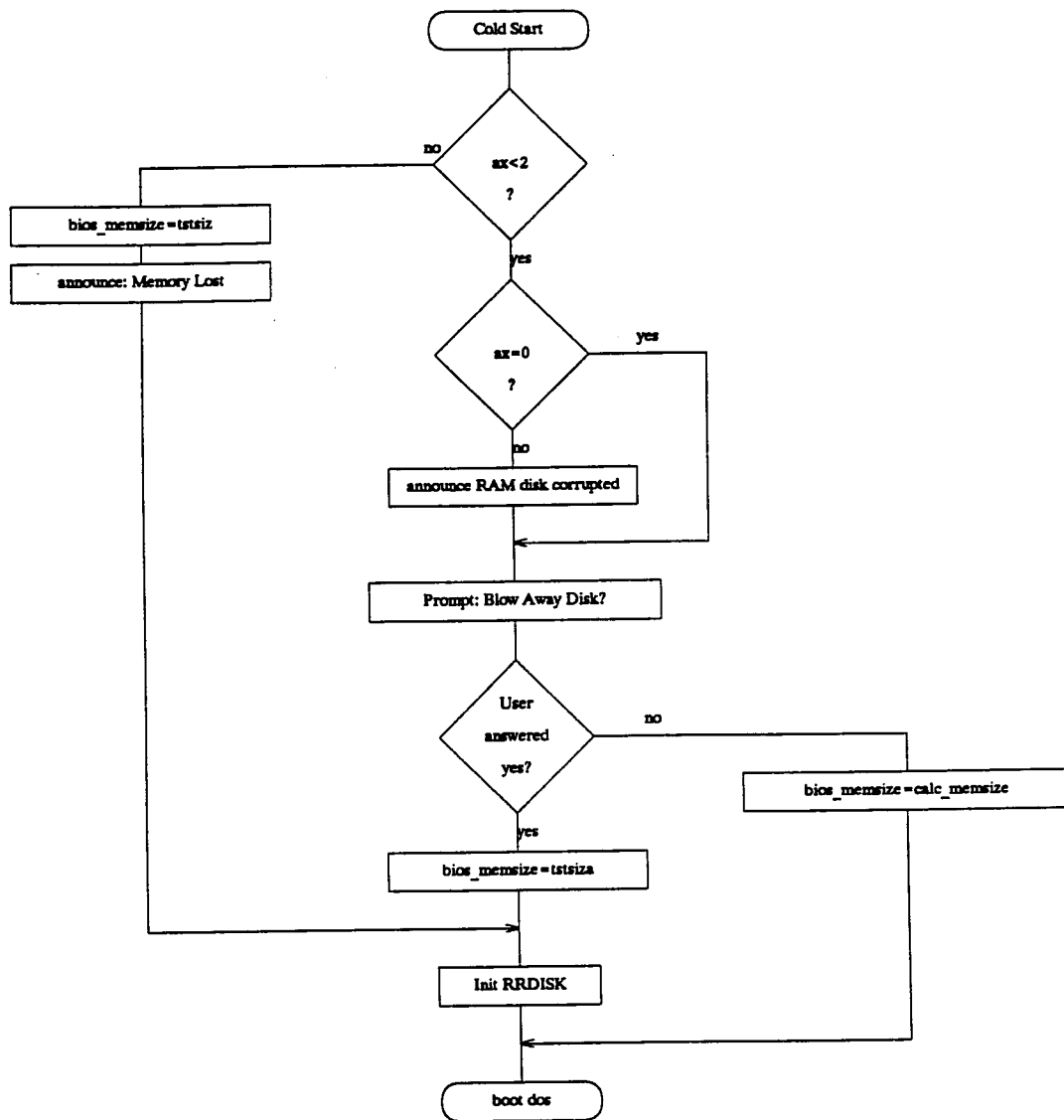


Power Management





Power Management



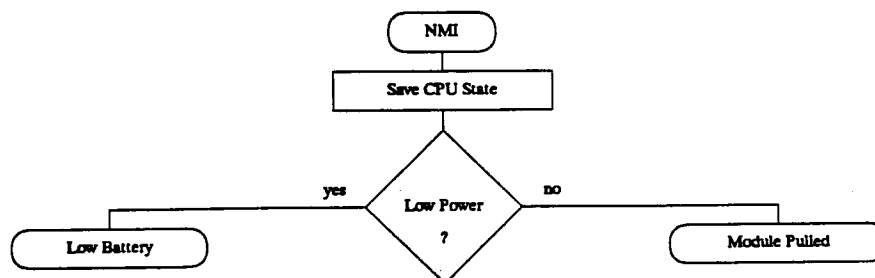
Int 02h - Nonmaskable Interrupt

The hardware nonmaskable interrupt is invoked when either a module pulled or low battery event occurs. The Nonmaskable Interrupt routine handles these two events:

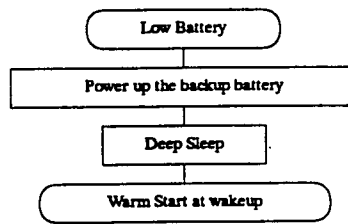
Low Battery If a low battery event occurs, Jaguar goes to deep sleep as quickly as possible. The checksums for User RAM, CPU registers, and hopper registers are not computed. A warm start will always occur on the next wake up after a low battery shut down.

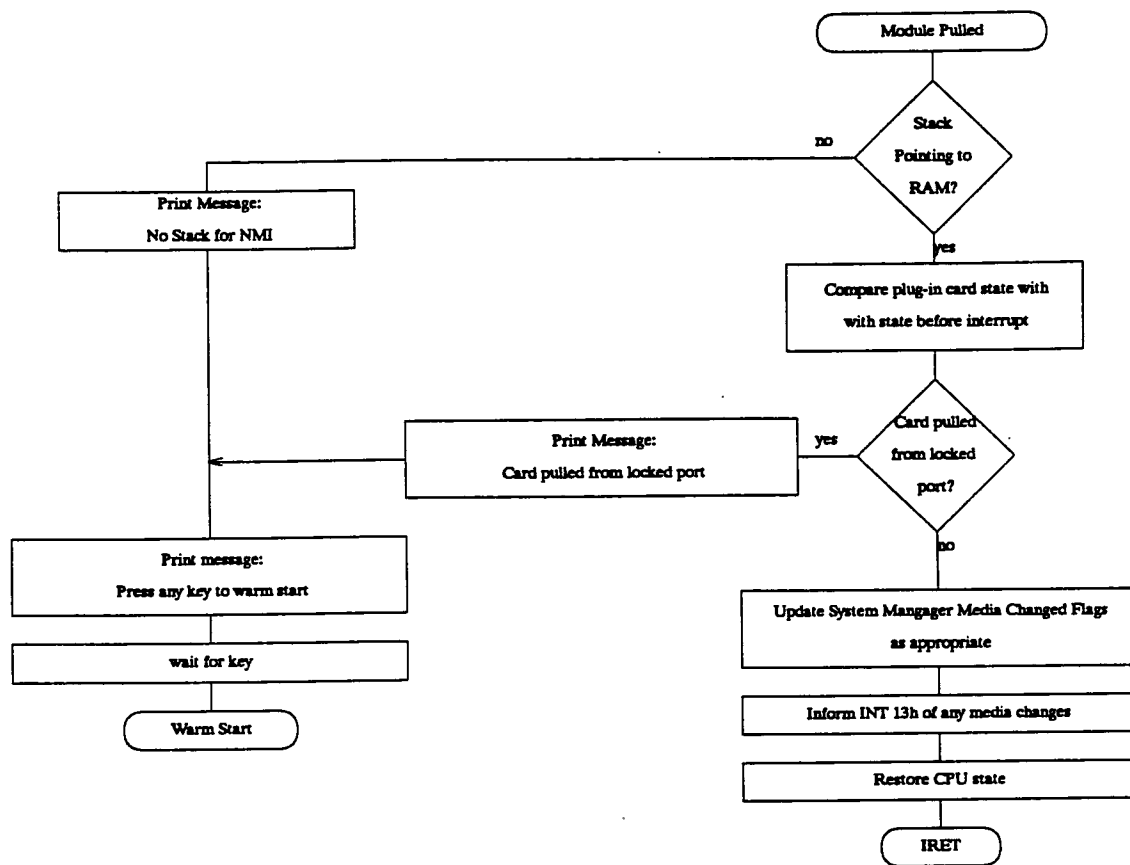
Module Pulled Interrupt. The behavior of the module pulled event depends on whether the machine is in deep sleep or not. If the machine is in deep sleep, the module pulled interrupt is disabled, and the module pulled event is not detected until the next time the machine powers on. If the machine is not in deep sleep (i.e. CPU running or in light sleep), then an interrupt is generated.

The behavior of the nonmaskable interrupt code is shown in the following flow charts:



Int 02h





Int 05h

Int 05h - Print Screen Interrupt

This interrupt executes the BIOS print screen function, causing the current screen contents to be written to serial printer port 0. The cursor position is saved before the operation is begun, and restored once the printout is complete. The Print Screen service routine can be initiated by either pressing the [Print Screen] key on the keyboard, or by issuing an Int 05h in a program. When executed, the Print Screen service routine updates a status byte at address 40h:100h. The value of this byte is interpreted as follows:

- 00h No Print Screen activity.
- 01h A Print Screen operation is in progress.
- FFh The previous Print Screen operation terminated with an error.

The Print Screen routine is not re-entrant. Additional Print Screen calls are ignored while a Print Screen operation is in progress. This prevents multiple screen printouts from being queued (for example, if the [Print Screen] key is quickly pressed more than once).

Print screen prints characters in the active display window. It will not print the entire contents of display memory.

Int 06h - Low Power Hook

This interrupt is called by the system:

- Just before entering deep sleep. (AH=0).
- Just after leaving deep sleep. (AH=1). When this hook is called, the value in AL indicates the cause of the wakeup:

AL	WAKEUP CAUSE
1	ON key press
2	UART wake up
4	Real Time Clock Alarm
8	XINT wake up

- Just before entering light sleep. (AH=2).
- Just after leaving light sleep. (AH=3).

Int 08h

Int 08h - Timer Hardware Interrupt

A periodic hardware timer interrupt occurs on hardware interrupt level 0 (IRQ0) at a rate of 18.2 times per second. IRQ0 maps to interrupt vector 08h. The BIOS interrupt service routine for Int 08h performs several housekeeping duties for the BIOS. As part of the routine, Int 1Ch (Timer Tick) is called for each hardware timer interrupt. If an application needs a periodic interrupt, it should take over the Timer Tick interrupt hook (Int 1Ch). (The default Int 1Ch service routine is just an iret.) The application program should not attempt to take over the Int 08h service routine directly. The timer tick interrupt service routine does the following:

- Increments the BIOS software clock in the double word at 40:6C. If the count equals 1800B0h then the count at 40:6C is reset and the rollover flag at 40:70 is set to 1.
- The Timer Tick Int 1Ch is called, so that a user routine can obtain a periodic call. The default handler for Int 1C is just a dummy IRET in the BIOS.
- When control is returned from Int 1C resets the 8259 programmable interrupt controller and enables interrupts.
- The timer routine handles automatic windowing of the display. If the hardware indicates that the cursor has moved, the timer routine attempts to move the display window so it contains the cursor.
- The timer routine decrements two counters that measure the time until display timeout (deep sleep) and the next battery measurement. However, the timer routine does not invoke either the deep sleep code or the battery measurement code. This is done by the light sleep code (Int 16h) when the display timeout timer or battery measurement timer has counted down to 0.
- The routine returns with an IRET

Int 09h - Keyboard Translate Interrupt

Three interrupt service routines are used to input keyboard data and to process scancodes - Int 0Bh, Int 09h and Int 16h.

- Int 0Bh is invoked when a key is pressed. It performs a software scan of the keyboard and computes the scan code each time a key is pressed or released. It places the scan code in an I/O port (060h) and invokes the Int 09h service routine.
- Int 09h obtains a single byte scancode from I/O port 060h and translates it into a two-byte key code based on the state of the control, shift and alt keys. It puts the two byte code in the keybuffer.
- Int 16h reads the two byte key code from the buffer. When key information is desired, the operating system or an application calls Int 16h, which returns the key codes in a register.

The BIOS interrupt service routine for Int 09h processes the incoming scancode as follows:

1. The routine reads the scancode from the keyboard I/O port (060h)
2. The routine calls the Keyboard Translation Hook function (Int 15h, AH = 4Fh). An application can take over this hook and insert its own handler. If this call returns with carry clear, Int 09h stops processing and returns. If carry is set, Int 09h proceeds with step 3.
3. The routine checks for a Break. If the Break is detected, Int 09h clears the keyboard input buffer and calls Int 1Bh. (A "dummy" scancode of "00h/00h" is entered into the keyboard input buffer.)
4. The Int 09h routine translates the scancode and enters the scancode and its ASCII equivalent into the keyboard input buffer. Some scancodes have no ASCII equivalent. In this case Int 09h does one of the following:
 - Discard the scancode (enter no data into the keyboard input buffer). If [Shift], [Ctrl], [Alt], [Caps Lock], [Char] or [Scroll Lock] is detected, the state of the keyboard is updated, but the scancode is discarded.
 - Enter a two-byte pair "00h/XXh" into the keyboard input buffer. "XXh" may be the original scancode, or it may be a translated hexadecimal code for the key or key combination pressed.

Once a scancode/ASCII pair has been entered into the keyboard input buffer, an application can read the data by calling Int 16h.

Char Key The [Char] works like a special function key which changes the operation of the alphabet keys. If the Char function is active, several alphabet keys return non-English characters. See the character code tables at the end of this section for a list of character codes returned when Char function is active.

The [Char] key also activates 'Mute' functions. These are special key sequences that return many non-English characters. See the table of mute functions at the end of this section.

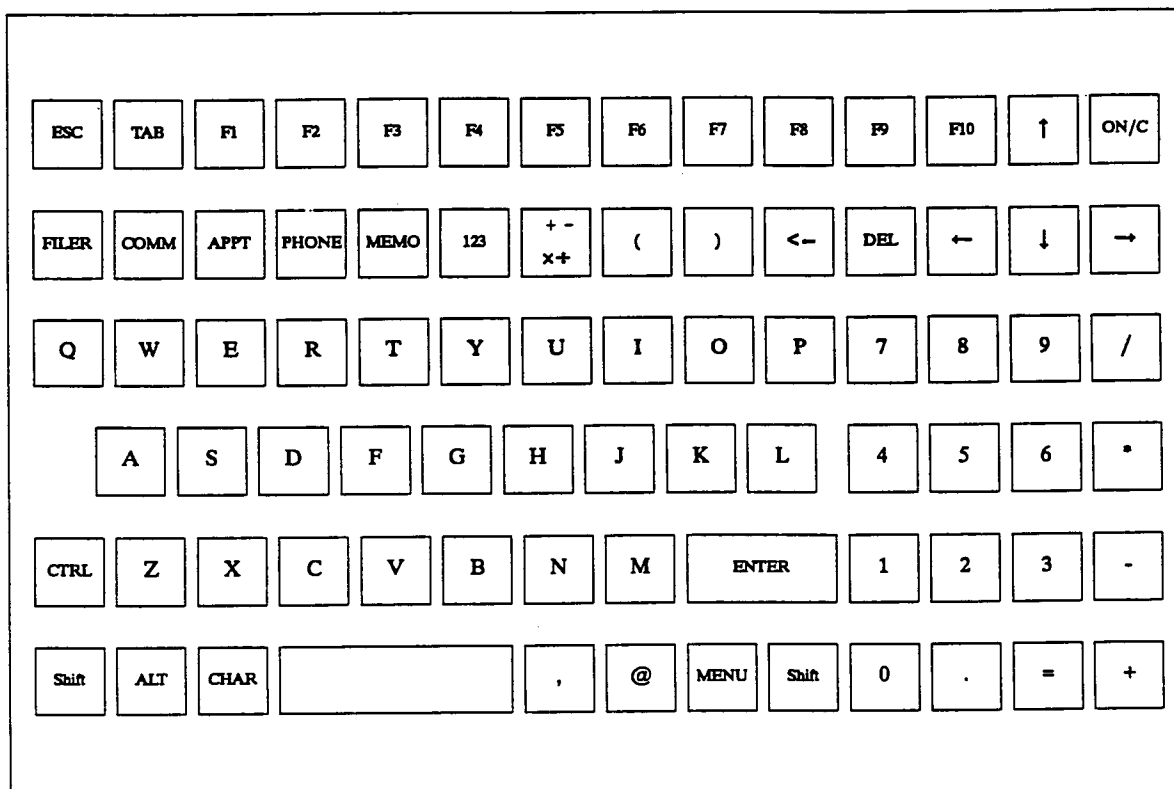
The [Char] key is 'sticky'. Pressing and releasing [Char] causes the Char function to stay on. Pressing and releasing a second time causes the Char function to turn off. If the Char state is on but the Char key is released, then pressing and releasing any other key causes the Char state to turn off. The sticky Char is hard coded. There is no user option to disable the sticky Char.

Sticky Shift. The [Shift] key is also 'sticky'. It works much like the Char key. Pressing and releasing [Shift] causes the shift state to stay on. Pressing and releasing a second time causes the shift state to turn off. If the shift state is on but the shift key is released, then pressing and releasing any other key causes the shift state to turn off.

Int 09h

When the Shift key is released and the Shift State is ON, an annunciator is displayed in the lower right portion of the display. The annunciator is removed on the next key press.

Key Cap Legends. The following diagram shows key cap legends for the Jaguar Keyboard.



Scancode Conversion Tables. Int 09h translates scancodes to ASCII character codes or other hexadecimal codes as shown in the tables below. The first table shows scancodes returned when the Char function is OFF. The second table shows scancodes returned when the Char function is ON. For each scancode, the tables give the equivalent character codes for each keyboard state: normal, shifted, [Ctrl] active, and [Alt] active. If a scancode has an ASCII equivalent, the ASCII character is returned in register AL. If a converted code of the form "xxh/00" is shown, a zero value is returned in register AL to indicate that there is no ASCII value for the key combination. BIOS Int 16h returns the value "XXh" in register AH.

Note: Some shifted characters on Jaguar are non-shifted on standard IBM keyboards and some non-shifted characters on Jaguar are shifted on standard IBM keyboards. For example,

- [()] is non-shifted on Jaguar and shifted on IBM
- [Home] is shifted on Jaguar and non-shifted on IBM.

On these non-compatible key mappings, Int 0Bh will force the status of the shift bits in 40h:17h to the state used by a compatible keyboard, regardless of the actual state of the shift keys. For example:

- [()] is reported to Int 09h as a shifted character even though it is non-shifted on Jaguar.

- [Home] is reported to Int 09h as a non-shifted character even though it is shifted on Jaguar.

In IBM compatible machines, the [Shift], [Ctrl], and [Alt] keys effect the interpretation of a scancode with ascending priority. That is, the [Alt] key has the highest priority. If [Alt] is pressed, the [Shift] and [Ctrl] keys have no effect. Likewise, if [Ctrl] is pressed, the [Shift] key has no affect. The only valid combination involving both [Ctrl] and [Alt] is the Warm Start sequence [CTRL][ALT][DEL].

In Jaguar, however, there are some valid combinations of [Shift] and [Ctrl]. In particular:

- [Shift][Ctrl][PGUP] yields the compatible keycode for [Ctrl][PGUP]
- [Shift][Ctrl][HOME] yields the compatible keycode for [Ctrl][HOME]
- [Shift][Ctrl][PGDN] yields the compatible keycode for [Ctrl][PGDN]
- [Shift][Ctrl][END] yields the compatible keycode for [Ctrl][END]
- [Shift][Ctrl][] yields the compatible keycode for [Ctrl][]
- [Shift][Ctrl][] yields the compatible keycode for [Ctrl][]

Int 09h

SCANCODE TABLE WHEN CHAR FUNCTION IS OFF

KEY LEGEND		SCANCODE		CHARACTER CODES (hex)				
				Normal	Shifted	Control	Alt	Shift+Control
Normal	Shifted	Normal	Shifted	ah/al	ah/al	ah/al	ah/al	ah/al
ESC	PrtScr	01	37	01/1B	PrtScr	01/1B		
TAB		0F	0F	0F/09	0F/00	94/00	A5/00	94/00
F1		3B	3B	3B/00	54/00	5E/00	68/00	5E/00
F2		3C	3C	3C/00	55/00	5F/00	69/00	5F/00
F3		3D	3D	3D/00	56/00	60/00	6A/00	60/00
F4		3E	3E	3E/00	57/00	61/00	6B/00	61/00
F5		3F	3F	3F/00	58/00	62/00	6C/00	62/00
F6		40	40	40/00	59/00	63/00	6D/00	63/00
F7		41	41	41/00	5A/00	64/00	6E/00	64/00
F8		42	42	42/00	5B/00	65/00	6F/00	65/00
F9		43	43	43/00	5C/00	66/00	70/00	66/00
F10		44	44	44/00	5D/00	67/00	71/00	67/00
↑	PGUP‡	48	49	48/00	49/00	8D/00		84/00
ON/OFF	ON/OFF	70	70			A2/00		
FILER	SETUP	72	71	A8/00	A4/00	AE/00	AB/00	AA/00
COMM	‡	73	29	AC/00	29/60	B2/00	AF/00	
APPT	~	74	29	B0/00	29/7E	B6/00	B3/00	
PHONE	!	75	02	B4/00	02/21	BA/00	B7/00	
MEMO	#	76	04	B8/00	04/23	BE/00	BB/00	
123	\$	77	05	BC/00	05/24	C2/00	BF/00	
+ -	&	78	08	C0/00	08/26	C6/00	C3/00	
x÷								
(†		0A	2B	0A/28	2B/7C		80/00	2B/1C
)†	\†	0B	2B	0B/29	2B/5C		81/00	2B/1C
<--	<--	0E	0E	0E/08	0E/08	BREAK	0E/00	BREAK
DEL	INS‡	53	52	53/00	52/00	93/00		92/00
←	HOME‡	4B	47	4B/00	47/00	73/00		77/00
↓	PGDN‡	50	51	50/00	51/00	91/00		76/00
→	END‡	4D	4F	4D/00	4F/00	74/00		75/00
Q		10	10	10/71	10/51	10/11	10/00	10/11
W		11	11	11/77	11/57	11/17	11/00	11/17
E		12	12	12/65	12/45	12/05	12/00	12/05
R		13	13	13/72	13/52	13/12	13/00	13/12
T		14	14	14/74	14/54	14/14	14/00	14/14
Y		15	15	15/79	15/59	15/19	15/00	15/19
U		16	16	16/75	16/55	16/15	16/00	16/15
I		17	17	17/69	17/49	17/09	17/00	17/09
O		18	18	18/6F	18/4F	18/0F	18/00	18/0F
P		19	19	19/70	19/50	19/10	19/00	19/10
7	[†	08	1A	08/37	1A/5B			1A/1B
8]†	09	1B	09/38	1B/5D			1B/1D
9	{	0A	1A	0A/39	1A/7B			1A/1B
/	}	35	1B	35/2F	1B/7D			1B/1D

† Character Code reported to INT 09h as a shifted character.

‡ Character Code reported to INT 09h as an unshifted character.

SCANCODE TABLE WHEN CHAR FUNCTION IS OFF

KEY LEGEND		SCANCODE		CHARACTER CODES (hex)				
				Normal	Shifted	Control	Alt	Shift + Control
Normal	Shifted	Normal	Shifted	ah/al	ah/al	ah/al	ah/al	ah/al
A		1E	1E	1E/61	1E/41	1E/01	1E/00	1E/01
S		1F	1F	1F/73	1F/53	1F/13	1F/00	1F/13
D		20	20	20/64	20/44	20/04	20/00	20/04
F		21	21	21/66	21/46	21/06	21/00	21/06
G		22	22	22/67	22/47	22/07	22/00	22/07
H		23	23	23/68	23/48	23/08	23/00	23/08
J		24	24	24/6A	24/4A	24/0A	24/00	24/0A
K		25	25	25/6B	25/4B	25/0B	25/00	25/0B
L		26	26	26/6C	26/4C	26/0C	26/00	26/0C
4	†	05	27	05/34	27/3B			
5	:	06	27	06/35	27/3A			
6	†	07	28	07/36	28/27	07/1E		
*	"	37	28	37/2A	28/22	96/00	37/00	
CTRL		1D	1D					
Z		2C	2C	2C/7A	2C/5A	2C/1A	2C/00	2C/1A
X		2D	2D	2D/78	2D/58	2D/18	2D/00	2D/18
C		2E	2E	2E/63	2E/43	2E/03	2E/00	2E/03
V		2F	2F	2F/76	2F/56	2F/16	2F/00	2F/16
B		30	30	30/62	30/42	30/02	30/00	30/02
N		31	31	31/6E	31/4E	31/0E	31/00	31/0E
M		32	32	32/6D	32/4D	32/0D	32/00	32/0D
ENTER		1C	1C	1C/0D	1C/0D	1C/0A	1C/00	1C/0A
1	<	02	33	02/31	33/3C			
2	>	03	34	03/32	34/3E	03/00		
3	?	04	35	04/33	35/3F			
-	^	4A	07	4A/2D	07/5E	8E/00		07/1E
Shift		2A	2A					
ALT		38	38					
Char		79	79					
<space>		39	39	39/20	39/20	39/20	39/20	39/20
,	†	33	33	33/2C	33/2C		33/00	
@†	@	03	03	03/40	03/40	03/00	79/00	03/00
MENU		7A	7A	C8/00	C9/00	CA/00	CB/00	CA/00
Shift		36	36					
0	CAPS†	0B	3A	0B/30				
.	SCRL†	34	46	34/2E			34/00	BREAK
=		0D	0C	0D/3D	0C/5F		83/00	0C/1F
+	%	4E	06	4E/2B	06/25	90/00		

† Character Code reported to INT 09h as a shifted character.

‡ Character Code reported to INT 09h as an unshifted character.

Int 09h

SCANCODE TABLE WHEN CHAR FUNCTION IS ON

KEY LEGEND		SCANCODE		CHARACTER CODES (hex)				
Normal	Shifted	Normal	Shifted	Normal ah/al	Shifted ah/al	Control ah/al	Alt ah/al	Shift + Control ah/al
ESC	PrtScr	01	37	01/1B	PrtScr	01/1B		
TAB		0F	0F	0F/09	0F/00	94/00		94/00
F1		3B	3B	DB/00	F4/00	5E/00		5E/00
F2		3C	3C	DC/00	F5/00	5F/00	69/00	5F/00
F3		3D	3D	DD/00	F6/00	60/00	6A/00	60/00
F4		3E	3E	DE/00	F7/00	61/00	6B/00	61/00
F5		3F	3F	DF/00	F8/00	62/00	6C/00	62/00
F6		40	40	E0/00	F9/00	63/00	6D/00	63/00
F7		41	41	E1/00	FA/00	64/00	6E/00	64/00
F8		42	42	E2/00	FB/00	65/00	6F/00	65/00
F9		43	43	E3/00	FC/00	66/00	70/00	66/00
F10		44	44	E4/00	FD/00	67/00	71/00	67/00
↑	PGUP†	48	49	48/00	49/00	8D/00		84/00
ON/OFF	ON/OFF	70	70			A2/00		
FILER	SETUP	72	71			AE/00	AB/00	AA/00
COMM	†	73	29	04/EF	29/60	B2/00		
APPT	-	74	29	05/F9	29/7E	B6/00		
PHONE	!	75	02	06/AD	02/AD	BA/00	B7/00	
MEMO	#	76	04	07/EE	04/23	BE/00	BB/00	
123	\$	77	05	08/B8	05/24	C2/00	BF/00	
+ - x ÷	&	78	08	09/A9	08/26	C6/00	C3/00	
(†		0A	2B	0A/DD	2B/7C		80/00	2B/1C
)†	\†	0B	2B	0B/29	2B/5C		81/00	2B/1C
<--	<--	0E	0E	0E/08	0E/08	BREAK	0E/00	BREAK
DEL	INS†	53	52	53/00	52/00	93/00		92/00
←	HOME†	4B	47	4B/00	47/00	73/00		77/00
↓	PGDN†	50	51	50/00	51/00	91/00		76/00
→	END†	4D	4F	4D/00	4F/00	74/00		75/00
Q		10	10	10/A6	10/A6	10/11	10/00	10/11
W		11	11	11/A7	11/A7	11/17		11/17
E		12	12	12/91	12/92	12/05		12/05
R		13	13	*****	*****			
T		14	14	*****	*****			
Y		15	15	*****	*****			
U		16	16	*****	*****			
I		17	17	*****	*****			
O		18	18	18/9B	18/9D	18/0F	18/00	18/0F
P		19	19	19/E7	19/E8	19/10	19/00	19/10
7	[†	08	1A	08/AC	1A/5B			1A/1B
8]†	09	1B	09/AB	1B/5D			1B/1D
9	{	0A	1A	0A/F3	1A/7B			1A/1B
/	}	35	1B	35/F6	1B/7D			1B/1D

† Character Code reported to INT 09h as a shifted character.

***** Mute function enabled.

‡ Character Code reported to INT 09h as an unshifted character.

SCANCODE TABLE WHEN CHAR FUNCTION IS ON

KEY LEGEND		SCANCODE		CHARACTER CODES (hex)				
				Normal	Shifted	Control	Alt	Shift+Control
Normal	Shifted	Normal	Shifted	ah/al	ah/al	ah/al	ah/al	ah/al
A		1E	1E	1E/86	1E/8F	1E/01		1E/01
S		1F	1F	1F/E1	1F/E1	1F/13		1F/13
D		20	20	20/D0	20/D1	20/04	20/00	20/04
F		21	21	21/9F	21/9F	21/06	21/00	21/06
G		22	22	22/CF	22/CF	22/07	22/00	22/07
H		23	23	23/BE	23/BE	23/08	23/00	23/08
J		24	24	24/24	24/24	24/0A	24/00	24/0A
K		25	25	25/BD	25/BD	25/0B	25/00	25/0B
L		26	26	26/9C	26/9C	26/0C	26/00	26/0C
4	‡	05	27	05/34	27/3B			
5	:	06	27	06/35	27/3A			
6	‡	07	28	07/36	28/27	07/1E		
*	"	37	28	37/9E	28/22	96/00	37/00	
CTRL		1D	1D					
Z		2C	2C	2C/F4	2C/F4	2C/1A		2C/1A
X		2D	2D	2D/F8	2D/F8	2D/18		2D/18
C		2E	2E	2E/87	2E/80	2E/03	2E/00	2E/03
V		2F	2F	2F/F5	2F/F5	2F/16	2F/00	2F/16
B		30	30	30/FE	30/FE	30/02	30/00	30/02
N		31	31	31/D5	31/D5	31/0E	31/00	31/0E
M		32	32	32/E6	32/E6	32/0D	32/00	32/0D
ENTER		1C	1C	1C/0D	1C/0D	1C/0A	1C/00	1C/0A
1	<	02	33	02/AE	33/FB			
2	>	03	34	03/AF	34/FD	03/00		
3	?	04	35	04/A8	35/FC			
-	^	4A	07	4A/F0	07/5E	8E/00		07/1E
Shift		2A	2A					
ALT		38	38					
Char		79	79					
<space>		39	39	39/20	39/20	39/20	39/20	39/20
,	‡	33	33	33/F7	33/F7		33/00	
@†	@	03	03	03/40	03/40	03/00	79/00	03/00
MENU		7A	7A		C9/00	CA/00	CB/00	CA/00
Shift		36	36					
0	CAPS‡	0B	3A	0B/30				
.	SCRL‡	34	46	34/FA			34/00	BREAK
=		0D	0C	0D/F2	0C/5F		83/00	0C/1F
+	%	4E	06	4E/F1	06/25	90/00		

† Character Code reported to INT 09h as a shifted character.

‡ Character Code reported to INT 09h as an unshifted character.

MUTE KEY SEQUENCES

KEY SEQUENCE	CHARACTER CODES (hex)	
	Normal ah/al	Shifted ah/al
[Char][r][a]	1E/A0	1E/B5
[Char][r][e]	12/82	12/90
[Char][r][i]	17/A1	17/D6
[Char][r][o]	18/A2	18/E0
[Char][r][u]	16/A3	16/E9
[Char][r][y]	15/EC	15/ED
[Char][r][n]	31/6E	31/4E
[Char][t][a]	1E/85	1E/B7
[Char][t][e]	12/8A	12/D4
[Char][t][i]	17/8D	17/DE
[Char][t][o]	18/95	18/E3
[Char][t][u]	16/97	16/EB
[Char][t][y]	15/79	15/59
[Char][t][n]	31/6E	31/4E
[Char][y][a]	1E/83	1E/B6
[Char][y][e]	12/88	12/D2
[Char][y][i]	17/8C	17/D7
[Char][y][o]	18/93	18/E2
[Char][y][u]	16/96	16/EA
[Char][y][y]	15/79	15/59
[Char][y][n]	31/6E	31/4E
[Char][u][a]	1E/84	1E/8E
[Char][u][e]	12/89	12/D3
[Char][u][i]	17/8B	17/D8
[Char][u][o]	18/94	18/99
[Char][u][u]	16/81	16/9A
[Char][u][y]	15/98	15/59
[Char][u][n]	31/6E	31/4E
[Char][i][a]	1E/C6	1E/C7
[Char][i][e]	12/65	12/45
[Char][i][i]	17/69	17/49
[Char][i][o]	18/E4	18/E5
[Char][i][u]	16/75	16/55
[Char][i][y]	15/79	15/59
[Char][i][n]	31/A4	31/A5

The following table gives Jaguar key sequences to obtain character codes 80h through 0ffh. Most characters are assigned to a [CHAR] sequence. However the drawing characters are not; they must be entered via [ALT] [decimal keycode] sequences.

Note that some [CHAR] sequences require the [SHIFT] key to be pressed. In these sequences, the [CHAR] and [SHIFT] keys may be pressed in either order, [CHAR] first or [SHIFT] first.

Character Code (hex)	Character Name	Key Sequence
80h	C cedilla	[CHAR][SHIFT][C]
81h	u diaeresis	[CHAR][U][u]
82h	e acute	[CHAR][R][e]
83h	a circumflex	[CHAR][Y][a]
84h	a diaeresis	[CHAR][U][a]
85h	a grave	[CHAR][T][a]
86h	a ring	[CHAR][a]
87h	c cedilla	[CHAR][c]
88h	e circumflex	[CHAR][Y][e]
89h	e diaeresis	[CHAR][U][e]
8Ah	e grave	[CHAR][T][e]
8Bh	i diaeresis	[CHAR][U][i]
8Ch	i circumflex	[CHAR][Y][i]
8Dh	i grave	[CHAR][T][i]
8Eh	A diaeresis	[CHAR][SHIFT][U][A]
8Fh	A ring	[CHAR][SHIFT][A]
90h	E acute	[CHAR][SHIFT][R][E]
91h	a ligature	[CHAR][e]
92h	A ligature	[CHAR][SHIFT][E]
93h	o circumflex	[CHAR][Y][o]
94h	o diaeresis	[CHAR][U][o]
95h	o grave	[CHAR][T][o]
96h	u circumflex	[CHAR][Y][u]
97h	u grave	[CHAR][T][u]
98h	y diaeresis	[CHAR][U][y]
99h	O diaeresis	[CHAR][SHIFT][U][O]
9Ah	U diaeresis	[CHAR][SHIFT][U][U]
9Bh	o with oblique stroke	[CHAR][o]
9Ch	Pound	[CHAR][L]
9Dh	O with oblique stroke	[CHAR][SHIFT][O]
9Eh	multiply sign	[CHAR][*]
9Fh	Guilder	[CHAR][F]

Character Code (hex)	Character Name	Key Sequence
A0h	a acute	[CHAR][R][a]
A1h	i acute	[CHAR][R][i]
A2h	o acute	[CHAR][R][o]
A3h	u acute	[CHAR][R][u]
A4h	n tilde	[CHAR][T][n]
A5h	N tilde	[CHAR][SHIFT][T][N]
A6h	Feminine ordinal	[CHAR][q]
A7h	Masculine ordinal	[CHAR][w]
A8h	upside down ?	[CHAR][3]
A9h	registered trademark sign	[CHAR][HP CALC]
AAh		[ALT][1][7][0]
ABh	1/2	[CHAR][8]
ACh	1/4	[CHAR][7]
ADh	upside down !	[CHAR][PHONE]
AEnh	Left French quote	[CHAR][1]
AFh	Right French Quote	[CHAR][2]
B0h		[ALT][1][7][6]
B1h		[ALT][1][7][7]
B2h		[ALT][1][7][8]
B3h		[ALT][1][7][9]
B4h		[ALT][1][8][0]
B5h	A acute	[CHAR][SHIFT][R][A]
B6h	A circumflex	[CHAR][SHIFT][Y][A]
B7h	A grave	[CHAR][SHIFT][T][A]
B8h	copyright sign	[CHAR][LOTUS 123]
B9h		[ALT][1][8][5]
BAh		[ALT][1][8][6]
BBh		[ALT][1][8][7]
BCh		[ALT][1][8][8]
BDh	Cents sign	[CHAR][K]
BEh	Yen sign	[CHAR][H]
BFh		[ALT][1][9][1]

Character Code (hex)	Character Name	Key Sequence
C0h		[ALT][1][9][2]
C1h		[ALT][1][9][3]
C2h		[ALT][1][9][4]
C3h		[ALT][1][9][5]
C4h		[ALT][1][9][6]
C5h		[ALT][1][9][7]
C6h	a tilde	[CHAR][I][a]
C7h	A tilde	[CHAR][SHIFT][I][A]
C8h		[ALT][2][0][0]
C9h		[ALT][2][0][1]
CAh		[ALT][2][0][2]
CBh		[ALT][2][0][3]
CCh		[ALT][2][0][4]
CDh		[ALT][2][0][5]
CEh		[ALT][2][0][6]
CFh	general currency sign	[CHAR][G]
D0h	lower case eth	[CHAR][d]
D1h	upper case eth	[CHAR][SHIFT][D]
D2h	E circumflex	[CHAR][SHIFT][Y][E]
D3h	E diaeresis	[CHAR][SHIFT][U][E]
D4h	E grave	[CHAR][SHIFT][T][E]
D5h	i without dot	[CHAR][N]
D6h	I acute	[CHAR][SHIFT][R][I]
D7h	I circumflex	[CHAR][SHIFT][Y][I]
D8h	I diaeresis	[CHAR][SHIFT][U][I]
D9h		[ALT][2][1][7]
DAh		[ALT][2][1][8]
DBh		[ALT][2][1][9]
DCh		[ALT][2][2][0]
DDh	broken vertical bar	[CHAR][C]
DEh	I grave	[CHAR][SHIFT][T][I]
DFh		[ALT][2][2][3]

Character Code (hex)	Character Name	Key Sequence
E0h	O acute	[CHAR][SHIFT][R][O]
E1h	sharp s	[CHAR][S]
E2h	O circumflex	[CHAR][SHIFT][Y][O]
E3h	O grave	[CHAR][SHIFT][T][O]
E4h	o tilde	[CHAR][T][o]
E5h	O tilde	[CHAR][SHIFT][T][O]
E6h	mu	[CHAR][M]
E7h	Lower case thorn	[CHAR][P]
E8h	Upper case thorn	[CHAR][SHIFT][P]
E9h	U acute	[CHAR][SHIFT][R][U]
EAh	U circumflex	[CHAR][SHIFT][Y][U]
EBh	U grave	[CHAR][SHIFT][T][U]
ECh	y acute	[CHAR][R][Y]
EDh	Y acute	[CHAR][SHIFT][R][Y]
EEh	ordinal indicator	[CHAR][MEMO]
EFh	acute	[CHAR][COMM]
F0h	minus sign	[CHAR][-]
F1h	plus/minus	[CHAR][+]
F2h	subscript =	[CHAR][=]
F3h	3/4	[CHAR][9]
F4h	Paragraph sign	[CHAR][Z]
F5h	Section sign	[CHAR][V]
F6h	divide sign	[CHAR][/]
F7h		[CHAR][,]
F8h	degree sign	[CHAR][X]
F9h	umlaut	[CHAR][APPT]
FAh	middle dot	[CHAR][.]
FBh	superscript 1	[CHAR][SHIFT][1]
FCh	superscript 3	[CHAR][SHIFT][3]
FDh	superscript 2	[CHAR][SHIFT][2]
FEh	block	[CHAR][B]
FFh		[ALT][2][5][5]

Int 0Ah

Int 0Ah - Miscellaneous Hardware Interrupt

The Miscellaneous Interrupt services the following hardware interrupts:

- **Timer 1 interrupt**
- **Display Cursor Update Request**

Timer 1 interrupt. The timer1 interrupt is used to implement keyboard peeks if a key is down. It repeatedly causes keyboard scans to determine which key is pressed. When the key board changes state it issues an INT 09 indicating a new key is down or up.

Display Cursor Update Request Interrupt. This interrupt is normally disabled. The display cursor update request is detected by polling in the Timer 0 interrupt routine (INT 08h).

Int 0Bh - Keyboard Hardware Interrupt

Int 0Bh is invoked when a key is pressed. It performs a software scan and debounce delay of the keyboard to detect any newly pressed or released keys. If a key is newly pressed, it sets a bit which causes timer1 interrupts to call the keyscan code, thereby implementing periodic keyboard peeks. Whenever a key is newly pressed or released, it computes its one byte scan code and places it in an I/O port (060h). Then it invokes the Int 09h service routine, which is the IBM compatible keyboard hardware interrupt.

Int 0Fh

Int 0Fh - Real-Time Clock Interrupt

The real-time clock hardware interrupt is intended to implement a software real time clock. The interrupt can be set to wake up the CPU at time intervals of 1 second up to 9.1 hours.

Int 10h - Video Services Interrupt

The video services control the display. These services provide a number of standard functions for setting the mode of the display, writing characters and dots to the display, and controlling character attributes. Int 10h supports two modes:

- Mode 07: 80 x 25 monochrome alphanumeric. Mode 7 is compatible with the industry-standard Monochrome Display Adapter (MDA).
- Mode 20h: 240 x 128 Graphics mode. Mode 20h is a unique graphics mode not compatible with any IBM mode.

To set the desired mode, use the Set Mode function (Int 10h, AH = 00h). The power-on default is Mode 7.

Alphanumeric Mode 7 The physical size of Jaguar's display is smaller than a standard display (40 x 16 vs 80 x 25). However, the Jaguar display RAM is the same size as the industry standard MDA (4K bytes). There is provision to window around in the display RAM, so the user can see the contents of all 4K of display memory.

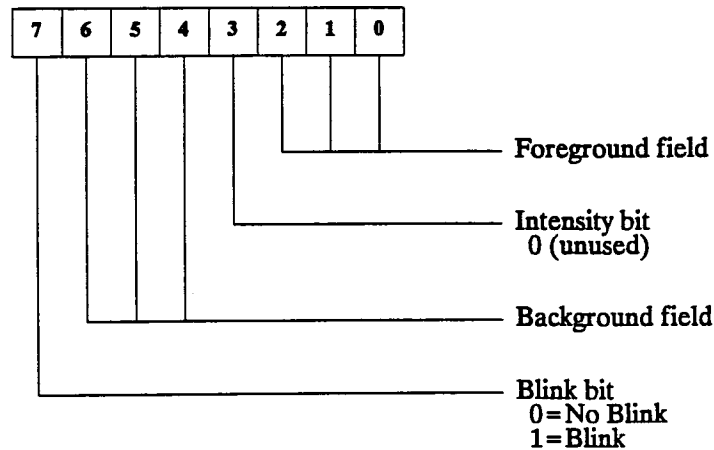
Windowing is done by the BIOS to keep the cursor always in view. When the cursor is moved, the hardware causes an INT 0Ah to occur. The Interrupt 0Ah service routine sets bit 5 of the CurFlag byte in the bios data area to indicate a cursor movement has recently occurred. The Timer 0 interrupt service routine always shifts CurFlag left one bit position. This effectively debounces the CurFlag byte. If the shift results in CF set and CurFlag = 0, it indicates the cursor has been moving, but is now quiet. Timer 0 recognizes this state and moves the display window so it contains the cursor.

When configured for mode 7, the display memory is organized into cells containing 2 bytes each. Byte 0 of each cell contains the 8 bit character code and byte 1 defines the display attributes for that character. Character cells are arranged in rows of 80 cells each, and there are 25 rows of cells. There is only one 80 x 25 page of text. The base address of display memory is B0000h.

	Cell 0	Cell 1	...	Cell 79	
0B0000h			...		Row 0
0B00A0h			...		Row 1
	.	.	.		
	.	.	.		
	.	.	.		
0B00F00h			...		Row 24

The attribute byte has the following bit definitions:

Int 10h



Only four foreground and background field combinations are useful:

Background Field (Hex)	Foreground Field(Hex)	Definition
0	0	White on white (does not display)
0	1	Underlined characters
0	7	Normal video (black characters on a white background)
7	0	Inverse video (white characters on a black background)

Graphics Mode 20h The graphics mode (mode 20h) is used to display either graphics or alpha information. When the display is in graphics mode, each bit in display memory is mapped to a display pixel. The base address of display memory is at 0B0000h.

	0	1	1Dh	
0B0000h	7 6 5 4 3 2 1 0		...	Row 0
0B001Eh			...	Row 1
	.			
	.			
	.			
0B00EE2h			...	Row 127

Within each byte of display memory, bit 7 maps to the left most pixel and bit 0 maps to the right most pixel. The upper left pixel of the display is bit 7 of 0B0000h.

Video Services The Video Services are described below. Specify the desired function code in register AH (or AX), with additional parameters passed in other registers as indicated in the table.

- AH = 00h Set Mode

This function sets the display mode. The new mode is determined by the value passed in the AL register.

Input: AH = 00h.
 AL = 07h = 80 char x 25 char monochrome adapter mode.
 20h = 240 pixel x 128 pixel graphics mode.

Output: None. The mode is changed in the BIOS and hardware.
 The screen is cleared as a side effect of changing the mode.

Error conditions: None.
 Registers modified: AX.

- AH = 01h Set Cursor Size

This function sets the size of the cursor displayed in the alphanumeric display modes. Each character cell in the alphanumeric display modes is eight scan lines high. The cursor size is defined by specifying the starting scan line within the character cell. The scan lines are numbered from 0 (top of cell) to 7 (bottom). The ending scan line is always fixed at 7. The size of the cursor is defined by passing the starting scan line in register CH. The default value is CH=7. If bit 5 of CH is set to "1", the cursor will be suppressed. In graphics mode bit 5 is automatically set, thus no cursor is displayed.

NOTE: This is slightly different from 100% IBM compatible displays where the both the starting and ending scan lines for the cursor can be defined. In these displays, the BIOS accepts the starting scan line in AH and ending scan line in AL.

Input: AH = 01h.
 CH = Starting scan line.
 Output: None.
 Error conditions: None.
 Registers modified: AX.

- AH = 02h Set Cursor Position

This function sets the cursor position to the specified row and column address on the specified page. For Jaguar, the display page should be set to 0 in either graphics or text mode. This function applies to both text and graphics modes. In graphics mode, the cursor is invisible, but is used to define a position on the screen.

Input: AH = 02h.
 BH = Display page number.
 DH = Row address of cursor (0 - 24 for alpha, 0 - 15 for graphics).
 DL = Column address of cursor (0 - 79 for alpha, 0 - 39 for graphics).
 Output: None.
 Error conditions: None.
 Registers modified: AX.

- AH = 03h Read Cursor Position

This function returns the current address and size of the cursor on the specified page. For Jaguar,

Int 10h

the display page should be set to 0 for either graphics or text mode.

Input: AH = 03h.
BH = Display page number.

Output: CH = Starting scan line of the cursor.
CL = Ending scan line of the cursor.
DH = Row address of the cursor (0 - 24 for alpha, 0 - 15 for graphics).
DL = Column address of the cursor (0 - 79 for alpha, 0 - 39 for graphics).

Error conditions: None.

Registers modified: AX, CX, and DX.

- AH = 04h Read Light Pen Position

This function returns the current state and position of a light pen. Since Jaguar does not support a light pen, the function always returns AH = 0 to indicate the light pen is not activated.

Input: AH = 04h.

Output: AH = Light pen state (0 = not activated).

Error conditions: None.

Registers modified: AX.

- AH = 05h Set Active Display Page

This function sets the active display page. In Jaguar, the only allowed display page is 0.

Input: AH = 05h.
AL = Page number: Must be 0.

Output: None.

Error conditions: None.

Registers modified: AX.

- AH = 06h Scroll Rectangle Up

This function scrolls the contents of a window up a specified number of lines. The window is defined by the row and column addresses specified in the CX and DX registers. The number of lines to be scrolled is passed in register AL. If AL is set to "0", the entire window is blanked.

NOTE: This video service function operates only display RAM. It is performed regardless of the position of the video cursor and has nothing to do with the windowing done in the background to keep the cursor always in view in the 40 x 16 LCD.

Input: AH = 06h.
AL = Number of lines to scroll (0 = blanks entire scroll area).
BH = Attribute of blanked lines (alpha mode) or
Fill character for blanked lines (graphics mode)
CH = Row address of character in upper left corner of window.
CL = Column address of character in upper left corner of window.
DH = Row address of character in lower right corner of window.
DL = Column address of character in lower right corner of window.

Output: None.

Error conditions: None.

Registers modified: AX.

- AH = 07h Scroll Rectangle Down

This function scrolls the contents of a window down a specified number of lines. The window is defined by the row and column addresses specified in the CX and DX registers. The number of lines to be scrolled is passed in register AL. If AL is set to "0", the entire window is blanked.

NOTE: This video service function operates only display RAM. It is performed regardless of the position of the video cursor and has nothing to do with the windowing done in the background to keep the cursor always in view in the 40 x 16 LCD.

Input: AH = 07h.
 AL = Number of lines to scroll (0 = blanks entire scroll area).
 BH = Attribute of blanked lines (alpha mode) or
 Fill character for blanked lines (graphics mode)
 CH = Row address of character in upper left corner of window.
 CL = Column address of character in upper left corner of window.
 DH = Row address of character in lower right corner of window.
 DL = Column address of character in lower right corner of window.

Output: None.
Error conditions: None.
Registers modified: AX.

• AH = 08h Read Character and Attribute at Cursor Position

If the display is in alphanumeric mode, this function returns the character and attribute bytes at the current cursor location. If the display is in graphics mode, the BIOS attempts to match the bit pattern at the cursor position with a character pattern from the graphics-character font resident in the BIOS ROM. If it finds a match, the character is returned in AL. If no match is found, AL is set to zero. No attribute is returned when in graphics mode. The display page must be 0 for alpha mode. Display page is a don't care for graphics mode.

Input: AH = 08h.
 BH = Page number (must be 0 alpha mode; don't care for graphics mode)

Output: AH = Attribute byte (valid for alphanumeric modes).
 AL = Character.

Error conditions: AL = 00h if in graphics mode and no match was found for the current cursor position.

Registers modified: AX.

• AH = 09h Write Character and Attribute at Cursor Position

If the display is in alphanumeric mode, this function writes character and attribute bytes at the current cursor location. The value in BL determines the character attributes.

If the display is in graphics mode, no page number is required and the value in BH is ignored. The attribute byte in BL has different meaning in graphics mode. If bit 7 of BL is set, an exclusive OR (XOR) of the pixel data is performed with existing display data. If bit 7 is clear, the pixel data overwrites the existing display data.

For both the alphanumeric and graphics modes, more than one copy of a single character (with attribute) can be written to the display. Specify the number of copies desired in register CX. In alphanumeric mode this function will cause line wrap and screen wrap to occur if too many characters are specified. In graphics mode no wrap-around will occur.

Int 10h

Input: AH = 09h.
AL = Character to write.
BH = Page number; must be 0 in alpha mode; not used in graphics mode
BL = Attribute byte if in alphanumeric mode
(Bit 7 set means XOR pixel data if in graphics mode)
CX = Number of characters to write.
Output: None.
Error conditions: None.
Registers modified: AX.

- AH = 0Ah Write Character at Cursor Position

This function writes a character to the current cursor location, but leaves the attribute byte at that location unchanged. The function is otherwise identical to function 09h (Write Character and Attribute at Cursor Position).

Input: AH = 0Ah.
AL = Character to write.
BH = Page number (must be 0 for alpha mode; not used in graphics mode).
CX = Number of characters to write.
Output: None.
Error conditions: None.
Registers modified: AX.

- AH = 0Bh Set Color Palette

Since Jaguar's LCD does not support color, this function has no effect.

- AH = 0Ch Write Pixel

This function writes a pixel on the screen. If bit 7 of register AL is set, an exclusive OR (XOR) is performed on the current pixel value in display memory and the bit value given in bit 0 of register AL. If bit 7 is clear, bit 0 of AL is written as the new pixel value.

Input: AH = 0Ch.
AL = Pixel value:
Bit 7:
If "1", XOR current value with bit 0.
If "0", replace current value with value given by bit 0.
Bit 0: Pixel value
CX = Horizontal pixel address.
DX = Vertical pixel address.
Output: None.
Error conditions: None.
Registers modified: AX.

- AH = 0Dh Read Pixel

This function returns the value of the specified pixel.

Input: AH = 0Dh.
 CX = Horizontal pixel address.
 DX = Vertical pixel address.
 Output: AL = value of pixel (0 or 1)
 Error conditions: None.
 Registers modified: AX.

- AH = 0Eh Write Teletype Character

This function writes a character to the display memory, then advances the cursor one location. At the end of a line, the cursor will wrap to the start of the next line. At the end of the screen, the BIOS will scroll the screen up one line, blank a line at the bottom of the screen, and place the cursor at the start of that line. Four characters have special interpretations: Line Feed (0Ah), Carriage Return (0Dh), Backspace (08h), and Bell (07h). The BIOS performs the appropriate actions when it senses these characters. When in alphanumeric mode, the current screen attributes are unchanged.

Input: AH = 0Eh.
 AL = Character.
 Output: None.
 Error conditions: None.
 Registers modified: AX.

- AH = 0Fh Get Video State and Mode

This function returns the current state of the display, including the current mode, number of characters per line, and current display page. Refer to the Set Mode function (AH = 00h) for a description of the modes.

Input: AH = 0Fh.
 Output: AH = Number of characters per line.
 AL = Current mode.
 BH = Current display page.
 Error conditions: None.
 Registers modified: AX.

- AH = 10h Reserved
- AH = 12h Reserved
- AX = 1300h Write String, Global Attribute

This function writes a string with one global attribute. After the write is complete, the cursor is restored to its original position on the screen. This function uses the Write Teletype Character function (Int 10h, AH = 0Eh) to place the characters in display memory.

Int 10h

Input: AX = 1300h.
BH = Display page number.
BL = String attribute byte.
(Bit 7 set means XOR pixel data if in graphics mode)
CX = Length of string.
DH = Row address of first character.
DL = Column address of first character.
ES:BP = Pointer to start of string.
Format of string is: Char, Char, ...

Output: None (display memory is updated).
Error conditions: None.
Registers modified: AX.

- AX = 1301h Write String, Global Attribute, Move Cursor

This function operates in the same way as function AX = 1300h, except that once the operation is complete, it moves the cursor to the character cell following the last character written.

Input: AX = 1301h.
BH = Display page number.
BL = String attribute byte.
(Bit 7 set means XOR pixel data if in graphics mode)
CX = Length of string.
DH = Row address of first character.
DL = Column address of first character.
ES:BP = Pointer to start of string.
Format of string is: Char, Char, ...

Output: None (display memory is updated).
Error conditions: None.
Registers modified: AX.

- AX = 1302h Write String, Individual Attributes

This function operates like function AX = 1300h, except that it writes each character in a string with its own attribute. After the write is complete, the cursor is restored to its original position on the screen.

Input: AX = 1302h.
BH = Display page number.
CX = Length of string.
DH = Row address of first character.
DL = Column address of first character.
ES:BP = Pointer to start of string.
Format of string is: Char, Attr, Char, Attr ...

Output: None (display memory is updated).
Error conditions: None.
Registers modified: AX.

- AX = 1303h Write String, Individual Attributes, Move Cursor

This function operates in the same way as function AX = 1302h, except that once the operation is complete, it moves the cursor to the character cell following the last character written.

Input:	AX = 1303h. BH = Display page number. CX = Length of string. DH = Row address of first character. DL = Column address of first character. ES:BP = Pointer to start of string. Format of string is: Char, Attr, Char, Attr ...
Output:	None (display memory is updated).
Error conditions:	None.
Registers modified:	AX.

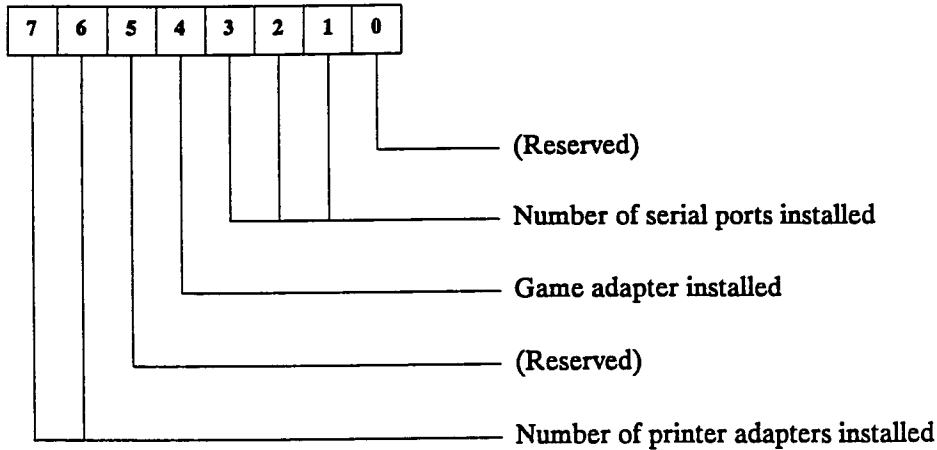
Int 11h

Int 11h - Equipment Check Interrupt

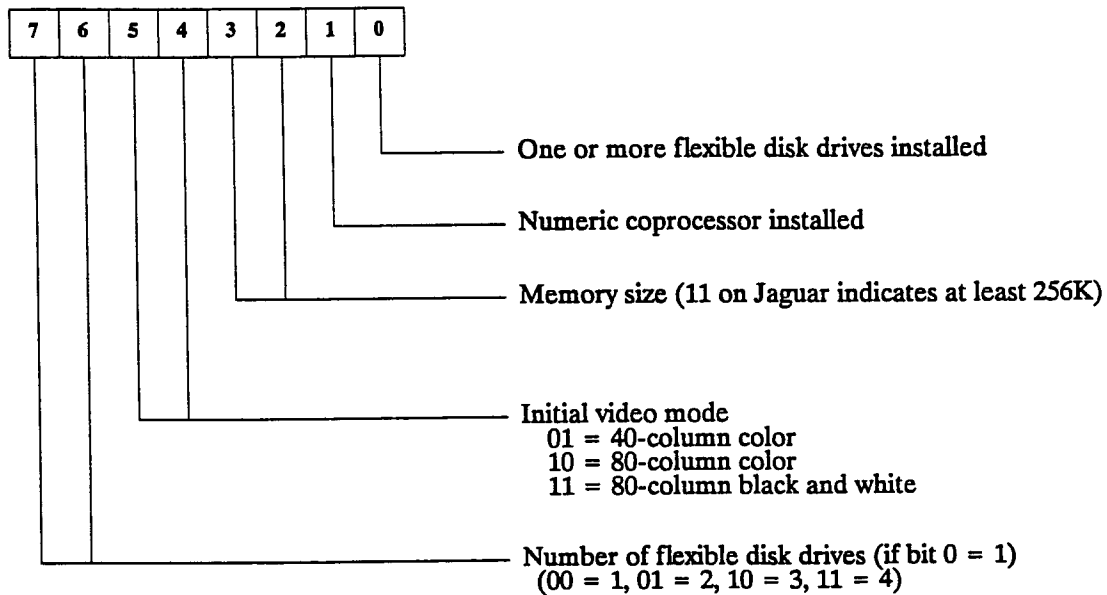
The BIOS returns a copy of its internal equipment list in the ax register. The list is compiled in the word at 40:10.

Input: AH = 11h.

Output: AH :



AL:



Error conditions: None.
Registers modified: AX

Int 12h - Memory Size Interrupt

The Int 12h service, when executed, returns the number of 1K byte blocks of system RAM in register AX. Note that the number returned is the amount of user RAM found in the system during the power-on and initialization process. It does not include any expanded RAM that may be present.

The current Jaguar design has 512K bytes of memory built in as standard equipment. This can be increased by adding a plug-in RAM card. No method is currently provided to disable a portion of the 512K RAM. The number returned by Int 12h is approximately 512K minus the number of Kbytes in the RAM portion of the drive C: Edisk.

Note An application can use this service to determine the total amount of user RAM in the system. However, this service does not indicate how much RAM is available ("free") for running applications.

Int 13h

Int 13h - Disk Services Interrupt

The Int 13h services provide low-level support of the Built-In and Plug-In RAM disks. These services directly access the memory addresses that comprise the RAM Disks, and are responsible for bank selection of the Plug-In RAM disk. They are also responsible for maintaining a table that contains a checksum of each sector in the disks.

The Int 13h services provide the ability to read, write, and verify sectors. The services also perform the formatting of tracks on a disk, and provide a number of functions to obtain status information about the disks. Many of the functions are not relevant to a RAM disk, but are included to maintain compatibility with the IBM XT.

The disk parameter tables are provided for compatibility reasons. There are two tables, one for plug-in disks (drives a: and b:) and one for the built-in ROM-RAM disk. The table for drives a: and b: is pointed to by the Int 1Eh vector. The table for drive c: is pointed to by the Int 41h vector.

The default tables, provided in the BIOS ROM, are described below.

Plug-In Disk Parameter Table (Drives A: and B:)

Offset	Bytes	Definition
00h	1	FDC Specify command: step rate and head unload time.
01h	1	FDC Specify command: head load time and DMA mode.
03h	1	Bytes per sector: 0 = 128, 1 = 256, 2 = 512, 3 = 1024.
04h	1	Last sector number on track.
05h	1	read/write gap length between sectors.
06h	1	Data length for read/write operations.
07h	1	Format gap length between sectors.
08h	1	Format filler byte for sectors.
09h	1	Head-settle time after seek command, in milliseconds.
0Ah	1	Motor-start time in 1/8-second units.

Built-In Disk Parameter Table (Drive C:)

Offset	Bytes	Definition
00h	2	Number of Cylinders
02h	1	Number of Heads
03h	2	Reserved(0)
05h	2	Starting write pre-comp cylinder (0)
07h	1	Max ECC burst len (0)
08h	1	Control Byte (0C0h)
09h	3	Reserved (0,0,0)
0Ch	2	Landing zone (0)
0Eh	1	Sectors/Track (10h)
0Fh	1	Reserved (0)

The Disk Services interrupt (Int 13h) functions are described in the table below. Specify the desired function code in register AH, with additional parameters passed in other registers as indicated in the table.

- **AH = 00h Reset Disk System**

This function does nothing in Jaguar. It is provided for compatibility reasons.

Input: AH = 00h.

Output: AH = Return disk drive status
See Function 01 below.

Error conditions: As indicated in AH

Registers modified: AX, status

- **AH = 01h Read Status of Last Operation**

This function returns the error status code that resulted from the last disk operation (00h is returned if the last operation was successful).

The function returns with carry clear, even if a non-zero value is returned (indicating an error).

The return codes are defined in the table below.

Disk Return Codes

Value	Error
04h	Requested sector could not be found.
10h	Checksum error encountered on disk read.
80h	Invalid Drive

Input: AH = 01h.

Output: AL = Return status of last disk operation.
AH = 0

Registers modified: AX, status

- **AH = 02h Read Disk Sectors**

Based on the supplied parameters, one or more sectors are transferred from the disk into a data buffer in system RAM. Application programs must ensure that the data area provided is large enough to contain the requested data.

Int 13h

Input: AH = 02h.
AL = Number of sectors(1-16)
CH = bits 0-7 of track number (0-max track)
CL bits 6-7: bits 8-9 of track number
CL bits 0-5: Sector number(1-16)
DH = Head number (always 0)
DL = Drive number (1 for drive c:,0 for drive a:,2 for drive d:,etc)

ES:BX = Pointer to buffer in which to put the data read from the disk.

Output: AH = Return status (refer function 01)
AL = Number of sectors read.

Error conditions: Carry flag is not set if the operation was successful.
Carry flag is set on an error condition.

Registers modified: AX, Status.

- AH = 03h Write Disk Sectors

This function is very similar to the Read Disk Sectors function, except that it writes data from the data buffer to a disk. See the description of Read Disk Sectors (above) for more details.

Input: AH = 03h.
AL = Number of sectors (1-16)
CH = bits 0-7 of track number (0-max track)
CL bits 6-7: bits 8-9 of track number
CL bits 0-5: Sector number(1-16)
DH = Head number (0 for drive c:)
DL = Drive unit number.
ES:BX = Pointer to buffer from which to write data to the disk.

Output: AH = Return status (refer to function 01)
AL = Number of sectors written.

Error conditions: Carry flag is not set if the operation was successful. Carry flag is set on an error condition.

Registers modified: AX, Status.

- AH = 04h Verify Disk Sectors

This function performs a read function without transferring any data. This function insures that the track, head, and sector can be located on the disk, and that the data in the sector can be read. The description of the Read Disk Sectors function is applicable, except that no data is transferred. The number of sectors verified is returned in AL.

Input: AH = 04h.
 AL = Number of sectors.
 CH = bits 0-7 of track number (0-max track)
 CL bits 6-7: bits 8-9 of track number
 CL bits 0-5: Sector number(1-16)
 DH = Head number.
 DL = Drive unit number.

Output: AH = Return status (refer to function 01)
 AL = Number of sectors verified.

Error conditions: Carry flag is not set if the operation was successful.
 Carry flag is set on an error condition.

Registers modified: AX, Status.

- AH = 05h Format a Track

This function is a NOP on Jaguar.

Input: AH = 05h.

Output: AH = Return status (Always 0)

Error conditions: None

Registers modified: AH, Status.

- AH = 08h Get Drive Parameters

This function returns a set of disk drive parameters for the drive unit number specified in register DL. These parameters reflect the recommended formatting parameters for the drive.

The register pair ES:DI, if valid, points to a disk parameter table that contains the values recommended for use in formatting the disk drive.

If this function is called with a hard disk drive unit number specified in DL (That is, the drive unit number is greater or equal to 80h), the function sets the carry flag and sets AH to 01h, indicating a bad device number.

Int 13h

Input: AH = 08h.
DL = Drive unit number.

Output: If a drive exists for the drive unit number:
AX = 00h.
BH = 00h.
BL = Drive type code: Always 0 for Jaguar
CH = bits 0-7 of max track number
CL bits 6-7: bits 8-9 of max track number
CL bits 0-5: Sectors per track (10h)
DH Maximum head number (0)
DL = Number of disk drives in the system (01)
ES:DI = Pointer to disk parameter table.

Error conditions: If specified drive does not exist returns Carry Set and AH=80h

Registers modified: AX, BX, CX, DX, DI, ES.

- AH = 15h Get Disk Drive Type

This function returns the disk drive type code for the specified device.

Input: AH = 15h.
DL = Drive unit number.

Output: AH = Disk drive code:
00h = No drive present.
01h = Plug-In disk present, no disk change line available.
02h = Plug-In disk present, disk change line is available.
03h = built-in disk
CX:DX = number of fixed disk sectors

Error conditions: Carry flag set on any error.

Registers modified: AH, Status.

- AH = 16h Disk Change Status

This function reports the status of the Disk Change line of the specified disk drive.

Input: AH = 16h.
DL = Drive unit number (0 - 1).

Output: AH = 00h if Disk not changed.
AH = 01h and carry flag is set if value in DL is invalid.
AH = 06h and carry flag is set if Disk changed.

Error conditions: As given in AH (refer to function 01)

Registers modified: AH, Status.

- **AX = 0FF00h Modify RAM-ROM DISK (Drive C:) RAM Partition**

This function changes the size of the drive c: ram partition to BX kbytes.

Input: AH = 0ffh.
AL = 00h
DL = 1h
BX = #kbytes to allocate to RAM portion of Disk (default = 384 if BX = 0)

Output: BX = -1h if insufficient memory
BX = 1h if RAM disk too full for shrink
BX = 0h if successful

Error conditions: As given in BX

Registers modified: AX, BX, Status

- **AX = 0FF01h Initialize RAM-ROM DISK (Drive C:)**

This function initializes drive c:

Int 13h

Input: AH = 0ffh.
 AL = 01h
 DL = 1h
 BX = #kbytes to allocate to RAM Disk (default=384 if BX=0)
 CL = # root directory sectors (16 dir entries/sector)
 (0 = use default of 4 sectors)
 CH = init data sectors flag
 (1= clear to 0's, 0= leave alone)

Output: BX = -1h if insufficient memory
 BX = remaining user memory in kbytes if successful

Error conditions: As given in BX

Registers modified: AX, BX, Status.

- **AX = FD00 Find logical page and offset for "filename"**

Input: AX = FD00
 ES:BX = address of filename. The filename is an array
 of 11 bytes, with the primary portion of the
 filename in the first 8 bytes (blank-filled)
 and the filename extension in the last 3 bytes
 (blank-filled).
 DL = BIOS drive # (0=A, 2=D, 3=E, 4=F) (drive C is
 not supported).

Output: if NC: BX = 16K logical page, CX = offset in that 16K page
 if CY: filename was not found

- **AX = FD01 Set checksums/flags for cards in Ports 0 and 1**

Input: DL = 0
Output: nothing

- **AX = FD02 Check checksums/flags for cards in Ports 0 and 1**

Input: DL = 0
Output: AL Bit 0=0 if no change for PORT 0
 Bit 1=0 if no change for PORT 1
 Bit 2-7=0

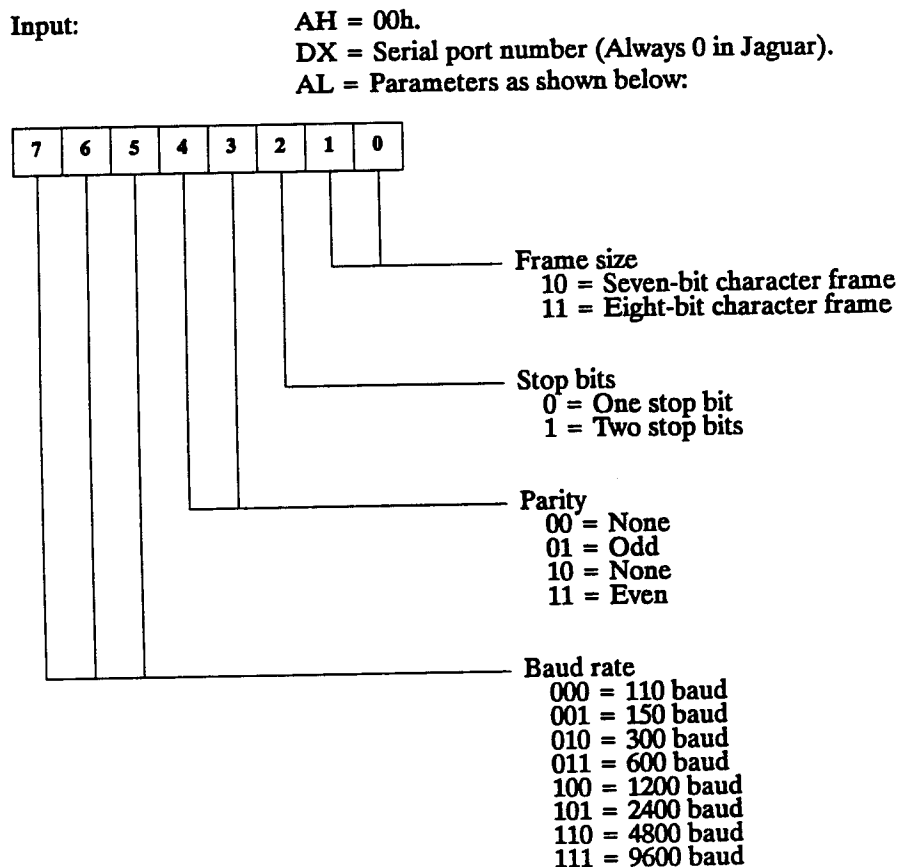
Int 14h - Serial Port Services Interrupt

This service provides IBM-XT compatible support for the serial port. BIOS serial port operations are performed only in a polled mode. While Jaguar hardware supports serial port interrupts, the BIOS provides no support for interrupt-driven serial operations. However, an application program can provide interrupt-driven support by writing directly to the serial port hardware.

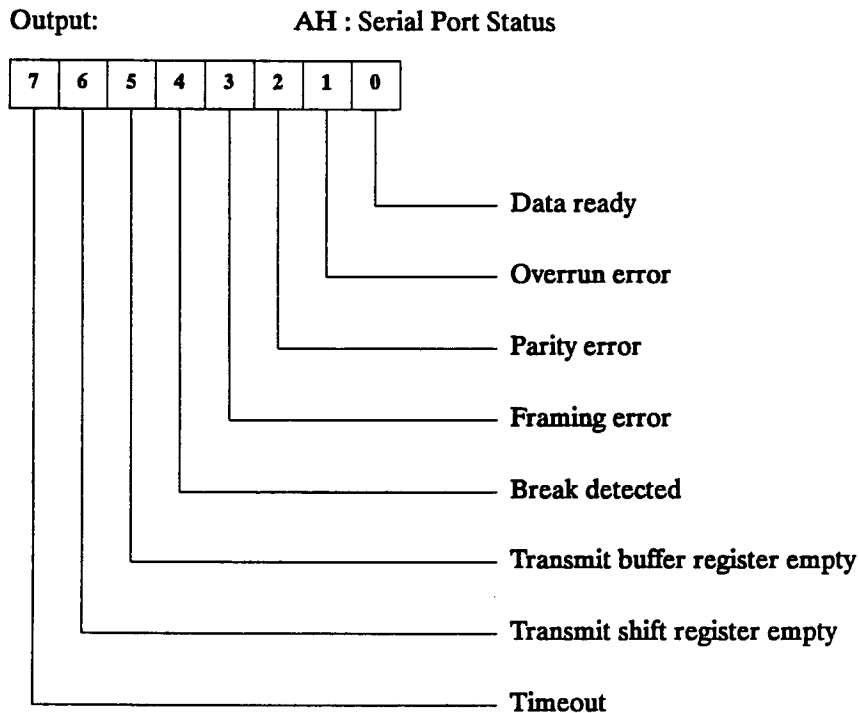
The Serial Port Services interrupt (Int 14h) functions are described in the table below. Specify the desired function code in register AH, with additional parameters passed in other registers as indicated in the table.

- **AH = 00h** **Initialize Serial Port Parameters**

This service sets the baud rate, parity, number of stop bits, and character frame size for the specified serial port. Register AL is used to pass the serial port initialization parameters for the Int 14h, AH = 00h function, as shown in figure below:



Int 14h



Error conditions: Timeout error (bit 7) is set in AH if DX is not in range.

Registers modified: AX.

- AH = 01h Transmit One Character

This function transmits one character through the serial port. the function waits until the UART transmit buffer is empty, then transmits the character by loading it into the buffer.

When transmitting a character, the BIOS service routine loops until the serial port indicates that it can transmit the character. If the port does not indicate that it is ready within a timeout period, the function returns a timeout error.

Input:
AH = 01h.
AL = Character to transmit.
DX = Serial port number (Always 0 in Jaguar)

Output: AH = Status as described for 'Initialize Serial Port Function 00'

Error conditions: Timeout is indicated by setting bit 7 in AH.

Registers modified: AH.

- AH = 02h Receive One Character

This function returns the character received by the serial port. The function waits until the serial port reports that a character has been received, then reads the serial port status register and reports

any error conditions that may have occurred. The character is returned in AL. NOTE: This function changes the serial port interrupt vector to point to a dummy interrupt service routine (IRET) at 0f000h:0ff53h.

Input: AH = 02h.
DX = Serial port number (Always 0 in Jaguar).

Output: AH = Status as described for 'Initialize Serial Port Function 00' except that only bits 7, 4, 3, 2, and 1 are reported.
AL = Character received (valid only if AH = 0).

Error conditions: If AH is non-zero, an error has occurred and the character in AL should be discarded. If bit 7 is set in AH, a timeout occurred, and the other bits in AH may not be valid.

Registers modified: AX.

• **AH = 03h** **Get Serial Port Status**

This function reports the status of the serial port and the modem-control lines connected to the serial adapter.

Input: AH = 03h.
DX = Serial port number (Always 0 for Jaguar)

Output: AH = Status as described for 'Initialize Serial Port Function 00'

Error conditions: Timeout error set in AH if DX is not in range.

Registers modified: AX.

Int 15h

Int 15h - System Functions Interrupt

The Int 15h system functions provide a number of general services not related to a particular hardware function in Jaguar.

The Int 15h system functions are described in the table below. Specify the desired function code in register AH, with additional parameters passed in other registers as indicated in the table.

Function 4Fh is found universally in all late model PCs, and allows applications to translate keys easily. Function C0 provides a pointer to a system description table which describes the machine. This table and function is also new, and present across the IBM PC line, including the PS/2 machines.

Note: If a call is made to a function code not listed in the table, Int 15h will return with AH = 86h and the carry flag set, indicating an error.

- AH = 00h, 01h, 02h, 03h, 04h Cassette Functions.

These functions are mentioned for compatibility reasons. Jaguar does not support the cassette drive. Register AH is set to 86h the carry flag is set to indicate an error.

Input: AH = 00h, 01h, 02h or 03h

Output: AH=86h and CF set to indicate an error

- AH = 41h WAIT UNTIL EVENT FUNCTION

This function tests a specified byte in either System RAM or in I/O space. If the test is true, the function returns. If the test is false, the function goes into light sleep. A hardware interrupt or NonMaskable Interrupt will then wake up the processor. Control is returned to the calling routine if the specified condition becomes true.

There are two timeouts involved with this function:

- Return timeout:

The calling routine can specify a return timeout value in BL. The return timeout can vary between 1 and 255 55msec intervals. If BL=0, return timeout is disabled.

If return timeout is enabled and the interval is exceeded, the function will return to the caller with carry set. The timeout value is stored in the BIOS data segment at 40:7b. The timer0 interrupt decrements this timeout value.

- Display timeout:

This timeout is specified by invoking INT 15h service 46h. If the display timeout period is exceeded, Jaguar will go to deep sleep. Upon subsequent wakeup, it will return to the caller with carry set, just as if a return timeout had occurred.

Input: AH=41h
AL=Type of test to perform:

- 00h - Wait for any interrupt (just goes to light sleep until next interrupt)
- 01h - Compare RAM byte with BH, return when equal to BH
- 02h - Compare RAM byte with BH, return when not equal to BH
- 03h - Test RAM byte with BH mask, return when not equal to 0
- 04h - Test RAM byte with BH mask, return when equal to 0
- 10h - Wait for any interrupt (just goes to light sleep until next interrupt)
- 11h - Compare I/O byte with BH, return when not equal
- 12h - Compare I/O byte with BH, return when equal
- 13h - Test I/O byte with BH, return when not equal to 0
- 14h - Test I/O byte with BH, return when equal to 0

BH= Value or mask
BL=Timeout value in 55ms intervals. 0=timeout disabled
Plus Either:
ES:DI=Pointer to byte in memory to test (AL=1-4)
DX=I/O port (AL=11h-14h)

Output: Carry Clear if test is true
Carry Set if timeout occurred or invalid test type in AL

Registers modified: AX,Status

• AH = 42h Deep Sleep

This function implements deep sleep. Jaguar remains in deep sleep until one of the following events:

- ON key is pressed.
- Five characters are received by the serial port within approximately 1.5 seconds of each other and there is at most one framing or parity error.
- XINT line is asserted for 50 ms.
- Real Time Clock wake up alarm occurs.

When a wake up occurs, the machine exits from this service and returns to the calling program.

Input: AH = 42h.

Registers modified: none

• AH = 45h (DE)ACTIVATE Window keys and Cursor Tracking

This function controls whether ALT-ARROW keys and ALT-SHIFT-ARROW keys will cause window movements as well as whether cursor tracking is enabled.

Int 15h

Input: AH = 45h.
AL = 0 : Enables ALT-ARROW key windowing, Enables cursor tracking
 = 1 : Disable ALT-ARROW key windowing, Disables cursor tracking
 = 2 : Enables ALT-ARROW key windowing, Disables cursor tracking
 = 3 : Disables ALT-ARROW key windowing, Enables cursor tracking
 > 3 : No Operation

Output: Windowing and cursor tracking enabled or disabled according to AL

Registers modified: none

• AH = 46h Set Display Timeout

This function enables display timeout and sets its interval. Display timeout occurs only in light sleep. See INT 15h service 41h.

Input: AH = 46h.
BX = 1-65535 : time out interval in 55 msec increments
 = 0 : Display timeout disabled

Output: None

Registers modified: none

• AH = 47h Set/Read Contrast Register

This function sets or reads the contrast register. The range of permissible values is 0 - 0fh. The highest display contrast corresponds to a value of 0fh.

Input: AH = 47h.
AL = Subfunction
 = 0 - Set Contrast value in BL
 = 1 - Read Contrast value and return it in AL
BL = 0-0fh : Contrast value (If AL=0)
 > 0fh : Contrast value set to 0fh

Output: AL = contrast value if AL was 1 at entry

Registers modified: AX

• AH = 48h Set Digital Beeper Volume

Input: AH = 48h.
AL = 0-3 : Digital Beep Volume Setting (0=quietest)
 > 3 : Volume set to loudest setting

Output: None

Registers modified: none

• **AH = 49h Set RS-232 Channel**

This function sets the RS-232 Channel to either WIRE or IR. If IR mode is selected, the UART baud rate is changed to 2400 baud.

Input: AH = 49h.
AL = 0h : WIRE (default)
AL = 1h : IR
AL > 1h : No Operation

Output: None

Registers modified: none

• **AH = 4Ah Set RS-232 Power**

This function turns power to RS232 device on or off.

Input: AH = 4Ah.
AL = 0h : RS232 power turned off
AL = 1h : RS232 power turned on
AL > 1h : No Operation

Output: None

Registers modified: none

• **AH = 4Bh Init BIOS Time**

This service loads time from the RTC and checks for a valid value. If the RTC contains an illegal value (such as seconds count more than 59), the double word at 40h:6ch is cleared to 0 and the service returns with CF=1. Otherwise, the time value from the RTC is converted to the number of 18.2 Hz ticks since midnight. This is stored in the double word at 40h:6ch and the service returns with CF=0.

Int 15h

Input: AH = 4Bh.

Output: CF=0 if successful
CF=1 if failed

Registers modified: none

- AX = 4C00h Select Keyboard

This function is a NOP in the current version of jaguar. Future versions will employ it to select the keyboard types.

Input: AX = 4C00h.
BX = keyboard code

= 0008h FINLAND
= 0010h FRANCE
= 0020h GERMANY
= 0040h ITALY
= 0200h PORTUGUL
= 0400h SPAIN
= 0800h SWEDEN
= 8000h US

Output: ah=0ffh if more than one keyboard requested or
keyboard type is not supported
ah=000h if keyboard successfully selected

Registers modified: ax

- AX = 4C01h Return Selected Keyboard Type

This function returns the selected keyboard type.

Input: AX = 4C01h.
Output: AX = selected keyboard type

= 0008h FINLAND
= 0010h FRANCE
= 0020h GERMANY
= 0040h ITALY
= 0200h PORTUGUL
= 0400h SPAIN
= 0800h SWEDEN
= 8000h US

Registers modified: AX

- AX = 4C02h Return Available Keyboard Types

This function returns the types of keyboards available. The return code in AX is the sum of keyboard codes listed in service 4c01h. For example, a return code of AX=8001h would indicate

that US and BELGIUM keyboards are supported.

Input: AX = 4C02h.
Output: AX = available keyboard type(s)

Registers modified: AX

- AX = 4C10h Select Language

This function is used to tell the BIOS whether the selected language is English or Non-English.

Input: AX = 4C10h.
 BL = Language

 = 00h English
 > 00h Non-English

Output: none
Registers modified: ax

- AX = 4C11h Return Language Type

This function returns the selected language type.

Input: AX = 4C11h.
Output: AL = selected language type

 = 00h English
 = 01h Non-English

Registers modified: AX

- AH = 4Dh Return Model Specific Information

Returns Model Specific Information

Input: AH = 4Dh.
 AL = D4h

Output: BH = 'H'
 BL = 'P'
 CH = Family type (1)
 CL = Model type (1)
 DL = Code Revision (0)

Registers modified: AX, BX, CX, DX

- AH = 4Eh Enable Light Sleep In Key Test

Enables or disables light sleep in key test. If light sleep is disabled, the display time out (shut down) is also disabled. This function should be used with caution since it is possible for the batteries to run down if light sleep in key test is disabled indefinitely.

Int 15h

Input:
AH = 4Eh.
AL = 00h - light sleep disabled, display time out disabled.
AL = 01h - light sleep enabled
AL > 01h - nop

Output: Light sleep enabled or disabled depending on value of AL at entry.

Registers modified: none

- AH = 4Fh Keyboard Translation Hook

The Keyboard Hardware interrupt (Int 09h) service routine calls this function once for each scancode received. It does so after it obtains the scancode from the keyboard shift register, but before it does any processing on the scancode. This call is provided so that an application can take over the function 4Fh "hook" to perform special processing or translation of the key. If the application service routine clears the carry before it returns to the Int 09h routine, the Int 09h routine will discard the scancode. If the carry is set, the Int 09h routine will continue with its normal processing. When the Int 9h Keyboard Interrupt service routine issues an Int 15h function 4Fh, the stack frame is set as shown below upon entry to the 4Fh function:

	Saved Flags
	Saved CS
	Saved IP from Int 9h
	Saved AX
	Saved BX
	Saved CX
	Saved DX
	Saved SI
	Saved DI
	Saved DS
	Saved ES
	Saved Flags
	Saved CS
SP →	Saved IP (from Int 15h)

Input:
AH = 4Fh.
AL = Scancode obtained from I/O port 60h.
DS = 40h (from Int 09h).

Output: none
Registers modified: none

- AH = 50h Measure Battery

This function measures the voltage of the specified battery. The return value in AX is in the range from 0-255 if the battery measurement was successfully taken. A return value of 255 represents 5.0

volts and 0 represents 0.0 volts. If an error occurred (such as battery voltage was too noisy to measure) then AX=0ffffh on exit.

Input: AH = 50h.
AL = Battery to measure
= 00 for system battery
= 01 for backup battery

Output: AX = Measured value (0-255) 0=0 volts, 255=5 volts
= 0ffffh if error occurred

Registers modified: AX

- AH = C0h Get Pointer to System Description Table

This function returns a pointer to the System Description Table (SDT) for Jaguar.

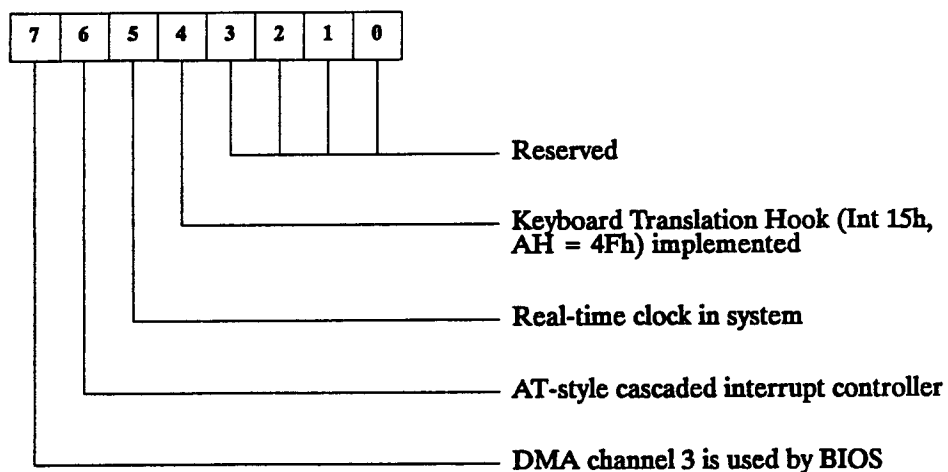
Input: AH = C0h.
Output: AH = 00h.
ES:BX = Pointer to the System Description Table.
Error conditions: None.
Registers modified: AH, BX, ES.

The SDT provides an eight byte description of the BIOS capabilities:

ES:BX →	08h	No. of bytes in SDT (LSB)
	00h	No. of bytes in SDT (MSB)
	FBh	ID Byte
	00h	Secondary ID Byte
	01h	BIOS ROM Version
	30h	Parameter Byte*
	00h	Reserved
	00h	Reserved
	00h	Reserved
	00h	Reserved

Int 15h

*The parameter byte describes certain hardware and BIOS features:



Jaguar returns 30h as its parameter byte. This indicates that it does not use DMA channel 3 and does not have a cascaded interrupt controller, but it does have a real-time clock and supports the Keyboard Translation Hook function.

Int 16h - Keyboard Services Interrupt

Int 16h is used to access character information in the keyboard buffer. Each character is stored in the buffer as a two-byte pair by Int 09h. Normally, one byte is the scancode for the character and the other byte is the equivalent ASCII character code. When key information is desired, the operating system or an application calls Int 16h, which reads the scancode and character code from the buffer and returns them. The scancode is returned in AH; the ASCII character code, in AL.

Light Sleep. Int 16h services 0, 1, 10h, 11h and 13h invoke light sleep. This is a low power mode that greatly improves battery life. In addition, light sleep code handles Shift annunciator updates and performs battery checks and display timeouts as scheduled by their respective counters.

The low power state is exited by any hardware interrupt. When an interrupt occurs, it is serviced and control is passed back to the INT 16h code. The longest interval that can occur between successive interrupts is 55 msecs, which is the period of the 18.2 Hz timer. Thus the longest interval that the machine can be in low power mode is 55 msecs.

When light sleep is exited, control is passed back to INT 16h code. The behavior of the INT 16h code depends on which service was called by the application. If service 0 or 10h was called, the INT 16h code checks to see if a keycode was placed in the key buffer. If no keycode is found, it quickly returns to light sleep and low power mode.

If service 1 or 11h was called, the behavior is to check the status of the key buffer (key code present or absent) and return to the caller.

An application that calls service 0 or 10h will save power because the machine quickly goes back to light sleep after each timer interrupt. The percentage of time that the machine is low power mode is above 98%.

An application that repeatedly tests for a key down by calling INT 16h services 1 or 11h will get some power savings, because the machine goes to low power mode on each call to INT 16h. However, the percentage of time the CPU is in low power mode will be lower than that of INT 16h services 0h or 10h. For this reason, it is recommended that services 0 or 10h be used whenever possible.

Scancode processing. The scancodes for some keys do not have an ASCII equivalent. Int 09h processes such scancodes in one of the following ways, depending on the key:

- Int 09h may discard the scancode (no data is entered into the keyboard input buffer). For example, if a scancode for a keyboard state-defining key ([Shift], [Ctrl], [Alt], [Caps Lock], [Num Lock], or [Scroll Lock]) is received, the state of the keyboard is updated, but the scancode is discarded.
- Int 09h may enter a two-byte pair "00h/XXh" into the keyboard input buffer. "XXh" may be the original scancode, or it may be a translated hexadecimal code for the key or key combination pressed. Int 16h returns "XXh" in register AH and "00h" in register AL.
- Int 09h may enter a two-byte pair "E0h/XXh" into the keyboard input buffer. This occurs if the received scancode "XXh" is prefixed with "E0h". Int 16h returns "XXh" in register AH and "E0h" in register AL.

If the [ON] key is pressed with the machine running, INT 16 invokes deep sleep. No key code is returned to the calling routine.

There are some non-compatible keys on jaguar. These include UTIL, FILER, COMM, APPT,

Int 16h

PHONE, MEMO, LOTUS 123 and HP CALC. There are special key codes for these keys, which are not part of the IBM compatible key code set. These key codes are passed on to the calling application the same as 'compatible key codes'. See the keycode table in the INT 09 section for a complete list of jaguar key codes.

The Keyboard Services interrupt (Int 16h) functions are described below. Specify the desired function code in register AH, with additional parameters passed in other registers as indicated in the table.

Note: Int 16h functions 00h and 01h discard scancode values greater than 84h. These extended scancodes can only be read with Int 16h functions 11h and 12h.

COMPATIBILITY:

The IBM KEYBXX utilities perform the following test to determine if the BIOS supports function 10h, 11h, and 12h:

```
mov    AH, 92h
Int     16h
cmp    AH, 80h
ja     NoExtendedSupport
```

Therefore Jaguar Int 16h service must make sure that AH is decremented by 12h when an invalid function code greater than 12h is passed to the service.

- AH = 00h Read Character From Keyboard Input Buffer

This function attempts to read a character from the keyboard input buffer. Each character is stored in the buffer as a two-byte pair consisting of the scancode and its ASCII equivalent. If a scancode/ASCII pair is available, it is removed from the buffer and returned in AX. Note that this function cannot return scancode values greater than 84h. Use function 10h for the extended scancodes. The function waits until a scancode/ASCII pair is present in the keyboard input buffer. Except for hardware interrupts, no other processing occurs until a key is pressed on the keyboard.

Input:	AH = 00h.
Output:	AH = Scancode.
	AL = ASCII character code, or
	00h for a special scancode.
Error conditions:	None.
Registers Modified:	AX.

- AH = 01h Report If Character Available

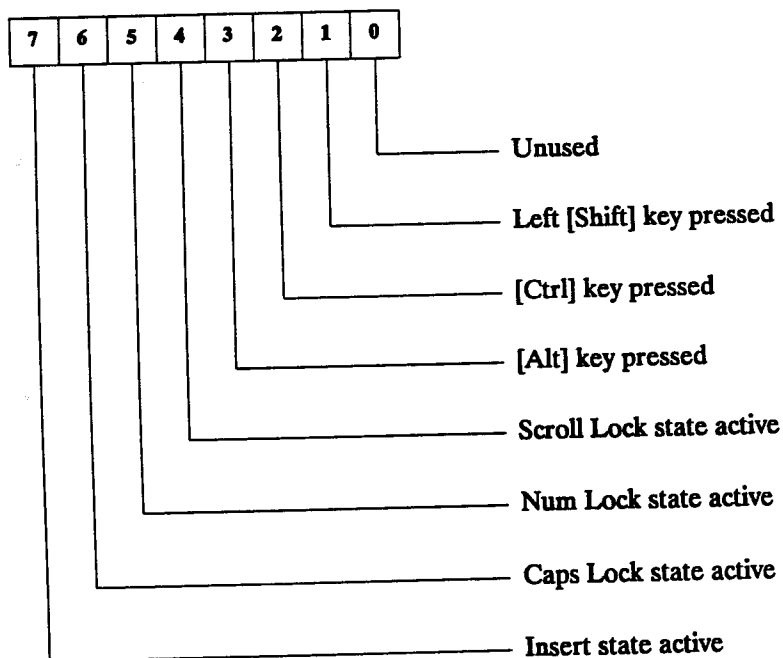
This function examines the keyboard input buffer to see if a scancode/ASCII pair is available. If a pair is available, it is returned without removing it from the buffer. The function returns immediately, regardless of whether a scancode/ASCII pair is available. This function does not recognize scancode values above 84h. Use function 11h for the extended scancodes.

Input: AH = 01h.
Output: Zero flag is clear if character is available:
AH = Scancode.
AL = ASCII character code, or
00h for a special scancode.
Zero flag is set if no characters are available:
AX is indeterminate.
Error conditions: None.
Registers modified: AX, Status.

• AH = 02h Get Shift Status

This function returns the shift status as defined below:

Input: AH = 02h.
Output: AL = Shift status bits as follows:



Error conditions: None.
Registers modified: AX.

Int 16h

- **AH = 03h** **Set Typematic Rate and Key Delay**

This function sets the typematic rate and delay between key press and when key repeat takes effect.

Input:

AH = 03h

AL = 05h Set typematic rate

BH = 00h - 03h for delays of 250ms, 500ms, 750ms, or 1s

BL = 00h - 1Fh for typematic rates of 30cps down to 2cps

BL=00h -> 30cps	BL=10h -> 7.5cps
BL=01h -> 30cps	BL=11h -> 7.5cps
BL=02h -> 30cps	BL=12h -> 6.0cps
BL=03h -> 30cps	BL=13h -> 6.0cps
BL=04h -> 30cps	BL=14h -> 5.0cps
BL=05h -> 30cps	BL=15h -> 5.0cps
BL=06h -> 30cps	BL=16h -> 4.3cps
BL=07h -> 30cps	BL=17h -> 4.3cps
BL=08h -> 15cps	BL=18h -> 3.7cps
BL=09h -> 15cps	BL=19h -> 3.3cps
BL=0ah -> 15cps	BL=1ah -> 3.0cps
BL=0bh -> 15cps	BL=1bh -> 2.7cps
BL=0ch -> 10cps	BL=1ch -> 2.5cps
BL=0dh -> 10cps	BL=1dh -> 2.3cps
BL=0eh -> 10cps	BL=1eh -> 2.1cps
BL=0fh -> 10cps	BL=1fh -> 2.0cps

Output:

None

Error conditions:

None.

Registers Modified:

- **AH = 05h** **Write Character to Keyboard Input Buffer**

This function writes the character code and scancode in CX to the keyboard input buffer. The character and scancode are placed at the end of the buffer. The function will return an error status if the buffer is full.

Input:

AH = 05h.

CH = Scancode to write to buffer.

CL = ASCII character code to write to buffer, or

00h for a special scancode, or

E0h for an extended scancode.

Output:

AL = 00h if the write succeeded.

AL = 01h if the write failed due to a full buffer.

Error conditions:

The write will fail if the buffer is full.

Registers modified:

AX.

- **AH = 10h** **Extended Read Character From Keyboard Input Buffer**

This function, like function 00h, attempts to read a character from the keyboard input buffer. However, function 10h can read both standard scancodes and the new extended scancodes. Function 10h reads characters just like function 00h, but scancode values above 84h are recognized.

If a scancode/ASCII pair is available, it is removed from the buffer and returned in AX. The function waits until a scancode/ASCII pair is present in the keyboard input buffer. Except for hardware interrupts, no other processing occurs until a key is pressed on the keyboard.

Input: AH = 10h.
 Output: AH = Scancode.
 AL = ASCII character code, or
 00h for a special scancode, or
 E0h for an extended scancode.
 Error conditions: None.
 Registers Modified: AX.

• AH = 11h Extended Report if Character Available

This function, like function 01h, examines the keyboard input buffer to see if a scancode/ASCII pair is available. However, function 11h recognizes scancode values above 84h. Thus, it can read the extended scancodes. If a scancode/ASCII pair is available, it is returned without removing it from the buffer. The function returns immediately, regardless of whether a scancode/ASCII pair is available.

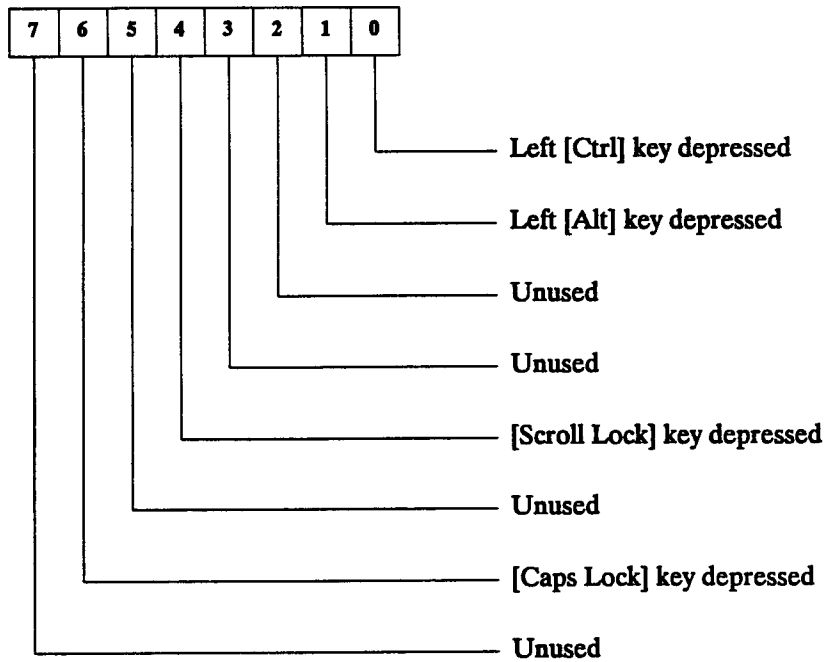
Input: AH = 11h.
 Output: Zero flag is clear if character is available:
 AH = Scancode.
 AL = ASCII character code, or
 00h for a special scancode, or
 E0h for an extended scancode.
 Zero flag is set if no characters are available:
 AX is indeterminate.
 Error conditions: None.
 Registers modified: AX, Status.

• AH = 12h Get Extended Keyboard Status

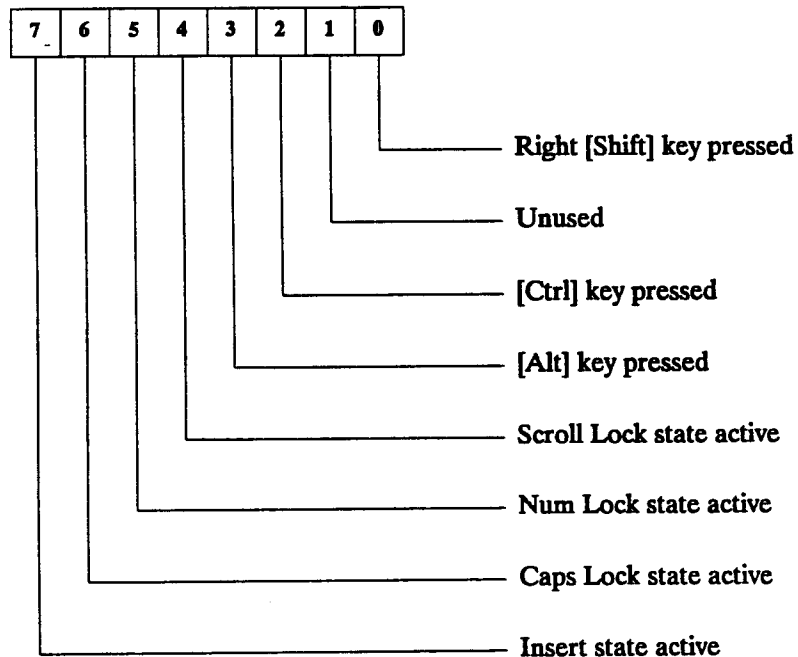
This function returns an extended shift status byte in AH. The byte returned in AL is the same as that returned by function 02h.

Int 16h

Input: AH = 12h.
Output: AH = Status bits as follows:



AL = Status bits as follows:



Error conditions: None.
Registers modified: AX.

- **AH = 13h** Wait for keyboard event

This function waits until a key has been pressed or the keyboard shift flags change. If a new key code is found in the key buffer, it returns with ZF=0 and AX=key code/ascii code for the key. If any of the keyboard shift flags change, it returns with ZF=1 and AX=shift flags.

This routine times out after 9 timer ticks (4950. seconds) and returns even if there have been no keyboard events. If timeout occurs, the routine returns with ZF=1 and AX=shift flags.

Input: **AH = 13h**
 BX = current keyboard shift flags (same format as service 12h)

Output: **Zero flag is clear if character is available:**
 AH = Scancode.
 AL = ASCII character code, or
 00h for a special scancode, or
 E0h for an extended scancode.

Zero flag is set if timeout or keyboard flags changed.
AX is loaded with the keyboard flags as described in service 12h.

Error conditions: **None.**
Registers Modified: **AX, flags**

Int 17h

Int 17h - Printer Services Interrupt

The Int 17h service routine supports one serial printer. This is different from the standard PC-XT bios where Int 17h drives a set of up to three parallel printers. However, it is similar to the situation when the DOS command

MODE LPT1:=COM1:

is used to redirect LPT1 output of a PC to a serial printer. In this case, MS-DOS redirects the INT 17h interrupt vector to point to an interrupt service routine that supports a serial printer.

This service performs several general tests before performing a specified function. It checks to make sure that the port number requested is in the range (0 - 1). It also checks the value of the function requested to make sure it is in range (0 - 2). If any of the tests fail, the service will set AH to 29h and return. The carry flag is not affected by any of the functions.

The printer timeout values are stored at 40h:78h.

The Printer Services interrupt (Int 17h) functions are described below. Specify the desired function code in register AH, with additional parameters passed in other registers as indicated in the table.

NOTE: The printer services change the serial port interrupt vector (INT 0Ch) to point to its own serial port interrupt service routine.

- **AH = 00h** Write a Byte to a serial Printer

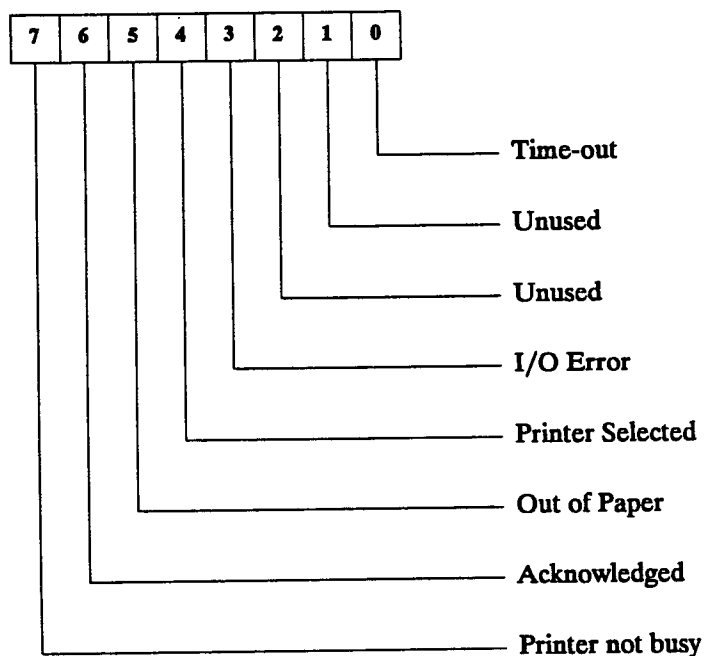
This function checks to see whether an XOFF character has been received. If no XOFF was received, the routine writes a byte to the serial port and returns.

If an XOFF was received, it waits until an XON is received, or the timeout period is elapsed. If the XON is received before the timeout period elapsed, the routine writes a byte of data to a serial printer port and returns. If the timeout period is elapsed before an XON is received, the routine simply returns with bit 0 of AH set.

Input: AH = 00h.
AL = Data byte to be written.
DX = Port number (0 - 1).

Output: AH = Printer status as shown below

Registers modified: AH.



Error conditions: None.

Registers modified: AX.

• **AH = 01h Initialize Printer**

This function initializes a serial printer port. It clears the XOFF received flag. It does not set the baud rate or parity. This should be done with INT 14h function 00.

Input: AH = 01h.
DX = Port number (0 - 1).

Output: AH = Printer status (see function AH=00h)

Registers modified: AH.

• **AH = 02h Get Printer Status**

This function returns the status of the specified serial printer port.

Int 17h

Input:

AH = 02h.

DX = Port number (0 - 1).

Output:

AH = Status byte (see function AH = 00h).

Error conditions: As indicated in AH.

Registers modified: AH.

Int 19h - Boot Interrupt

This service boots the MS-DOS operating system.

Int 1Ah

Int 1Ah - Time-of-Day Services

The Time-of-Day services provide access to the real-time clock and the BIOS clock. The BIOS clock is a software tool that is incremented by the Int 08h service routine once every timer "tick" (hardware timer interrupt). This occurs 18.2065 times per second. The software clock consists of a count of the timer "ticks."

The Time-of-Day Services interrupt (Int 1Ah) functions are described below. Specify the desired function code in register AH, with additional parameters passed in other registers as indicated in the table. If an unsupported function is requested, Int 1Ah will return with the carry flag set.

COMPATIBILITY: For compatibility, the Int 1A services does the following:

- Immediately enable interrupts upon entry into the Int 1A services.
- All functions in the 1Ah set of services return via a RET 2 instruction.
- Interrupts are NOT be disabled while processing functions 0 and 1.

- AH = 00h Read the Current Clock Count

This function returns the number of BIOS clock ticks since midnight. If AL is non-zero, the application should increment the date by one day.

Input: AH = 00h.
Output: AH = 00h.
 CX = Timer "tick" count, most significant word.
 DX = Timer "tick" count, least significant word.
 AL is nonzero if the timer has not been read in 24 hours.
 The carry flag is cleared in the Status register.
Error conditions: None.
Registers modified: AX, CX, DX, Status.

- AH = 01h Set Current Clock Count

This function sets the number of timer "ticks" in the software clock.

Input: AH = 01h.
 CX = Timer "tick" count, most significant word.
 DX = Timer "tick" count, least significant word.
Output: AH = 00h.
 The carry flag is cleared in the Status register.
Error conditions: None.
Registers modified: AH, Status.

- AH = 02h Read the Time From the Real-Time Clock

This function reads the time from the real-time clock (RTC).

Input: AH = 02h.
 Output: AH = 00h.
 CH = Hours in BCD.
 CL = Minutes in BCD.
 DH = Seconds in BCD.
 The carry flag is cleared if the RTC is operating.
 Error conditions: The carry flag is set if the RTC is not operating.
 Registers modified: AX, CX, DX, Status.

- AH = 03h Set the Time in the Real-Time Clock

This function sets the time in the real-time clock.

Input: AH = 03h.
 CH = Hours in BCD.
 CL = Minutes in BCD.
 DH = Seconds in BCD.
 DL = 01h if daylight saving time.
 00h if standard time.
 The carry flag is cleared in the Status register.
 Output: AH = 00h.
 Error conditions: None.
 Registers modified: AX, Status.

- AH = 04h Read Date From the Real-Time Clock

This function reads the date from the real-time clock.

Input: AH = 04h.
 Output: AH = 00h.
 DL = Day in BCD.
 DH = Month in BCD.
 CL = Year in BCD.
 CH = Century:
 19 if 20th century.
 20 if 21st century.
 Error conditions: Carry flag set if the real-time clock is not operating,
 otherwise the carry flag is cleared.
 Registers modified: AX, CX, DX, Status.

- AH = 05h Set Date in Real-Time Clock

This function sets the date in the real-time clock.

Int 1Ah

Input: AH = 05h.
DL = Day in BCD.
DH = Month in BCD.
CL = Year in BCD.
CH = Century:
19 if 20th century.
20 if 21st century.
The carry flag is cleared in the Status register.

Output: AH = 00h.

Error conditions: None.

Registers modified: AX, Status.

- AH = 06h Set Alarm

This function sets the alarm to generate an Int 4Ah at the specified time. The user must place a pointer to an appropriate interrupt handling routine in the Int 4Ah vector. The alarm will reoccur every 24 hours if not reset using AH = 07h.

Input: AH = 06h.
CH = Hours in BCD.
CL = Minutes in BCD.
DH = Seconds in BCD.

Output: AH = 00h.

Error conditions: Carry flag set if the alarm is already set or the real-time clock is not operating, otherwise the carry flag is cleared.

Registers modified: AX, Status.

- AH = 07h Reset Alarm

This function resets the alarm hardware and clears any pending alarm. (An alarm, when set, will reoccur every 24 hours until it is reset.)

Input: AH = 07h.

Output: AH = 00h.
The carry flag is cleared in the Status register.

Error conditions: None.

Registers modified: AX, Status.

- AH = 08h Set Alarm to Power-On the System

This function sets an alarm in the real-time clock. When the alarm time is reached with the system in the powered-down state, the system powers on and boots. If the system is powered on when the alarm time is reached, the alarm interrupt routine will issue an Int 4Ah. The Int 4Ah call is made as if the alarm had been set up by Int 1Ah function 06h.

Input: AH = 08h.
 CH = Hours in BCD.
 CL = Minutes in BCD.
 DH = Seconds in BCD.
 Output: AH = 00h.
 Error conditions: Carry flag set if the alarm is already set or the real-time clock is not operating.
 Registers modified: AX, Status.

• AH = 09h Get the Current Alarm Setting

This function returns the alarm setting currently in the real-time clock.

Input: AH = 09h.
 Output: CH = Hours in BCD.
 CL = Minutes in BCD.
 DH = Seconds in BCD.
 DL = Status of the alarm:
 00h: The alarm is not enabled.
 01h: The alarm is enabled but will not power on the system.
 02h: The alarm is enabled and will power on the system.
 Error conditions: The carry flag is set if the real-time clock is not operating. Otherwise, the carry flag is cleared.
 Registers modified: AX, CX, DX, Status.

• AH = 0Ah Read Day Counter

The BIOS maintains a count of the number of times the software clock has accumulated 24 hours worth of timer "ticks." This variable allows an application to determine how many days have passed since it last read the date.

Input: AH = 0Ah
 Output: AH = 00h.
 CX = Day count (number of times the software clock has overflowed). The carry flag is cleared in the Status register.
 Error conditions: None.
 Registers modified: AX, CX, Status.

• AH = 0Bh Write Day Counter

This function writes CX into the day counter.

Input: AH = 0Bh.
 CX = Value to write to the day counter.
 Output: AH = 00h.
 The carry flag is cleared in the Status register.
 Error conditions: None.
 Registers modified: AX, Status.

Int 1Bh

Int 1Bh - Keyboard Break Interrupt

The Keyboard Break interrupt is called when [Ctrl] [Break] is pressed. It is called from the Keyboard Hardware interrupt (Int 09h) via an Int 1Bh instruction. Applications may chain into the Keyboard Break interrupt. Interrupts are enabled when Int 1Bh is called. (The BIOS points the Int 1Bh vector to an Iret instruction.) Just prior to calling Int 1Bh, the Int 09h routine clears the keyboard buffer. When the Int 1Bh routine returns, the Int 09h routine puts the pseudo scancode/ASCII pair "00h/00h" into the keyboard buffer, then returns.

When the Int 1Bh routine is entered, the BIOS has established a stack frame as shown below:

	Saved Flags
	Saved CS
	Saved IP from Int 9h
	Saved AX
	Saved BX
	Saved CX
	Saved DX
	Saved SI
	Saved DI
	Saved DS
	Saved ES
	Saved Flags
	Saved CS
SP →	Saved IP (from Int 1Bh)

The register contents are as shown below when Int 1Bh is called:

AH: Undefined

AL: Break scancode (46h)

BX: copy of Keyboard buffer read pointer word at 40h:1Ah

DS: 40h

ES: Undefined

Int 1Ch - Timer Tick Interrupt

The Timer Tick interrupt is called from the Timer Hardware interrupt (Int 08h) via an Int 1Ch instruction. The interrupt is called every time the hardware timer issues an interrupt (a timer "tick"). This occurs at a nominal rate of 18.2 Hz (once every 55 ms). Applications may chain into the Timer Tick interrupt. Interrupts are enabled when the Int 1Ch is issued. (The BIOS points the Int 1Ch vector to an Iret instruction.) When the Int 1Ch routine is entered, the BIOS has established a stack frame as shown below:

	Saved Flags
	Saved CS
	Saved IP from Int 8h
	Saved DS
	Saved AX
	Saved DX
	Saved Flags
	Saved CS
SP →	Saved IP (from Int 1Ch)

The register contents are as shown below when Int 1Ch is called:

AX, BX, CX, DX, BP, SI, DI: contain undefined values
 DS: 40h
 ES: Undefined

Int 1Dh

Int 1Dh - Video Parameter Table Pointer

Int 1Dh is a pointer to a video parameter table. This table contains parameters used to initialize the display controller for a particular display mode. The modes are described in the section on Int 10. The video parameter table is structured into fields as shown below:

Field Length (Bytes)	Definition
16	Alphanumeric 40 x 25 initialization parameters.
16	Alphanumeric 80 x 25 initialization parameters.
16	Graphics initialization parameters.
16	Monochrome 80 x 25 initialization parameters.
8*	Video buffer size.
8	Number of columns for each video mode.
8	Video mode parameter bytes.

*The video buffer size field consists of 4 two-byte words.

The initialization parameters can be used to set up the horizontal and vertical size of the display, and to establish the cursor size and position.

The video buffer size parameter can set the size of the graphics buffer for the graphics modes. This parameter should not be used in monochrome mode 7.

The number of columns parameter can be used to indicate the number of columns to display. This parameter can be used for subsequent display address calculations.

The video mode parameter bytes are written to the display hardware to set it up appropriately for a given mode.

Int 1Fh - Graphics Character Table Pointer

This is a pointer to the font table for Code Page 850 characters 128 thru 255. Each character in the character table is defined by eight bytes. The first byte defines eight dots along the top row of the character. The next byte defines eight dots along the next row down, and so on. The character is defined as an 5-by-7 font.

For example, this is the bit representation for the A acute character taken out of the font file:

DB 02H,004H,00EH,011H,01FH,011H,011H,000H ; Char B5: A acute

```
;      . . . . X .
;      . . . X . .
;      . . X X X .
;      . X . . . X
;      . X X X X X
;      . X . . . X
;      . X . . . X
;      . . . . .
```

The BIOS also has a built in character font table for Code Page 850 characters from 00h through 7Fh. There is no interrupt vector pointing to this table, but it is fixed in location at 0f000h:0fa6eh.

Int 4Ah

Int 4Ah - User Alarm Interrupt

Int 4Ah is called from the real-time clock interrupt service routine (Int 70h) when an interrupt is issued by an alarm. Applications may chain into this interrupt so that they may be alerted when an alarm is active. (The BIOS points the Int 4Ah vector to an Iret instruction.) When the Int 4Ah routine is entered, the BIOS has established a stack frame as shown below:

	Saved Flags
	Saved CS
	Saved IP from Int 70h
	Saved DS
	Saved AX
	Saved DI
	Saved DX
	Saved Flags
	Saved CS
SP →	Saved IP (from Int 4Ah)

The register contents are as shown below when Int 4Ah is called:

AL = 0Dh
AH, BX, CX, DX, BP, SI, DI: contain undefined values
DS: 40h
ES: Undefined

Int 63h - eXecute In Place (XIP) Services Interrupt

The Int 63h XIP services provide low-level program location and bank switching capabilities which can be used to run code directly from a plug-in ROM card.

Plug-in card ports 0 and 1 are supported. However, since Jaguar has only one card port, a custom card or a custom extender board would be required to access two cards. When a PCMCIA standard card is inserted into Jaguar, that card resides in Port 0.

The term "logical page" will refer to memory in a plug-in card and the term "physical page" will refer to memory in the address space of the CPU.

Memory on a plug-in card is viewed as a sequence of 16 KB logical pages. The logical pages start on successive 16 KB boundaries starting with logical page 0 which starts at card physical address 0.

Jaguar has four 16 KB physical pages and two 64 KB physical pages. The physical page numbers, starting addresses, and sizes are:

Physical Page Number	Starting Address	Size
3	EC000	16 KB
2	E8000	16 KB
1	E4000	16 KB
0	E0000	16 KB
8	D0000	64 KB
4	C0000	64 KB

The XIP Services interrupt (Int 63h) functions are described in the table below. Specify the desired function code in register AH, with additional parameters passed in other registers as indicated in the table.

- AH = 80h Return File Location and Lock Card Port

This function searches the root directory of plug-in card drives A, D, E, and F (in that order) for the specified file. If the file is found, the card port containing the file is locked.

This function is intended to be used by an XIP loader program to locate an XIP program on a plug-in ROM card. If successful, the port is locked which means that a warm start will result if the card is removed prior to unlocking the port. Thus, the program is assured that the logical pages containing the file are available for mapping until the program explicitly unmaps the pages and unlocks the port.

Int 63h

Input: AH = 80h.
ES:BX = Pointer to an 11 byte buffer containing the file name in the same format as used in a DOS disk directory. That is, the file name left justified and blank filled in bytes 0 through 7 and the extension left justified and blank filled in bytes 8 through 10.

Output: AH = Return status
00h = Success
A2h = File not found
BX = logical page which contains the beginning of the file
CX = offset from beginning of logical page to beginning of file
DX = port number of card that contains the file

Error conditions: As given in AH

Registers modified: AH, BX, CX, DX

- **AH = 81h Unlock Card Port**

This function unlocks a card port which has been previously locked by function 80h. Prior to unlocking, all the mapped pages of the port must be unmapped.

This function is intended to be used by the XIP loader's exit processing so that the card port will be available for use by other programs.

Input: AH = 81h.
DX = port number to unlock

Output: AH = Return status
00h = Success
A0h = Invalid port number
A1h = Port not locked
A3h = Port has mapped pages

Error conditions: As given in AH

Registers modified: AH

- **AH = 82h Map/Unmap Card Port Pages**

This function maps a logical page from a plug-in card memory into the physical address page frames. This function operates on 16 KB pages when requested to map into physical pages 0, 1, 2, or 3, and operates on 64 KB pages when requested to map into physical pages 4 or 8.

For physical page numbers 4 and 8, the logical page number must be a multiple of 4 so that the logical page starts on a 64 KB boundary. In this case four consecutive logical pages are mapped into the C or D bank corresponding to physical pages 4 or 8, respectively.

This function can also unmap physical pages, which makes the previously mapped logical pages inaccessible for reading or writing. You unmap a physical page by setting its associated logical page to FFFFh.

This function is intended to be used by the XIP loader to map XIP code and data into CPU address space for execution. It is subsequently used to unmap card pages prior to unlocking a port.

Input: AH = 82h.
 AL = Physical page number, 0, 1, 2, 3, 4, or 8
 BX = Logical page number
 DX = Port number

Output: AH = Return Status
 00h = Success
 8Ah = Invalid logical page number (physical page is 4 or 8, but logical page is not a multiple of 4)
 8Bh = Invalid physical page number
 A0h = Invalid port number
 A1h = Port not locked

Error conditions: As given in AH

Registers modified: AH

- AH = 83h Get/Set Page Map

This function is composed of four subfunctions which perform general saving and restoring of the page mapping state. The state is saved to or restored from a save array provided in the calling program. One of these subfunctions is used to determine the required length for the array.

These functions are intended to be used to save or restore Jaguar's page map state as a unit; the internal representation of the page map state should not be manipulated.

- AX = 8300h Get Page Map Subfunction

Input: AX = 8300h
 ES:DI = pointer to destination save array

Output: AH = Return Status
 00h = Success

Registers modified: AH

- AX = 8301h Set Page Map Subfunction

Input: AX = 8301h
 DS:SI = pointer to source save array

Output: AH = Return Status
 00h = Success

Registers modified: AH

Int 63h

- **AX = 8302 Get and Set Page Map Subfunction**

Input: **AX = 8302h**
 ES:DI = pointer to destination save array
 DS:SI = pointer to source save array

Output: **AH = Return Status**
 00h = Success

Registers modified: AH

- **AX = 8303 Get Size of Page Map Save Array Subfunction**

Input: **AX = 8303h.**

Output: **AH = Return Status**
 00h = Success
 AL = Save array size in bytes

Registers modified: AH

Appendix A - Compatibility Issues

BIOS Special Compatible Subroutine The following code fragment must be present at the address indicated for compatibility with the Industry Standard.

; This subroutine must be placed here for compatibility.

```
F000:E00D 204942    AND    [BX+DI+42],CL
F000:E010 4D      DEC    BP
F000:E011 C3      RET
F000:E00D      DB      20h, 49h, 42h, 4Dh    ; " IBM"
```

Compatibility Addresses The table below shows the compatible entry points and data table addresses which the Jaguar BIOS system supports.

Int	Rom Entry	Type	Function
	F000:E05B	code	Reset
02	F000:E2C3	code	Nonmaskable Interrupt
19	F000:E6F2	code	Boot
--	F000:E729	data	Baud rate divisor table
14	F000:E739	code	Serial
16	F000:E82E	code	Keyboard
09	F000:E987	code	Keyboard interrupt service routine
13	F000:EC59	code	Disk
0E	F000:EF57	code	Disk int svc routine (Not used in Jaguar)
--	F000:EFC7	data	Disk parameter table (Not found in Jaguar)
17	F000:EFD2	code	Printer
10	F000:F065	code	Video
1D	F000:F0A4	code	Video parameter table
12	F000:F841	code	Memory size
11	F000:F84D	code	Equipment check
15	F000:F859	code	System functions
--	F000:FA6E	data	Character table (lower 128 chars only)
1A	F000:FE6E	code	Time and date
08	F000:FEA5	code	Timer interrupt service routine
--	F000:FEF3	data	Interrupt Vector Table
--	F000:FF23	data	Default Interrupt handler Handles interrupts not specifically handled by the BIOS.
--	F000:FF53	data	Dummy IRET
05	F000:FF54	code	Print screen
--	F000:FFF0	code	Hardware reset point
--	F000:FFF5	data	BIOS date stamp
--	F000:FFFE	data	Hardware ID byte

Appendix B

Appendix B - BIOS Messages

The following is a list of messages display by the BIOS. Most messages will be localized (translated) to foriegn languages, but a few messages will not be localized.

Messages which will be localized to foriegn languages

CR,LF,"RAM Disk Corrupted"

CR,LF,"Initialize RAM Disk? Enter Y or N: "

CR,LF,"Initializing RAM Disk",CR,LF

" MAIN BATTERY LOW "

" BACKUP BATTERY LOW "

" CARD BATTERY LOW "

"Exiting Backup Mode",CR,LF

CR,LF,"Card changed in locked port."

CR,LF,"No stack for NMI."

CR,LF,"Press any key to warm start ..."

CR,LF,"All files on drive C: will be erased!"

CR,LF,"Continue? Enter Y or N:"

Messages which will not be localized

"(C)Copyright Lotus Development Corp 1990",cr,lf

"(C)Copyright Hewlett-Packard 1990",cr,lf

"Copyright 1984,1985",cr,lf,

" Phoenix Software Associates Ltd",CR,LF

"Version @.QA.02",CR,LF

Appendix C - PASSWORD

TECHNICAL DETAILS

The Jaguar password is implemented in a fairly simple fashion. At three places in the main BIOS, there are calls to password routines:

1. At the beginning of the routine which places Jaguar into DeepSleep mode, a check is made to see if the ALT-ON key press is the cause and, if so, a call is made to the ALTON_PWD routine which, if a password is defined, sets a flag which says that the unit is "ALTON-locked".
2. At the end of the routine which brings Jaguar out of DeepSleep mode, a call is made to the CHK_PWD routine which checks two flags, the "ALTON-locked" flag and the "AUTO-locked" flag. If either is set then the display is scrambled and the password is required from the user. If the password fails, the unit is put back to sleep, otherwise the display is unscrambled, the "ALTON-locked" flag is cleared, and the unit is brought fully awake and available for use.
3. At the beginning of the COLDSTART routine (which gets control at CTRL-ALT-DEL and SHIFT-CTRL-ON) a call is made to the CHK_PWD routine, as described above in 2). If the password fails, the unit does not go to sleep (since it's not easily and cleanly done at that point by a simple call), but continues to loop on password entry until the password is entered correctly.

There is also some special code in the INT 16 functions for handling special cases involving an unattended wakeup by an alarm.

Additionally, a PASSWORD command is provided which implements the setting of the password, the setting/clearing of the "ALTON-lock" flag, and the clearing of the "PASSWORD-defined" flag (deleting the password).

The hope is that there is NO way to bypass the password, except to remove the main and backup batteries from the unit (thus losing the contents of the drive C: RAM disk).

USER'S PERSPECTIVE

There is a built-in command PASSWORD, which allows the user to set, change, and remove a password. It can also be used to set and clear an AUTO-LOCK mode. When in AUTO-LOCK mode, *ANY* time the unit turns off, or is re-booted, the password is required before the display contents can be viewed, or the computer used. When not in AUTO-LOCK mode (in MANUAL mode), the password is ONLY required if the user turns the unit off by holding down the ALT key, and then pressing and releasing the ON key. This "locks" the unit, requiring the password upon wakeup.

To set the password, you must go to DOS (by closing all applications, entering the filer, pressing MENU, and selecting System). Then, at the DOS prompt, type

PASSWORD

followed by the ENTER key. You will be prompted:

Appendix C

Enter new password:

At this point you should type in your password, which can consist of from 1 to 12 alphanumeric characters (ASCII values from 32 through 255). Control characters are not allowed and will abort password entry. After typing in your password, press ENTER. You will now be prompted:

Verify new password:

You should re-enter your password at this point and press ENTER. If the two passwords don't match, it will not be accepted, the password will not be set, and you will see the message:

Verify failed. Password unchanged.

Otherwise, the password will not be set, and you will see:

Password changed.

You may now turn the unit off AND secure it by pressing ALT-ON. When turned on again, most of the display will be blank, with a small box in the center containing the characters:

pwd:

At this point, you can turn the unit back off, type in a password and press ENTER, or do nothing. If the password is not typed correctly, the unit will beep and turn itself off. If you re-boot with CTRL-ALT-DEL or SHIFT-CTRL-ON, you will still be prompted for the password. At that point, if the password is not typed correctly, the unit will beep, but will NOT turn off.

When a unit is turned on after having been secured from graphics mode, the display will be scrambled and look like a TV screen after the station has gone off the air, but will have the 'pwd:' message in a box in the center of the display.

After a password has been specified, it can be changed by going to DOS and typing:

PASSWORD

You will be prompted:

Enter old password:

You will need to correctly enter your old password before being allowed to change the password. This prevents an unauthorized person from changing your password without your knowledge, preventing you from accessing your own machine. After your old password has been correctly entered, the new password is then entered in the same fashion as the first password (as documented above).

To enable AUTO-LOCK, go to DOS and type:

PASSWORD /A

('A' for Automatic.) To disable AUTO-LOCK, type:

PASSWORD /M

('M' for Manual.) To completely remove password protection, type:

PASSWORD /D

('D' for Delete.) Hence, the complete syntax for the PASSWORD command would look something like:

PASSWORD [/A | /M | /D]

If the machine has been shut off with password protection enabled, it can be awakened by a key press, an alarm (set in the appointment book), or a series of characters arriving over the serial cable. If the cause of the awakening is an alarm or serial characters, the unit will 'beep' for approximately four seconds or until a key is pressed. You must still enter the password correctly before being able to use the unit. No keys are discarded, so if the unit is beeping, just start typing the password and the beeping will stop after the first key press.

If the cause of the awakening was an alarm, the appointment book will display it's alarm message and start beeping immediately after the password has been entered.

If an alarm goes off and the user does not press any keys for about 15 seconds, the password code will allow execution to continue on into the main system so that the next alarm can be set, but the keyboard is locked out during this period, maintaining the systems security.

It should be recommended to the user that he set up a password, whether it's used or not, simply to prevent a malicious person from setting an unknown password that would lock the user out of his own machine. If the password LOCK is left in "manual" mode (the default) then the password will never intrude itself unless the user requests it (by pressing ALT-ON).

File Specifications for HP 95LX Built-in Applications

HP 95LX Appointment Book File Format

The HP 95LX Appointment Book file is structured as a file-identification record, followed by a settings record, followed by a variable number of data records, and terminated by an end-of-file record. There are multiple types of data records corresponding to the different types of appointment book entries.

The formats of these appointment book records is described in the following tables. In the descriptions, the type **int** refers to a two-byte integer stored least significant byte first, the type **swpint** refers to a two-byte integer stored most significant byte first, the type **char** refers to a one-byte integer, and the type **ASCII** refers to a string of ASCII characters.

HP 95LX Appointment Book File Identification Record

Byte Offset	Name	Type	Contents
0	ProductCode	int	-1 (FFh, FFh).
2	ReleaseNum	int	1 (01h, 00h).
4	FileType	char	1 (01h).

HP 95LX Appointment Book Settings Record

Byte Offset	Name	Type	Contents
0	StartTime	int	Daily display start time as the number of minutes past midnight.
2	Granularity	int	Daily display time line granularity in minutes.
4	AlarmEnable	char	1 = on, 0 = off.
5	LeadTime	char	Alarm default lead time in minutes.
6	CarryForward	char	To do carry forward default: 1 = on, 0 = off.

HP 95LX Appointment Book Daily Data Record

Byte Offset	Name	Type	Contents
0	RecordType	char	1 (01h).
1	RecordLength	int	Number of bytes in the remainder of this data record — see note 1.
3	ApptState	char	See note 2.
4	Year	char	Year counting from 1900.
5	Month	char	Month, 1 — 12.
6	Day	char	Day, 1 — 31.
7	StartTime	swpint	Start time in minutes since midnight.
9	EndTime	int	End time in minutes since midnight.
11	LeadTime	char	Alarm lead time in minutes, 0 — 30.
12	ApptLength	char	Length of appointment text in bytes.
13	NoteLength	int	Length of note text in bytes.
15	ApptText	ASCII	Appointment text — see note 4 below.
15+ApptLength	NoteText	ASCII	Note text where the null character is used as the line terminator — see note 5 below.

HP 95LX Appointment Book Weekly Data Record

Byte Offset	Name	Type	Contents
0	RecordType	char	2 (02h).
1	RecordLength	int	Number of bytes in the remainder of this data record — see note 1.
3	ApptState	char	See note 2.
4	DayOfWeek	char	Day of week, 1 = Sun., . . . , 7=Sat.
5	StartTime	swpint	Start time in minutes since midnight.
7	StartYear	char	Start year counting from 1900.
8	StartMonth	char	Start month, 1 — 12.
9	StartDay	char	Start day, 1 — 31.
10	EndTime	int	End time in minutes since midnight.
12	EndYear	char	End year counting from 1900.
13	EndMonth	char	End month, 1 — 12.
14	EndDay	char	End day, 1 — 31.
15	LeadTime	char	Alarm lead time in minutes, 0 — 30.
16	ApptLength	char	Length of appointment text in byte.s
17	NoteLength	int	Length of note text in bytes — see note 5 below.
19	ApptText	ASCII	Appointment text — see note 4 below.
19+ApptLength	NoteText	ASCII	Note text where the null character is used as the line terminator — see note 5 below.

HP 95LX Appointment Book Monthly by Date Data Record

Byte Offset	Name	Type	Contents
0	RecordType	char	3 (03h).
1	RecordLength	int	Number of bytes in the remainder of this data record — see note 1.
3	ApptState	char	See note 2.
4	DayOfMonth	char	Day of month, 1 — 31.
5	StartTime	swpint	Start time in minutes since midnight.
7	StartYear	char	Start year counting from 1900.
8	StartMonth	char	Start month, 1 — 12.
9	StartDay	char	Start day, 1 — 31.
10	EndTime	int	End time in minutes since midnight.
12	EndYear	char	End year counting from 1900.
13	EndMonth	char	End month, 1 — 12.
14	EndDay	char	End day, 1 — 31.
15	LeadTime	char	Alarm lead time in minutes, 0 — 30.
16	ApptLength	char	Length of appointment text in bytes.
17	NoteLength	int	Length of note text in bytes.
19	ApptText	ASCII	Appointment text — see note 4 below.
19+ApptLength	NoteText	ASCII	Note text where the null character is used as the line terminator — see note 5 below.

HP 95LX Appointment Book Monthly by Position Data Record

Byte Offset	Name	Type	Contents
0	RecordType	char	4 (04h).
1	RecordLength	int	Number of bytes in the remainder of this data record — see note 1.
3	ApptState	char	See note 2.
4	WeekOfMonth	char	Week of month, 1 — 5.
5	DayOfWeek	char	Day of week, 1 = Sun., ... , 7 = Sat.
6	StartTime	swpint	Start time in minutes since midnight.
8	StartYear	char	Start year counting from 1900.
9	StartMonth	char	Start month, 1 — 12.
10	StartDay	char	Start day, 1 — 31.
11	EndTime	int	End time in minutes since midnight.
13	EndYear	char	End year counting from 1900.
14	EndMonth	char	End month, 1 — 12.
15	EndDay	char	End day, 1 — 31.
16	LeadTime	char	Alarm lead time in minutes, 0 — 30.
17	ApptLength	char	Length of appointment text in bytes.
18	NoteLength	int	Length of note text in bytes.
20	ApptText	ASCII	Appointment text — see note 4 below.
20+ApptLength	NoteText	ASCII	Note text where the null character is used as the line terminator — see note 5 below.

HP 95LX Appointment Book Yearly Data Record

Byte Offset	Name	Type	Contents
0	RecordType	char	5 (05h).
1	RecordLength	int	Number of bytes in the remainder of this data record — see note 1.
3	ApptState	char	See note 2.
4	MonthOfYear	char	Month of year, 1 = Jan., ... , 12 = Dec.
5	DayOfMonth	char	Day of month, 1 — 31.
6	StartTime	swpint	Start time in minutes since midnight.
8	StartYear	char	Start year counting from 1900.
9	StartMonth	char	Start month, 1 — 12.
10	StartDay	char	Start day, 1 — 31.
11	EndTime	int	End time in minutes since midnight.
13	EndYear	char	End year counting from 1900.
14	EndMonth	char	End month, 1 — 12.
15	EndDay	char	End day, 1 — 31.
16	LeadTime	char	Alarm lead time in minutes, 0 — 30.
17	ApptLength	char	Length of appointment text in bytes.
18	NoteLength	int	Length of note text in bytes.
20	ApptText	ASCII	Appointment text — see note 4 below.
20+ApptLength	NoteText	ASCII	Note text where the null character is used as the line terminator — see note 5 below.

HP 95LX Appointment Book To Do Data Record

Byte Offset	Name	Type	Contents
0	RecordType	char	6 (06h).
1	RecordLength	int	Number of bytes in the remainder of this data record — see note 1.
3	ToDoState	char	See note 3.
4	Priority	char	Priority, 1 — 9.
5	StartYear	char	Start year counting from 1900.
6	StartMonth	char	Start month, 1 — 12.
7	StartDay	char	Start day, 1 — 31.
8	CheckOffYear	char	Check off year counting from 1900; 0 indicates not checked off.
9	CheckOffMonth	char	Check off month, 1 — 12; 0 indicates not checked off.
10	CheckOffDay	char	Check off day, 1 — 31; 0 indicates not checked off.
11	ToDoLength	char	Length of to do text in bytes.
12	NoteLength	int	Length of note text in bytes.
14	ToDoText	ASCII	To do text — see note 4 below.
14+ToDoLength	NoteText	ASCII	Note text where the null character is used as the line terminator — see note 5 below.

HP 95LX Appointment Book End of File Record

Byte Offset	Name	Type	Contents
0	RecordType	char	50 (32h).
1	RecordLength	int	0 (00h, 00h).

Notes:

1. Files created by the Appointment Book application may contain some padding following the last field of some data records. Hence, the RecordLength field must be used to determine the start of the next record. Appointment book files created by other programs do not require any padding.
2. ApptState has several bit fields. Only bit 0 is meaningful to software processing an appointment book file. Bit 0 being set or cleared corresponds to the alarm being enabled or disabled. Programs creating Appointment book files should clear all bits, except perhaps bit 0.
3. ToDoState has two one-bit bit fields. Bit 0 being set or cleared corresponds to 'carry forward' being enabled or disabled for this todo item. Bit 1 being set or cleared corresponds to the todo being checked off or not checked off.
4. Appointment and ToDo texts are each limited to a maximum of 27 characters.
5. Note text is limited to a maximum of 11 lines of 39 characters per line (not counting the line terminator).

HP 95LX Phone Book File Format

An HP 95LX Phone Book file is structured as a file-identification record, followed by a variable number of phone book data records, and terminated by an end-of-file record. Each data record contains the information for one phone book entry.

The formats of these phone book records is described in the following tables. In the descriptions, the type int refers to a two-byte integer stored least significant byte first, the type char refers to a one-byte integer, and the type ASCII refers to a string of ASCII characters.

HP 95LX Phone Book File Identification Record

Byte Offset	Name	Type	Contents
0	ProductCode	int	-2 (FEh, FFh).
2	ReleaseNum	int	1 (01h, 00h).
4	FileType	char	3 (03h).

HP 95LX Phone Book Data Record

Byte Offset	Name	Type	Contents
0	RecordType	char	1 (01h).
1	RecordLength	int	Number of bytes in the remainder of this data record — see NOTE below.
3	NameLength	char	Length of name text in bytes.
4	NumberLength	char	Length of number text in bytes.
5	AddressLength	int	Length of address text in bytes.
7	NameText	ASCII	Name text, 30 characters maximum.
7+NameLength	NumberText	ASCII	Number text, 30 characters maximum.
7+NameLength+ NumberLength	AddressText	ASCII	Address text where the null character is used as the line terminator. Addresses are limited to a maximum of 8 lines of 39 characters per line (not counting the line terminator).

HP 95LX Phone Book End of File Record

Byte Offset	Name	Type	Contents
0	RecordType	char	2 (02h).
1	RecordLength	int	0 (00h, 00h).

Note



Files created by the Phone Book application may contain some padding following the last field of some data records. Hence, you must use the RecordLength field to determine the start of the next record. Phone book files created by other programs do not require any padding.

HP 95LX Memory Management

Introduction

This chapter discusses ROM-executable XIP programs for the HP 95LX plug-in ROM cards.

The definition of the HP 95LX hardware is now complete, but software tools and support services are not. Since there are no standards for creating XIP software in the DOS environment, some of the eventual tools and services may depend on the needs of ISVs.

The PCMCIA is currently working on standards for XIP software. As of this printing, they have agreed on a basic level of hardware support. This level of support is built into the HP 95LX.

Bank Switch Areas

The HP 95LX hardware supports bank switching of plug-in card memory into the CPU address space as follows:

Start	Length	Contents
C0000h	64K Page	Selectable on 64K boundary
D0000h	64K Page	Selectable on 64K boundary
E0000h	16K Page	Selectable on 16K boundary
E4000h	16K Page	Selectable on 16K boundary
E8000h	16K Page	Selectable on 16K boundary
EC000h	16K Page	Selectable on 16K boundary

For ease of reference,

- The two 64K sections in the C and D blocks are designated as “code pages.”

Code pages are selectable on 64K boundaries—64K portions of the plug-in card that fall on 64K boundaries can be mapped into either the C or D block of the CPU address space.

- The four 16K sections in the E block are designated as “data pages.”

Data pages are selectable on 16K boundaries—16K portions of the plug-in card that fall on 64K boundaries can be mapped into any of the 16K subdivisions of the E block of the CPU address space.

ROM Card Structure

Applications using XIP will be distributed on a ROM card that the user inserts in the HP 95LX's plug-in card slot.

Plug-in ROM cards typically contain a ROM-disk structure at their beginnings. ROM disks each contain a stub program that is used to start the XIP program. Many applications will have additional files that they desire to access through DOS. These typically include help files, example files, and other files of application-specific data. All such files will be placed on the ROM disk that is accessible to the HP 95LX as the A: drive.

In addition to its ROM-disk portion, a plug-in ROM card may contain blocks of ROM-executable code intended to be bank switched into the code pages mentioned above.

XIP Program Execution

For an independent XIP program, the stub program that resides on the ROM disk will be loaded and executed as a standard DOS program. The stub program performs the necessary bank switching to access its code blocks.

For a System-Manager-compliant application, the stub program will be loaded by the System Manager.

XIP Bank Switching Services

The two fundamental services provided in the HP 95LX are:

1. A service that returns the card address of the XIP code. The stub program calls this service to determine the logical address of the XIP code it desires to access.
2. A service that bank switches a logical page of the plug-in card into a physical code or data page on the HP 95LX.

XIP Tools

Additional tools that we expect to provide in the full Developer's Kit are:

- ROM disk image builder that takes as input a collection of files and outputs a binary image of a ROM disk that can be included at the beginning of the plug-in ROM card.
- Locator tools that prepare XIP code to execute from the HP 95LX's code pages.
- ROM image builder that combines the ROM disk image with any blocks of XIP code to create a complete ROM image.
- Example stub programs that perform bank switching and other control functions typically needed by applications.

HP 95LX Low-Level Graphics Support

Introduction

The package of low-level routines built into the HP 95LX that are available for use by applications programs support these functions:

1. **Set Video Mode** (Set the display to alpha or graphics mode.)
2. **Set Fill Mask**
3. **Graphics Settings** (Get information about current graphics settings.)
4. **Set Logical Origin** (Specify origin to which all graphics operations relate.)
5. **Set Clip Region** (Specify coordinates of upper-left and lower-right corners of a rectangle.)
6. **Draw Rectangle** (Specify diagonally opposite corners of a rectangle.)
7. **Draw Line**
8. **Plot Point**
9. **Move Pen**
10. **Set Pen Color**
11. **Set Replacement Rule** (Specify how pen color combines with pixel color when plotting.)
12. **Set Line-type**
13. **Read Point** (How to read the color value of a point.)
14. **Read Area** (Reads rectangular area of display into specified buffer.)
15. **Write Area** (Writes to rectangular area of display from specified buffer.)
16. **Write Text** (Writes specified text to specified location of display.)

All of the routines which plot to the display always obey the current logical origin, clip region, pen color, replacement rule, and (where appropriate) line-type and fill-mask.

The pen color can be 0 or 1.

The replacement rule can be one of FORCE, AND, OR, or XOR. Writing a rectangular image can optionally invert the image before applying the specified replacement rule. Writing an image is different from all other plotting in that it uses an argument as the replacement rule rather than the current replacement rule.

The line-type is a 16-bit value whose bits are used repeatedly when drawing a line or an outlined rectangle.

The fill-mask is an 8-byte value which specifies an 8-bit by 8-bit rectangular mask which is used repeatedly when drawing a pattern-filled rectangle.

The general process to do graphics is:

1. Set the display mode to graphics.
2. Set the desired pen color, replacement rule, linetype, fillmask, logical origin, and clip region, if different than the default values set by the set-mode function.
3. Perform the desired drawing using the attributes setup by step 2).
4. Repeat steps 2) and 3) until done.
5. Set the display mode back to alpha.

The graphics routines are accessed through software interrupt 5F (hex). The required arguments are loaded into specific CPU registers, the requested function number is loaded into the AH register, and then an INT 5Fh instruction is executed. Unless otherwise stated, all functions preserve ALL registers except for AX.

X-coordinates always get larger (more positive) when moving to the right on the display. Y-coordinates always get larger when moving down (towards the bottom) on the display. The default origin is in the top-left corner of the display.

Although the interface is designed primarily as an assembly language interface, it is simple to write an assembly language module that can provide a library of corresponding functions to a C program. A sample is provided below.

Set Video Mode

This routine forces the current video mode to alpha or graphics and clears the display.

Entry conditions:

AH = 0

AL = requested mode:

07h = alpha (system manager compliant).

20h = graphics (system manager compliant).

87h = alpha (non-system manager).

A0h = graphics (non-system manager).

Sub-functions 07h and 20h call the System Manager routine to change video modes, thus letting the System Manager know that the display has changed modes and the display contents destroyed. This is important for any applications which are intended to be system-manager compliant.

Subfunctions 87h and A0h call the BIOS directly in order to change display modes, thus by-passing the System Manager. These should be used by programs that are NOT system-manager compliant, so that they will function correctly whether the system was booted into the system manager or if it was booted straight into DOS (by placing a SHELL= command in a CONFIG.SYS file).

After a SET_MODE call to change to graphics, the defaults are:

Operation	Coordinates
Logical origin	(0, 0)
Clip region	(0, 0) thru (239, 127)
Pen location	(0, 0)
Pen color	1
Replacement rule	FORCE
Line-type	0FFFFh
Fill mask	0FFh,0FFh,0FFh,0FFh,0FFh,0FFh,0FFh,0FFh

Sample assembly code:

```
mov    ax,0020h    ; set mode to GRAPHICS
int     5fh
.
.
.
mov    ax,0007h    ; set mode to ALPHA
int     5fh
```

Set Fill Mask

This routine sets the eight-byte fill mask used by `DRAW_RECTANGLE` when pattern-filling.

Entry conditions:

`AH = 1`

`ES:DI` = address of 8 bytes of fillmask.

The default fillmask after a mode set to graphics is eight bytes of `0ffh` (which would result in a solid-fill).

The fillmask is always aligned with the byte boundaries of the displaymemory, and it is then clipped by the rectangle being drawn. This means that as the rectangle is shifted bit-by-bit, the pattern appears to exist on a plane behind the rectangle, and that the rectangle is a moving window onto that plane. It's tough to describe, and a little experimentation should make it plain.

Sample assembly code:

```
fmask1  db      055h, 0aah, 055h, 0aah, 055h, 0aah, 055h, 0aah
        .
        .
        .
assume  es:dgroup
lea     di, fmask1
mov     ah, 1
int     5fh      ; set fillmask pattern to FMASK1
```

Get Current Graphics Information

This routine returns current information about the state of the graphics functions.

Entry conditions:

AH = 2

ES:DI = address of a 36-byte long buffer into which the graphics information will be placed.

At exit from this function, the buffer contents will be:

Offset	Size	Description
0	1-byte	CURRENT VIDEO MODE
1	1-byte	DEFAULT VIDEO MODE
2	1-word	WIDTH OF DISPLAY IN PIXELS
4	1-word	HEIGHT OF DISPLAY IN PIXELS
6	1-word	CURRENT X-LOCATION OF PEN
8	1-word	CURRENT Y-LOCATION OF PEN
10	1-word	CURRENT LINE-TYPE
12	1-word	CURRENT REPLACEMENT RULE
14	1-word	CURRENT PEN COLOR
16	1-word	CURRENT X-MINIMUM OF CLIP REGION
18	1-word	CURRENT X-MAXIMUM OF CLIP REGION
20	1-word	CURRENT Y-MINIMUM OF CLIP REGION
22	1-word	CURRENT Y-MAXIMUM OF CLIP REGION
24	1-word	CURRENT X-LOCATION OF LOGICAL ORIGIN
26	1-word	CURRENT Y-LOCATION OF LOGICAL ORIGIN
28-35	8-bytes	CURRENT FILL MASK (for rectangle fill)

At exit:

DX:AX = address of the 36-byte long buffer (for return to C).

Sample assembly code:

infobuf	label	byte	
curmode		db	?
defmode		db	?
dspwidth	dw	?	
dspheight	dw	?	
curpenx	dw	?	
curpeny	dw	?	
curlinetype	dw	?	
curreprule	dw	?	
curpen	dw	?	
curclipminx	dw	?	
curclipmaxx	dw	?	
curclipminy	dw	?	
curclipmaxy	dw	?	
curlogorgx	dw	?	

```

curlogorgy    dw      ?
curfmask      db      8 dup    (?)
               .
               .
               .
assume        es:dgroup
lea           di,infobuf
mov           ah,2           ; read currentvideo info into infobuf
int           5fh

```

Set Logical Origin

This routine sets the logical origin in terms of absolute screen pixels, regardless of previous settings of the logical origin or clip region. The (X,Y) of the logical origin may be specified off of the actual physical screen (ie, negative values or greater than (239, 127). All other coordinate arguments in this graphics system are relative to the logical origin, including those used to specify the clip region.

SET LOGICAL ORIGIN resets the CLIP REGION to the entire physical display (0,0) to (239,127). So, if clip_region is used, it must be set AFTER the set_logical_origin.

The default logical origin after a mode set is (0,0).

Entry conditions:

```

AH = 3
CX = x coordinate
DX = y coordinate

```

Sample assembly code:

```

mov     ah,3
mov     cx,120    ; move log org to approximately the
mov     dx,64     ; center of the display
int     5fh

```

Set Clip Region

This routine sets the coordinates of the upper-left and lower-right corners of the clip rectangle.

All reading/writing of the display in this graphics system is limited (clipped) by the current CLIP REGION.

Default after mode set is (0,0) and (239,127).

Entry conditions:

AH = 4

CX = x-minimum coordinate

DX = y-minimum coordinate

SI = x-maximum coordinate

DI = y-maximum coordinate

Sample assembly code:

```
mov    ah,4      ; clip (limit) all drawing to the
mov    cx,120    ; right half of the display
mov    dx,0
mov    si,239
mov    dx,127
int    5fh
```

Draw Rectangle

This routine draws a rectangle which has two diagonally opposite corners at the current pen location and (CX,DX). Hence, you will usually first do a MOVE_PEN, then a DRAW_RECTANGLE. ALL rectangle draws obey the current replacement rule. The pen location is left at the starting location (it is not changed).

If the rectangle drawn is just an outline, it is drawn using the current line-type. If the rectangle drawn is pattern-filled, it uses the current fill-mask. In all cases the current pen color and replacement rule are used.

Entry conditions:

AH = 5

AL = fill flag 0==outline, current linetype and pen color.

1==solid fill, current pen color.

2==pattern fill, current fillmask and pen color.

CX = x-coordinate of second corner of rectangle.

DX = y-coordinate of second corner of rectangle.

Sample assembly code:

```
mov     ah,8           ; move pen to (50,74)
mov     cx,50
mov     dx,74
int     5fh

mov     ah,5
mov     al,1           ; solidfill rectangle to (101, 99)
mov     cx,101
mov     dx,99
int     5fh
```


Draw Line

This routine draws a line from the current pen location to (CX,DX) using the current pen color, linetype, and replacement rule. The pen location is left at the end point. If another DRAW_LINE is executed after the first without an intervening MOVE_PEN, the starting point is not plotted. This is to avoid the problem of drawing connecting lines with a replacement rule of XOR. Since the starting point of the second line is the same as the ending point of the first line, it would get plotted twice, which in XOR mode is the same as not plotting it at all.

Entry conditions:

AH = 6

CX = x-coordinate of end point.

DX = y-coordinate of end point.

Sample assembly code:

```
mov    ah,6
mov    cx,21
mov    dx,10      ; draw from current pen location
int    5fh        ; to (21, 10)
```

Plot Point

This routine moves the current pen location to CX,DX and plots a single point there using the current pen color and replacement rule.

Entry conditions:

AH = 7

CX = x-coordinate of point.

DX = y-coordinate of point.

Sample assembly code:

```
mov    ah,7
mov    cx,239      ; plot point at (239, 127)
mov    dx,127
int    5fh
```

Move Pen

This routine moves the current pen location to (CX,DX). The default location after a mode set is (0,0).

Entry conditions:

AH = 8
CX = x-coordinate.
DX = y-coordinate.

Sample assembly code:

```
mov     ah,8
mov     cx,22      ; move pen to (22, 44)
mov     dx,44
int     5fh
```

Set Pen Color

This routine sets the current pen color to 0 or 1. The default after a mode set is 1 (black).

Entry conditions:

AH = 9
AL = new pen color (0 for white or 1 for black)

Sample assembly code:

```
mov     ah,9
mov     al,0      ; set pen color to white (0)
int     5fh
```

Set Replacement Rule

The replacement rule controls how the current pen color is combined with the existing color of a pixel on the display when performing any plotting function (except WRITE_AREA, which has its own replacement rule specified with each call). If the current replacement rule is FORCE then the resulting color of a pixel is equal to the current pen color at the time of the plotting action. For the other three replacement rules (AND, OR, and XOR), the resulting color is the logical operation between the current screen pixel color and the current pen color.

The default replacement rule after a mode set is 0 (FORCE).

Entry conditions:

AH = 10 (0Ah)
AL = new replacement rule 0==FORCE
 1==AND
 2==OR
 3==XOR

Sample assembly code:

```
mov     ah,10
mov     al,3      ; set replacement rule to XOR
int     5fh
```

Set Line Type

This is a 16-bit value that is repeated over and over as each pixel of a line or an outlined rectangle is drawn. 0xFFFF will cause solid lines to be drawn. The default after a mode set is 0xFFFF (solid line).

Entry conditions:

AH = 11
(0Bh) CX = new line type

Sample assembly code:

```
mov     ah,11
mov     cx,0c440h ; set line type to XX000X000X000000
int     5fh
```

Read Point

This routine returns AX==the color (0 for white, 1 for black) of the requested point. The current pen location is not modified.

Entry conditions:

AH = 12 (0Ch)

CX = x-location of point to read.

DX = y-location of point to read.

Sample assembly code:

```
mov     ah,12
mov     cx,49      ; read point (49,57)
mov     dx,57
int     5fh
```

Read Area (get image)

This routine reads a rectangular area of the display into the specified buffer. There is an 8-byte header at the beginning (specifying number of planes, number of bits/pixel, width of image, and height of image. The first two are always equal to 1 on the HP 95LX. The size needed for the buffer is: $8 + ((x2-x1+8)/8) * (y2-y1+1)$ bytes.

The diagonally opposite corner points (x1, y1) and (x2, y2) are included in the read area. Bits with a value of 0 are added to the right end of each row of pixels (if necessary) to fill out an integral number of bytes of data for that row. The image is always left justified within the buffer regardless of its byte-alignment on the display.

Entry conditions:

AH = 13 (0Dh)

CX = x-coordinate of corner 1.

DX = y-coordinate of corner 1.

SI = x-coordinate of corner 2.

BP = y-coordinate of corner 2.

ES:DI = address of buffer for image.

Sample assembly code:

```
tmpbuf  db          56 dup (?)
.
.
.
assume  es:dgroup
mov     ah,13
mov     cx,24
mov     dx,55
mov     si,41      ; read a rectangular area of the screen
mov     bp,70      ; from (24,55) thru (41,70) into
lea     di,tmpbuf   ;'tmpbuf'
int     5fh
```

Write Area (put image)

This writes to a rectangular area of the display from a specified buffer. The buffer should have the same eight-byte header described in READ_AREA (above). This call is different from all other “write”-type calls in that it specifies its own replacement rule rather than using the “current” replacement rule. It expands upon the replacement rule types by allowing the image to be inverted before being combined in the usual fashion (according to FORCE, AND, OR, or XOR) with the display contents. This does not modify the contents of the buffer. If the entire image doesn’t fit on the display, none of it is drawn.

Entry conditions:

AH = 14 (0Eh)

AL = replacement rule 0 FORCE
 1 AND
 2 OR
 3 XOR
 4 invert image and then FORCE
 5 invert image and then AND
 6 invert image and then OR
 7 invert image and then XOR

CX = x-location of top-left corner of image destination.

DX = y-location of top-left corner of image destination.

ES:DI = address of image.

Sample assembly code:

```
tmpbuf  db          56 dup (?)
.
.
.
assume  es:dgroup
mov     ah,14
mov     al,7          ; invert, then XOR
mov     cx,133
mov     dx,66
lea     di,tmpbuf     ; put image 'tmpbuf' at (133,66)
```

Write Text

This routine writes the specified text (all 256 chars are legal EXCEPT 0) to the specified location, horizontally or rotated 90 degrees counter-clockwise from horizontal, using the BIOS 6x8 font, the current pen color and replacement rule. The specified location (CX, DX) is the top-left corner of the text string, or if the rotate flag is non-zero, the string is rotated 90 degrees counterclockwise about the point (CX, DX) (such that it is now the bottom-left corner of the text).

Entry conditions:

AH = 15 (0Fh)

AL = rotate flag.

CX = x-coordinate of top-left corner of first character.

DX = y-coordinate of top-left corner of first character.

ES:DI = address of null-terminated string.

Sample assembly code:

```
txtstr    db          "This is a test",0
.
.
.
assume    es:dgroup
mov       ah,15
mov       al,0          ; plot text horizontally.
mov       cx,124
mov       dx,37         ; at (124, 37)
lea       di,txtstr
int       5fh
```

Sample assembly language module of C-callable functions

; Graphics interface module for calling The HP 95LX graphics from C programs.
; Copyright 1990 Hewlett Packard Company. All rights reserved.
; Author: Everett Kaser August 14, 1990.

.MODEL LARGE,C

.CODE

assume ds:nothing

; G_Mode(int BiosVideoMode);

G_Mode PROC BiosVideoMode:word
mov ax,BiosVideoMode
xor ah,ah
int 5fh
ret
G_Mode endp

; G_FillMask(maskptr);

G_FillMask PROC uses ES DI, maskptr:dword
les di,maskptr
mov ah,1
int 5fh
ret
G_FillMask endp

; _GetInfo(G_INFO *gp);

G_GetInfo PROC uses ES DI, gp:dword
les di,gp
mov ah,2
int 5fh
ret G_GetInfo endp

; G_LorgA(int x, int y);

G_LorgA PROC x:word, y:word
mov cx,x
mov dx,y
mov ah,3
int 5fh
ret
G_LorgA endp

```

; G_ClipL(int xmin, int ymin, int xmax, int ymax);
    G_ClipL PROC uses SI DI, xmin:word, ymin:word, xmax:word, ymax:word
        mov cx,xmin
        mov dx,ymin
        mov si,xmax
        mov di,ymax
        mov ah,4
        int 5fh
        ret
    G_ClipL endp

; G_Rect(int x, int y, int fillflag);
    G_Rect PROC x:word, y:word, fill:word
        mov cx,x
        mov dx,y
        mov ax,fill
        mov ah,5
        int 5fh
        ret
    G_Rect endp

; G_Draw(int x, int y);
    G_Draw PROC x:word, y:word
        mov cx,x
        mov dx,y
        mov ah,6
        int 5fh
        ret
    G_Draw endp

; G_Point(int x, int y);
    G_Point PROC x:word, y:word
        mov cx,x
        mov dx,y
        mov ah,7
        int 5fh
        ret
    G_Point endp

; G_Move(int x, int y);
    G_Move PROC x:word, y:word
        mov cx,x
        mov dx,y
        mov ah,8
        int 5fh
        ret
    G_Move endp

```



```

; G_ColorSel(int color);
    G_ColorSel PROC color:word
        mov ax,color
        mov ah,9
        int 5fh
        ret
    G_ColorSel endp

; G_RepRule(int rrule);
    G_RepRule PROC rrule:word
        mov ax,rrule
        mov ah,0ah
        int 5fh
        ret
    G_RepRule endp

; G_LineType(int ltype);
    G_LineType PROC ltype:word
        mov cx,ltype
        mov ah,0bh
        int 5fh
        ret
    G_LineType endp

; G_PointRead(int x, int y);
    G_PointRead PROC x:word, y:word
        mov cx,x
        mov dx,y
        mov ah,0ch
        int 5fh
        ret
    G_PointRead endp

; G_ImageGet(int x1, int y1, int x2, int y2, char far *image);
    G_ImageGet PROC uses ES SI DI, x1:word, y1:word, x2:word, y2:word, image:dword
        mov cx,x1
        mov dx,y1
        mov si,x2
        les di,image
        mov bp,y2
        mov ah,0dh
        int 5fh
        ret
    G_ImageGet endp

```

```

; G_ImagePut(int x, int y, char far *image, int replacerule);
    G_ImagePut PROC uses ES DI, x:word, y:word, image:dword, reprule:word
        mov cx,x
        mov dx,y
        les di,image
        mov ax,reprule
        mov ah,0eh
        int 5fh
        ret
    G_ImagePut endp

; G_Text(int x, int y, char far *string, rotflag);
    G_Text PROC uses DS ES SI DI, x:word, y:word, string:dword, rotflag:word
        mov ax,rotflag
        mov cx,x
        mov dx,y
        les di,string
        mov ah,0fh
        int 5fh
        ret
    G_Text endp

@curseg ends
end

```

Sample Header File for Use with C Programs.

/* Definitions */

```
#define G_ALPHA          0x07
#define G_GRAPHICS 0x20

#define G_FORCE          0
#define G_AND            1
#define G_OR             2
#define G_XOR            3
#define G_NOTFORCE       4
#define G_NOTAND         5
#define G_NOTOR          6
#define G_NOTXOR         7

#define G_OUTLINE        0
#define G_SOLIDFILL      1
#define G_PATTERNFILL    2

#define MINCOLOR          0
#define MAXCOLOR          1
```

/* Structures */

```
typedef struct g_info {
    unsigned char    vidmode;
    unsigned char    defmode;
    unsigned int     xpixels;
    unsigned int     ypixels;
    int              xloc;
    int              yloc;
    unsigned int     linetype;
    int              rrule;
    unsigned int     color;
    int              xclipmin;
    int              yclipmin;
    int              xclipmax;
    int              yclipmax;
    int              xlong;
    int              ylong;
    unsigned char    fillmask[8];
} G_INFO;
```

/* Graphics library function definitions. (All x,y locations relative to current logical origin unless specified otherwise.) */

```
void far cdecl      G_Mode(int);
G_INFO far * cdecl  G_GetInfo(G_INFO far *);
void far cdecl      G_ColorSel(int);
void far cdecl      G_RepRule(unsigned int);
void far cdecl      G_LineType(unsigned int);
void far cdecl      G_FillMask(unsigned char far *);
void far cdecl      G_LorgA(int, int);
void far cdecl      G_ClipL(int, int, int, int);
void far cdecl      G_Move(int, int);
void far cdecl      G_Point(int, int);
int far cdecl       G_PointRead(int, int);
void far cdecl      G_Draw(int, int);
void far cdecl      G_Rect(int, int, int);
void far cdecl      G_ImageGet(int, int, int, int, char far *);
void far cdecl      G_ImagePut(int, int, char far *, int);
void far cdecl      G_Text(int, int, char far *, int);
```

/* ----- */
****** G_MODE(mode):** mode = {G_TEXT | G_GRAPHICS} changes the display mode to text or graphics. ***/**

****** G_GETINFO(GraphInfoPtr):** GraphInfoPtr is a far pointer to a buffer of the programmer's choosing where the graphics information will be copied. See the typedef for the G_INFO structure in this file for the contents of the buffer. ***/**

/* ----- */
/* ----- */

****** G_COLORSEL(color):** color = 0 or 1 sets the current pen to "color" ***/**

****** G_REPRULE(reprule):** reprule = {G_FORCE | G_AND | G_OR | G_XOR} sets the current replacement rule for all other drawing (except for G_ImagePut, which specifies its own replacement rule). ***/**

****** G_LINETYPE(linetype):** linetype = a 16-bit image that is repeated while drawing lines and G_OUTLINE'd rectangles. Bits that are 1 cause the current pen color to be plotted using the current replacement rule. Bits that are 0 are not plotted and leave the display un-modified. ***/**

****** G_FILLMASK(buffer):** buffer is a far pointer to an 8-byte array which specifies the fillmask to use when doing drawings of rectangles with a fillflag of G_PATTERNFILL. Every bit that is a 1 will cause a point of the current color to be plotted. Every bit that is a 0 will cause that "point" of the display to be undisturbed. ***/**

/* ----- */
****** G_LORGA(x,y):** x,y = -32768 to +32767 sets the logical origin (0,0) to be located at the absolute screen coordinate specified by x,y. ***/**

```

/**** G_CLIPL(x1, y1, x2, y2): x1,y1,x2,y2 = -32768 to +32767 sets the current clip
boundary to the rectangle whose diagonally opposite corners are specified by the absolute
screen coordinates equal to the x1,y1 and x2,y2 offsets from the current logical origin.*/
/* ----- */
/**** G_MOVE(x, y): x,y = -32768 to +32767 causes logical pen to be moved to "x,y" */
/**** G_POINT(x,y): x,y = -32768 to +32767 plots a point of the current pen color with
the current replacement rule at "x,y" unless "x,y" is outside the current clip limits. */
/**** G_POINTREAD(x,y): x,y = -32768 to +32767 reads the color of the point located at
x,y and returns that as the value of the function. */
/**** G_DRAW(x,y): x,y = -32768 to +32767 draws a line of the current pen color with the
current replacement rule and the current linetype from the current pen location to "x,y";
only those points lying within the clip limits are actually plotted. */
/**** G_RECT(x,y,fillflag): x,y = -32768 to +32767 fillflag = {G_OUTLINE |
G_SOLIDFILL | G_PATTERNFILL} draws a rectangle with diagonally opposite corners at
the current pen location and at "x,y", using the current pen color and replacement rule. The
"type" of rectangle drawn is determined by fillflag. If fillflag==G_OUTLINE, the outline of
a rectangle is drawn, using the current linetype. If fillflag==G_SOLIDFILL, a solid, filled
rectangle of the current color is drawn. If fillflag==G_PATTERNFILL, a pattern filled
rectangle of the current color is drawn, using the current fill mask. */
/* ----- */
/**** G_IMAGEGET(x1, y1, x2, y2, buffer): x1,y1,x2,y2 = -32768 to +32767.
If both points x1,y1 and x2,y2 are within the current clip boundary, the display image
bounded by the rectangle whose diagonally opposite corners are x1,y1 and x2,y2 is read into
the bytes pointed to by "buffer".

NOTE: for G_ImageGet(), the required size of 'buffer' is (on The HP 95LX):
8 + ((x2-x1+8)/8) * (y2-y1+1) bytes */
/**** G_IMAGEPUT(x, y, buffer, reprule): x,y = -32768 to +32767 reprule = {G_FORCE
| G_AND | G_OR | G_XOR | G_NOTFORCE | G_NOTAND | G_NOTOR | G_NOTXOR}
If x,y and the un-specified bottom-right corner of the image are within the clip boundary, the
image from "buffer" is drawn on the display using reprule as the replacement rule. (For the
"NOT" replacement rules, the image is color-inverted first, then placed on the display using
the "rest" of the replacement rule.) */
/* ----- */
/**** G_TEXT(x,y,buffer,rotflag): x,y = -32768 to +32767, rotflag = 0 or 1 draws the string
pointed to by "buffer" on the display at the specified location x,y using the current font. If
rotflag = 0, it's drawn horizontally, else it's rotated 90 degrees counterclockwise. */

```


HP 95LX System Manager Operation and Programmer's Guide

Overview

The System Manager is a layer of control and services that resides between the operating system (MS-DOS) and the built-in applications on the HP 95LX. In addition to the built-in applications, the System Manager supports external applications that conform to the programming conventions discussed in this chapter. External applications that run under the System Manager will be called "System-Manager-compliant" applications to differentiate them from programs that run directly under MS-DOS.

A PC version of the System Manager is also used by the Connectivity Pack so that System Manager compliant applications can, usually with no or only minor changes, also run as a component of the Connectivity Pack.

The System Manager provides two basic functions:

1. Application control (includes launching and task swapping).
2. Common services for basic user-interface constructs, file I/O, memory management, and system requests.

This chapter provides general information for developing System-Manager-compliant applications. The information ranges from the general principals of System-Manager operation to a template for a System-Manager-compliant application written in C. See chapter 8 for descriptions of the services available to System-Manager-compliant applications.

System Manager Operation

The System Manager reads all keyboard input so that when an application's hot key is pressed, the System Manager can start that application. When no other applications are active, the System Manager displays the owner-information screen (which can be thought of as the default application).

System Manager Execution

The System Manager is the default shell. That is, it is the program that is started by MS-DOS at boot time and it always remains resident in memory. The System Manager does NOT process AUTOEXEC.BAT on startup.

If desired, the shell can be changed to the standard MS-DOS command processor. This is done by creating a config.sys file containing the command "SHELL=COMMAND /P" and rebooting. If this is done, the system will boot directly into MS-DOS, the System Manager will not be running, and the built-in applications will not be available. The /P parameter causes COMMAND to be a permanent shell and also causes COMMAND to process any AUTOEXEC.BAT file.

After booting directly into DOS, the System Manager can be invoked by the command `$SYSMGR`. There is no way to exit the System Manager, however you can spawn a DOS shell using the Filer's `System` command or by running `COMMAND.COM` from the Filer.

Task Management

First, consider the case where an application is being selected and no applications are currently active (i.e., the owner-information screen is showing).

When the System Manager starts an application, it loads the application into memory, sets up the segment registers, and transfers control to the application's entry point. For built-in applications (that run from ROM), the load step involves allocating enough system RAM for the application's data and copying the initialized data from ROM to RAM. For loading external applications, the System Manager must also allocate system RAM for the application's code. In this case, both the code and initialized data are read from disk into memory.

Once started, an application enters an event loop where it calls the System Manager `m_event` or `m_nevent` function to get the next key stroke or other event. Once started, an application will be called open until it calls the System Manager `m_fini` function, which is normally when the user quits the application. An open application will be called active if it has control and inactive if another application has control.

Now, consider the case where an application is being selected but a different application is currently active. In this case, the System Manager not only needs to load and launch the new application, but also needs to deactivate the previous application.

When switching to a new application, the System Manager sends a deactivate event to the current application, which changes its status from active to inactive, and then starts the new one. If there is insufficient memory to load the new application, the System Manager displays a low memory close down screen and gives the user the chance to terminate one of the open applications. Once there is sufficient memory, the new application is loaded and launched, while the data for the previously open applications remain in memory.

Special handling of the code space is required for external applications. Only one external application's code is kept in memory at any one time, so if the new application is an external application and another external application is open, this code space (after possible expansion) is reused for the code of the new external application.

Note



Whenever there is an external application open, the external application code space is *not* reduced in size because there must always be enough code space to restart any of the open (but inactive) external applications.

Memory Management

The System Manager is responsible for efficiently managing memory for System-Manager-compliant applications.

For example, to prevent memory fragmentation, an inactive application's data space may be moved in memory as other applications are launched and exited. This means that applications should not save the DS value in memory unless the application is prepared to modify the DS value in case its data segment has been moved.

As another example, the code space is overlaid for all external applications. This means that the code for an inactive external application will not be in memory if any other external application has been subsequently activated. This implies that variable data should not be stored in the code segment.

TSRs, Interrupt Vectors, and the System Manager

TSRs are DOS programs that terminate but stay resident in memory. There are two ways that DOS can be accessed on the HP 95LX. One way is to change the DOS shell from \$SYSMGR to COMMAND as mentioned above. The other way is to run a DOS command from the Filer or from the DOS command line accessed via the Filer's System function. Thus, there are two environments in which to run TSRs.

Running TSRs from the Filer has two drawbacks. One is memory fragmentation due to the hole left by COMMAND's data when the TSR terminates. This memory may or may not be usable by the System Manager. A more important drawback is that the System Manager may not be able to run an external application, after a TSR has been installed from the Filer. This is because the System Manager needs to expand a memory block in order to load the external application's code and the TSR may block this expansion.

Consequently, we generally recommend that TSRs be run *before* starting the System Manager. The TSRs can be started from an AUTOEXEC.BAT file which, if terminated with the \$SYSMGR command, will start the System Manager. This technique permanently ties up memory for COMMAND's data; but, since this is below the System Manager, it does not cause memory fragmentation.

TSR's usually make use of some interrupt vectors. The System Manager takes over some interrupt vectors without "chaining" onto the previous owner. This means that a TSR that is loaded first may not get control when it expects to if the System Manager has taken the TSR's interrupt. Thus, it is necessary to know which interrupts are used by the System Manager.

The interrupts taken by the System Manager are:

- Int 05h Print screen.
- Int 06h HP 95LX specific BIOS service used to signal entering and leaving sleep modes.
- Int 0Ch COM1 serial port interrupts.
- Int 1Bh Ctrl-Break interrupt. The System Manager saves the original value and restores it before accessing DOS from the Filer.
- Int 4Ah User alarm. Called by BIOS when real-time clock alarm goes off.
- Int 60h Used for calls to System Manager services.
- Int 61h Used by the System Manager to load its DS register.

In addition, the System Manager chains into Int 1Ch, the user timer tick interrupt.

Finally, Int 62h is used by the HP 95LX's Calculator application.

External Application Support

The System Manager supports adding external applications via entries in an *apname.lst* file. External applications can reside either on the C: drive or on a plug-in card. External applications have the file name extension EXM that distinguishes them from DOS applications that typically have the EXE extension.

The creation of EXM files is discussed in “Building Applications” below.

The total number of external applications, including those on the C: drive and the plug-in card, is limited to eight.

The APNAME.LST File

Each record in *apname.lst* contains registration information on one external application. The format of each record is:

filespec,hotkey,name(carriage return)

where

filespec is the complete drive, path, and file name of the executable file for the application. Note that the *filespec* must not be longer than 28 characters.

hotkey is the four hex digits of the scan/ASCII code for the application's hotkey. Refer to the Int 09h section in chapter 5 for a table of scan/ASCII codes. Char modified keys cannot be used for hot keys.

name is the application name that will be displayed in the System Manager low memory close out screen. *name* can be up to 12 characters long.

Applications on the C: Drive

At startup, the System Manager checks for the presence of the file C:_DAT\APNAME.LST. If the file exists, the applications listed in it are added to the System Manager task table as external applications.

This provides the opportunity to register applications that reside on the C: drive. All entries in C:_DAT\APNAME.LST should start with C:.

Applications on a Plug-in Card

The System Manager contains support for automatic registration of external applications that reside on a plug-in card. The general situation is that when a card is inserted, the System Manager checks for the presence of the file A:\APNAME.LST. If the file is found, its entries are added to the System Manager task table as (possibly additional) external applications. Likewise, when the card is removed, the entries are removed from the task table. However, there are special situations that can occur if a plug-in card application is “missing” due to its card being removed while the application was open.

Suppose a card containing an open external application is removed. In this case, the System Manager will not remove the card's applications from the task table and will refuse to register any applications from other cards that might be inserted.

Should reloading the missing application's code become necessary, an error condition exists since the code cannot be found. Reloading would be necessary, for example, if a second external application residing on the C: drive is run and then the missing application's hot key is pressed.

A distinctive beep is issued to signify various "missing application" conditions as follows:

- When any card is inserted. If this card is the missing application's card, then that application is again available for use. If this is a different card, then the beep is a warning that there is an open application from a previous card and that any applications that might be on this card have not been registered.
- When the hot key for a missing application is pressed.
- When other applications are exited causing the missing application to become the current application. In this case, the missing application will be skipped, allowing the next application on the stack to be restarted. In addition, the missing application will be hidden on the active task list so that it will be skipped in the future.
- When the application is selected from the low-memory close down menu. In this case, the application will remain on the menu, but it cannot be closed down until its card is plugged in again.

Access to Services

Accessing Services from the C Programming Language

C-language applications access services by calling the functions as listed in chapter 8.

Each source file containing services calls must include the header file `interfac.h` which contains macro definitions for each service. The macro expands the call to be a call to a common System Service request function. In addition, the macro adds a service number to the argument list and casts near pointers to far pointers as appropriate.

For example, the `m_disp` call in the source

```
#include "interfac.h"

int row,col,style,ostyle;
char *str;
...
m_disp(x,y,str,strlen(str),style,ostyle);
```

expands to

```
c_service(F_M_DISP,x,y,(void far *)str,strlen(str),style,ostyle);
```

where

`F_M_DISP` is the function code for `m_disp` and is defined in `interfac.h`.

The function `c_service` is provided in the object module `csvc.obj` and converts the call into a software interrupt that transfers control to the System Manager dispatch table.

Accessing Services from Assembler

Assembler programs access the services by pushing any required arguments on the stack and then using the `SMCALL` macro to "call" the service. The required arguments can be found in chapter 8.

For example, usage of `m_disp` might appear as:

```
include interfac.mac
...
push  dx      ; ostyle (not actually used)
push  dx      ; style
push  cx      ; string length
push  ds      ; string segment
push  si      ; string offset
push  ax      ; column
push  bx      ; row

SMCALL F_M_DISP ; display the string

add  sp,14    ; remove arguments from stack
```

There are three things to note in this example:

- The arguments are pushed in left to right order as is done by the C compiler.
- Pointers are passed as far pointers. See `interfac.h` for argument specifics where, in general, pointers are cast as far pointers.
- The function numbers such as `F_M_DISP` and the `SMCALL` macro can be found in `interfac.mac`.

Application Considerations

RAM Versus XIP Execution

RAM execution refers to having the application code load into RAM at run time. XIP (eXecute In Place) refers to having the application code run directly from ROM. All the built-in applications are XIP while external applications can be either strictly RAM or a combination of RAM and XIP.

The System Manager does not directly support external XIP applications. That is, the System Manager loads both the code and data for EXM files into RAM. However, an EXM program can in turn use the XIP services provided in Int 63h to launch an XIP program. In this scenario, the EXM program acts as a loader for the XIP portion of the program.

Since the XIP loader is a small amount of code, an application done as XIP will require less RAM for execution than if it was done as RAM-executable. However, the card containing an XIP application must not be removed while that application is active. Doing so will force a system warm start.

Application Initialization and Termination

All System Manager applications must call the System Manager functions `m_init` and `m_fini` on startup and termination, respectively.

Note `m_fini` never returns and hence plays a role analogous to the DOS terminate process function.



Event Handling

The System Manager implements non-preemptive multitasking for System-Manager-compliant applications. Hence, compliant applications are event driven and must make timely calls to the `m_event` or `m_nevent` function to receive keyboard input and other events.

The System Manager reports key strokes for normal keys, but reports a deactivate event when another application's hot key is pressed. In response to a deactivate event, the application is expected to do any necessary housekeeping to prepare for suspension and then request the next event. The next event, which will be an activate event, will not be returned until this application is activated again. Activation can occur for several reasons; for example, the application's hot key is pressed, all subsequently activated applications are quit, or the application has been selected for termination from the low memory close out menu.

In response to an activate event, the application must redraw its screen. For RAM efficiency, it is recommended that applications have a means of redisplaying the screen from primary data, rather than by saving a copy of display memory.

The System Manager may also return a termination event at any time. The application is required to respond to this as if the user issued a quit command, with any user-interface variations needed to make clear what is happening. For example, if an editor is being terminated and its buffer has been modified, the user should be prompted to save the file.

If possible, when an application terminates, it should save its state information in a file. When it is subsequently relaunched, it can inspect the file and reconstruct its state prior to termination.

The example at the end of this chapter shows the code for a typical event loop.

Interruptible Processes

An “interruptible process” within a program is one which the user can interrupt by pressing a key. An example of an interruptible process is function plotting in the built-in calculator application.

During an interruptible process, the program must periodically check the keyboard to see if the user has requested interruption. The HP 95LX BIOS uses keyboard checks as an occasion to go into light sleep. This is done as a power saving feature and is only done when running on batteries. In many cases keyboard checks are only done when a program is otherwise idle. In these cases going into light sleep does not affect performance and is an effective way to conserve batteries. However, during an interruptible process, going to light sleep can substantially slow down the process.

The BIOS provides a service (Int 15h, function 4Eh) which controls whether light sleep will be entered during a key test. This function should be used to disable going to light sleep during an interruptible process and then to reenale going to light sleep after the process is complete. Care must be taken to be sure and reenale light sleep for power conservation reasons. The code fragment below shows an interruptible process which can be terminated by the ESC key or by Cntl-Break. Included are routines which disable and enable going to light sleep during a key press.

```

/*-----*/

m_lock();    /* disable task swapping and thus ensure that we
               cannot exit without getting a chance to reenale
               light sleep */
disable_light_sleep();
while (1) {
    /* one iteration of interruptible process goes here */
    ...
    /* check for key */
    m_nevent(&appevent);
    if (appevent.kind == E_BREAK)
        break;
    if (appevent.kind == E_KEY) {
        m_event(&appevent);
        if (appevent.data == ESCkey)
            break;
        else
            m_beep();
    }
}
enable_light_sleep();
m_unlock(); /* reenale task swapping */

/*-----*/
void disable_light_sleep(void)
{
    _asm {
        mov     ax,4e00h
        int     15h
    }
}

/*-----*/
void enable_light_sleep(void)
{
    _asm {
        mov     ax,4e01h
        int     15h
    }
}

/*-----*/

```


Using Standard C Library Functions

Our general recommendation is: If the System Manager provides a given service, that service should be used instead of using a C-library function. This ensures compatability and reduces the application's code size.

Some specific points to note are:

- Keyboard input must be obtained only by the use of the System Manager `m_event` or `m_nevent` functions.
- Dynamic memory allocations must be done only by the use of the System Manager `m_alloc` or `m_alloc_large` functions, or by using DOS services directly. Standard C library memory management functions should not be used.
- Elementary C Library routines such as `strlen` and `atoi` can be used.
- System Manager file I/O uses a `FILE` structure that is not compatible with the `FILE` structure defined in Microsoft C `stdio.h`. Use care with including `stdio.h` and, in particular, don't include both `fileio.h` and `stdio.h` in the same module.
- The standard C library startup code is not used with System-Manager-compliant applications. This may affect the use of some types of C Library functions.

Using DOS and BIOS Services Directly

Our general recommendation is: If the System Manager provides a given service, it should be used instead of going directly to a DOS or BIOS function. This insures compatibility and also reduces the application's code size. In particular, keyboard input must be obtained only by the use of the System Manager `m_event` or `m_nevent` functions.

Memory Model Conventions

External System-Manager-compliant applications must be small model programs.

Specifically,

- Less than 64 KB of code.
- Less than 64 KB of preallocated data including the stack.

If larger code is required, the program *must* be made Execute-In-Place (XIP).

Additional data space, up to available memory, can be dynamically allocated using the System Manager `m_alloc` or `m_alloc_large` functions.

Data Pointer Considerations

As mentioned previously, the System Manager may move an application's data space under certain circumstances and in particular when an application is inactive. If only DS relative NEAR pointers are used, then this is not a problem since the System Manager will set up DS properly prior to activating the application. If FAR data pointers are used, then the application needs to fix these pointers each time the System Manager moves its data segment.

Startup Considerations

The special C-language program-startup situations for System-Manager-compliant applications are:

- Applications must link with a special version of the C-language startup code, `crt0.obj`. This version is needed because the System Manager launch mechanism has already performed the tasks performed by the normal C-language startup code.
- The command-line and environment variables, `argc`, `argv`, and `envp`, are *not* available.

Compiling and Linking Conventions

Three rules must be followed when compiling and linking System-Manager-compliant applications:

1. Applications must be compiled with the `/Gs` option to eliminate stack checking.
2. There must be at least 256 bytes of application stack space available for System Manager use whenever a System Manager service is called.
3. Applications must link with the System Manager services interface module, `csvc.obj`. This module contains the `c_service` function that performs the software interrupt to transfer control to the System Manager services jump table.

Building Applications

Development Tools

Compilers, Assemblers, and Linkers

We recommend that System-Manager-compliant applications be built using Microsoft's C 5.1 (or later version) compiler or the Microsoft 5.1 (or later version) assembler. The recommended linker is the Microsoft linker which is compatible with your compiler and assembler. The output from these tools is an EXE file.

EXE to EXM Utility

A utility program, E2M, is provided to convert the EXE file produced by the linker to an EXM program that can be loaded by the System Manager. See the example below.

Checkout Using Tkernel

Tkernel is a DOS TSR version of the System Manager that runs on a standard PC. While there are limitations, extensive development can be done on many applications using tkernel. When applicable, tkernel provides the fastest checkout and debugging environment for System Manager compliant applications.

Tkernel takes over Int 60h and processes System-Manager-function calls in a manner very similar to that on an actual HP 95LX. The HP 95LX screen will be simulated by a 40 character by 16-line region in the center of the PC screen.

Some special aspects of using tkernel are:

- When using tkernel, your application runs as a standard DOS application and a standard debugger can usually be used.
- Task switching and other interactions among applications cannot be tested because only one task runs at a time under tkernel.
- Since the HP 95LX's MENU key is not on a PC keyboard, some other key must be used. It is suggested that your application respond to both the F11 key (scan/ASCII code = 0x8500) and the MENU key in the same way. This enables the F11 key to be used as the MENU key under tkernel. To enable this to work on PC's which don't have an F11 key, tkernel converts Alt-F10 to the F11 key code so either F11 or Alt-F10 can be used as the MENU key.

Graphics Checkout Using TSRGraph

In a manner analogous to tkernel, TSRgraph.com is a DOS TSR which provides access to the HP 95LX's graphics routines on a PC. TSRgraph can be installed either before or after tkernel. Successive executions of TSRgraph switch between installing it and deinstalling it.

Example

Simple System Manager Program

The following listing is a template for a System-Manager-compliant application.

```
/*
 * SMFELLO.C - Small example of a System Manager compliant program.
 */

#define TRUE 1
#define FALSE 0

#include "..\headers\interfac.h"
#include "..\headers\event.h"

/* function prototypes */

void app_init(void);
void app_term(void);
void app_awake(void);
void app_sleep(void);
void app_break(void);
int app_key(void);
void app_display(char *msg);

/* global variables */
EVENT appevent;

/*-----*/
void main(void)
{
    int done = FALSE;

    m_init(); /* init call to system manager */
    app_init(); /* application initialization */

    /* event loop */
    do {
        m_event(&appevent); /* get next event */
        switch (appevent.kind) {
            case E_ACTIV:
                app_awake(); /* reactivate app */
                break;
            case E_DEACT:
                app_sleep(); /* prepare for suspension */
                break;
            case E_TERM:
                done = TRUE; /* being terminated */
                break;
        }
    } while (!done);
}
```

```

        case E_BREAK:
            app_break();          /* app ctrl-break handler */
            break;
        case E_KEY:
            done = app_key();      /* process key */
            break;
    }
}
while (!done);

app_term();          /* application termination */
m_fini();             /* terminate call to system manager, never returns */
}
/*-----*/
void app_init(void)
/*
 * Initialize application
 */
{
    app_display("app_init");
}
/*-----*/
void app_term(void)
/*
 * Terminate application
 */
{
}
/*-----*/
void app_awake(void)
/*
 * Reactivates application after suspension.
 */
{
    app_display("app_awake");
}
/*-----*/
void app_display(char *msg)
{
    char usage_str[] = "press q to exit ...";

    m_setmode(1);      /* set text mode */
    drawbox("SM Hello");

    m_disp(1,5,msg,strlen(msg),0,0);
    m_disp(3,5,usage_str,strlen(usage_str),1,0);
}
/*-----*/

```

```

void app_sleep(void)
/*
 * Prepares application for suspension.
 */
{
}
/*-----*/
void app_break(void)
/*
 * Application control-break handler.
 */
{
}
/*-----*/
int app_key(void)
/*
 * Application keystroke processor.
 *
 * Returns TRUE if user has requested termination, else returns FALSE.
 */
{
    if (appevent.data == 'q') {
        return TRUE;
    }
    else {
        m_beep();           /* signal error */
        return FALSE;
    }
}
/*-----*/

```

Make File for Building EXE and EXM Files

Below is an `nmake` make file that will build both a `tkernel` and an HP 95LX version of the example program. The `tkernel` version is called `smhello.exe` and the HP 95LX version is called `smhello.exm`.

This file as well as object modules `csvc.obj` and `crt0.obj` are provided in the developer's kit.

```
all: smhello.exe smhello.exm

# HP 95LX version

smhello.exm: jsmhello.exe
    ..\tools\e2m jsmhello
    copy jsmhello.exm smhello.exm
    del jsmhello.exm

jsmhello.exe: jsmhello.obj
    link @<<jsmhello.lnk
/M /NOE /NOI jsmhello.obj+..\tools\csvc.obj+..\tools\crt0.obj
jsmhello.exe
jsmhello.map;
<<NOKEEP

jsmhello.obj: smhello.c
    cl /c /AS /Gs /Fojsmhello.obj smhello.c

# tkernel version

smhello.exe: smhello.obj
    link @<<smhello.lnk
/M /NOE /NOI smhello.obj+..\tools\csvc.obj
smhello.exe
smhello.map;
<<NOKEEP

smhello.obj: smhello.c
    cl /c /AS /Fosmhello.obj smhello.c
```

Running SMHELLO Under Tkernel

To run SMHELLO under `tkernel`, use the commands

```
tkernel
smhello
```

If `tkernel` is executed when `tkernel` is already loaded, you are asked if you want to uninstall it.

If desired, the `smhello` command can be replaced with a command which runs `smhello.exe` from a debugger.

Running SMHELLO on the HP 95LX

Download `smhello.exm` to the root directory of the C: drive on the HP 95LX and using Memo, create or modify the file `C:_DAT\APNAME.LST` to include the line `C:\SMHELLO.EXM,C300,SMHello`

Now reboot the system.

At this point, the hot key Alt-CALC is assigned to the SMHELLO program.

HP 95LX System Manager Services Reference

This chapter describes the HP 95LX System-Manager services that are available to any System-Manager-compliant application.

Overview

The HP 95LX System-Manager Services provide a simple, memory-effective means by which all HP 95LX System-Manager-compliant applications can share a set of library functions. These services simplify development of HP 95LX applications and maximize memory efficiency.

Because System-Manager Services are inherently viewed as a set of subroutine calls made by applications, the services described herein are specified as 'C' Language function calls. Services are grouped by functionality, and each group has a brief operational overview, description of any data structures required by the group, and specifications for each function. Individual function descriptions include parameter conventions, return values, and a functional synopsis.

The functional areas are:

- Event Services.
- Menu Services.
- File Menu Services.
- Screen Services.
- Editing Services.
- File Services.
- Process Management Services.
- Clipboard Services.
- Sound Services.
- Memory Management Services.
- Date/Time Services.
- Printer Services.
- Configuration Services.
- Communications Services.
- Miscellaneous Services.
- Resource Services.
- Help Services.
- Collating Services.
- 1-2-3 Bridge Services.

These services are described in the following sections of this chapter.

Header Files

Header files are used for macro and structure definitions. All functional areas require the inclusion of `interface.h` for C modules or `interfac.mac` for assembler modules.

Some functional areas also require the inclusion of additional header files as follows:

Service Class	C Program	Assembler Program
Event	event.h	event.mac
Menu	menu.h	menu.mac
File Menu	fmenu.h	fmenu.mac
Editing	edit.h	smedit.mac
Clipboard	cbcodes.h	cbcodes.mac
File I/O	fileio.h	filio.mac
Date/Time	smtime.h	smtime.mac
Configuration	settings.h	settings.mac
Communication	comio.h	comio.mac
Bridge	bridge.h	bridge.mac
Misc	m_error.h	m_error.mac

A Note on C-Language Function Prototypes

The header files do not contain function prototypes for the services. Instead, `interfac.h` contains macro definitions that expand System-Manager-service calls into calls to a single service-dispatch routine.

Hence, if a service is accidentally called with the incorrect number of parameters, it will not match the macro and the compiler will issue a warning about too few or not enough actual parameters for the macro.

However, if a service is accidentally called with an incorrect argument type but with the correct number of parameters, this may not be caught by the compiler due to the lack of a function prototype.

The following functions in `interfac.h` are not usable by applications:

<code>com_timer_addr</code>	<code>com_timer_count_addr</code>
<code>InitCom</code>	<code>m_appcount</code>
<code>m_app_name</code>	<code>m_common_open</code>
<code>m_day_trigger</code>	<code>m_disable_macros</code>
<code>m_enable_macros</code>	<code>m_fall_printer</code>
<code>m_get_settings</code>	<code>m_ram_iv_info</code>
<code>m_reboot</code>	<code>m_set_daterule</code>
<code>m_set_settings</code>	<code>m_spawnarg</code>
<code>m_sys_rsrc_addr</code>	

A Note on Far Versus Near Pointers in Service Calls from C

All pointers in System-Manager-service calls are *far* pointers. However, the service macros contain casts of pointer arguments to far pointers. Consequently, near pointers can be used for data in cases where the compiler can supply the segment value due to the cast.

Event Services

While active, applications should poll the System Manager's event functions to get input from the user, and dispatch according to the type of event reported and the current state of the application (doing a menu, editing text, etc.). Keystrokes are translated for both applications (CP 850) and 123 (LICS and function key flags). Event information is passed through the structure of type `EVENT`, which is defined below.

```
typedef struct {
    enum event_kind kind;      /* event kind code defined below */
    unsigned int data;         /* For ASCII keys, this is just the ASCII
                               code in the low byte. For non-ASCII keys,
                               the scan code is in the high byte and the
                               low byte is zero. */
    unsigned char scan;        /* scan code from BIOS */
    unsigned char shifts;      /* shifts register, when function returns */
                               /* not necessarily when key struck */
    unsigned int lics;         /* LICS translation of keystroke */
    unsigned char fkey_num;    /* function key number for 123 only */
    void far *bridge;         /* pointer to 123 bridge data structure */
} EVENT;
```

The meanings of the `event_kind` codes are:

Code	Meaning
<code>E_ACTIV</code>	Application receiving the event has just been activated or reactivated. This should be taken as a signal to (re)display in the active state.
<code>E_ALARM_DAY</code>	The application's daily chance to set an alarm.
<code>E_ALARM_EXP</code>	The application's alarm has expired.
<code>E_BREAK</code>	Control-Break detected. shifts reports current stat of the keyboard shift flags.
<code>E_BRIDGE</code>	Event reported only to 123 when a bridge service is requested. Pointer to data is found in bridge field.
<code>E_DEACT</code>	Application receiving this event is about to be deactivated. It is given the opportunity to prepare for an inactive state that will begin following the next <code>m_event</code> call.
<code>E_GROW</code>	A request to 123 to grow.
<code>E_KEY</code>	Keystroke available to application. data contains the CP 850 ASCII value. scan contains the keyboard scan code. lics contains the LICS interpretation of the keystroke. fkey_num contains the function key code used by 123. shifts reports the current state of keyboard shift flags, not necessarily the state of the shifts when key was struck.

Code	Meaning
E_NONE	No event available. shifts reports current state of keyboard shift flags (immediately before control is returned).
E_SHRINK	A request to 123 to shrink.
E_TERM	Application receiving the event is about to be terminated. It should respond to this event as if {MENU,Quit} had been entered from the top level. An application may interact with the user and make calls to <code>m_event</code> as necessary. It should ultimately make a call to <code>m_fini()</code> to give up control, or to <code>m_no_fini()</code> if the user has indicated a desire to abort the shutdown process.
E_TIMECHANGE	Indicates that the system date or time has been changed.

Applications using the event services must include the header files `event.h` and `interfac.h`.

m_event

```
void m_event(eventptr);
EVENT far *eventptr;
```

Transfers control to the System Manager until a reportable event has occurred or until a timeout period of approximately 0.5 seconds has elapsed. The event is reported in `*eventptr` and is taken out of the system queue.

In the event of a timeout, `E_NONE` will be reported. This gives applications the opportunity to redisplay the current time.

m_flush_kb

```
void m_flush_kb();
```

Flushes the keyboard queue. Control is returned when the queue is empty.

m_nevent

```
void m_nevent(eventptr);
EVENT far *eventptr;
```

Transfers control to the System Manager. If no event has occurred, `E_NONE` will be reported. The event is reported in `*eventptr`.

If the event is an `E_KEY` event, the event is not removed from the system queue. However, all other events, such as activation and deactivation events are only reported once, despite the fact that `m_nevent()` has been called.

m_no_fini

```
void m_no_fini(eventptr);  
EVENT far *eventptr;
```

If, in response to an E_TERM event, an application discerns that the user wishes to abort the shutdown procedure, it should call m_no_fini(). This call will release the system locks (allowing context switching) and break the shutdown sequence. The application making the call will continue to be the active application.

An event structure must be passed, but its fields will be undefined on return, and may be safely ignored.

m_sh_status

```
int m_sh_status();
```

Gets the current status of the keyboard shift flags. Return value is same as value returned by BIOS (Int 16h, Service 2).

m_yield

```
void m_yield(eventptr);  
EVENT far *eventptr;
```

Voluntary suspension of an application. When called, the application is placed at the end of the application stack, and the active state reverts to the next application. Control will not return to the caller of m_yield until some external action causes it to be made active again (generally, user request via hotkey).

The return value will be the same as if the application had made an m_event() call and had been suspended; that is, the expected event type on return is E_ACTIV.

Menu Services

All applications should use the menuing system provided by the System Manager. The menu system uses the 1-2-3-styled two-line menu bar (where the top line contains keywords to all of the options available at a particular decision branch and the second line displays a long message corresponding to the currently highlighted option). Because of the screen space constraints, the two lines are used for keywords, and no long prompts are supported.

There is an option which supports a prompted menu, in which the top line is a constant prompt (it does not change as the item selection changes) and the second line is filled with keywords.

Menus are used by calling `menu_setup()` to fill a data structure of type `MENUDATA`. `menu_on()` should be called when the menu becomes active; `menu_off()` when menu selection has been made. The application should repeatedly call `menu_key()` and `menu_dis()` while keystrokes are available or until the user has selected an item or aborted the process. Note that process events (i.e., non-keystrokes) should be handled without calls to `menu_dis()`.

The definition for the structure `MENUDATA` is:

```
typedef struct {
    /* define the menu display storage area */
    /* the intent is to have one string per display line to make display
       management easier */
    char menu_text [MAX_MENU][MAX_MWID]; /* menu display storage area */
    int menu_count; /* number of keywords */
    int menu_highlight; /* index of highlighted item */
    /* -1 for no highlight */
    /* flag indicating special mode with single prompt on top line */
    int menu_tprompt; /* 0 ==> no */
    /* define the menu information table */
    char menu_line [MAX_KWDS]; /* which line of menu this word is on */
    char menu_offset [MAX_KWDS]; /* offset of this keyword in the string */
    char menu_length [MAX_KWDS]; /* length of this keyword */
    char menu_letter [MAX_KWDS]; /* first letter of this keyword */
    /* store the pointers to the long prompts */
    unsigned menu_prompt[MAX_KWDS]; /* DS-relative offsets... */

} MENUDATA;
```

Applications using the menu services must include the header files `menu.h` and `interfac.h`.

`menu_dis`

```
void menu_dis(m);
MENUDATA far *m;
```

Displays the menu, when appropriate, with any necessary highlighting.

menu_key

```
int menu_key(m, keystroke, presult);  
MENUDATA far *m;  
int keystroke;  
int far *presult;
```

Given the *keystroke*, updates the menu display and returns the index of the selected item (if any). The routine determines the appropriate action for any arbitrary keystroke. *keystroke* should be the value returned in the data field of the event structure.

Sets **presult* to -1 if no final selection has been made, otherwise to the index of the selected item.

Returns 0.

menu_off

```
void menu_off(m);  
MENUDATA far *m;
```

Deactivates menu management, removes highlight.

menu_on

```
void menu_on(m);  
MENUDATA far *m;
```

Activates menu management and highlights first item in list.

menu_setup

```
int menu_setup(m, keywords, keyword_count, double_space, top_prompt,  
               tprompt_len, long_prompts)  
MENUDATA far *m;  
char far *keywords;  
int keyword_count;  
int double_space;  
char far *top_prompt;  
int tprompt_len;  
char far *long_prompts;
```

Builds a menu structure to be used later.

m points to a structure of type MENUDATA to be filled in by the procedure. *keywords* points to a series of null-terminated strings which make up the keywords of the menu. The first character of the second word directly follows the null terminator of the first, and so on. *keyword_count* indicates the number of keywords present. *double_space* indicates whether one (value of 0) or two (value of 1) spaces will be displayed between menu items. The spacing argument is taken as a recommendation: single spacing will be used if it keeps all items on the top line or if the second line is filled.

For normal menus, **top_prompt* will be NULL, and *tprompt_len* will be 0. In that case, *long_prompts* points to an array of pointers (all relative to the constant resource segment) to the long prompts for the menu. There must be a one-to-one correspondence between long prompts and keywords.

Note

Long prompts are not supported and long-prompts will be ignored on the HP 95LX.

Special menus with top-line prompts are created by setting **top_prompt* and *tprompt_len*.
Returns 0.

File Menu Services

File-Menu services provide a flexible method for file name selection. The services can display a list of file names from which the user can make a selection by moving a cursor to the desired name and pressing **ENTER**.

The list of file names can be selected by means of a wild card. A common usage is to display all the file names that have the application-specific extension.

The services make use of three structures: FMENU, FILEINFO, and EDITDATA. FMENU and FILEINFO are described below; EDITDATA is described in the Editing Services section.

The FMENU structure is:

```
typedef struct {

    /*---Members to be Initialized by the Application---*/
    char far *fm_path;           /* base directory name C:\DATA\ */
    char far *fm_pattern;        /* file pattern, e.g. *.WK1 */
    FILEINFO far *fm_buffer;     /* workspace for file list (hold infos) */
    int fm_buf_size;             /* size of the buffer in bytes */
    int fm_startline, fm_startcol; /* starting row,col */
    int fm_numlines, fm_numcols; /* number of lines and columns */
    int fm_filesperline;         /* number of files displayed across */

    /*---Members thst are initialized by the File Menu Services---*/

    int fm_firstedit;            /* 0 if first edit char, else multi line */
    int fm_filesinbuf;           /* number of files in list */
    int fm_maxfinbuf;            /* maximum number of files buffer holds */
    int fm_topfile;              /* file at top of list */
    int fm_curselect;            /* index of the file highlight */
    int fm_oldselect;            /* index of file to un-highlight */
    int fm_focus;                /* 1 = fmenu, 2 = edit */
} FMENU;
```

As indicated by the comments, some elements of this structure are to be initialized by the application. This initialization must occur prior to the call to `fmenu_init`. These elements are:

<code>fm_path</code>	Pointer to null-terminated name of directory that is to be searched for files. For example, <code>C:_DAT</code> .
<code>fm_pattern</code>	Pointer to null-terminated string that contains a file name or a wild-card-file pattern. For example, <code>*.WK1</code> .
<code>fm_buffer</code>	Pointer to an array of <code>FILEINFO</code> structures (see below) that will be used to hold the file names that match the file pattern. The number of elements in this array is the total number of files that can be displayed by the File-Menu Services. Thus, choosing the size of this array is a tradeoff between memory usage for this array and how many files are supported.
<code>fm_buf_size</code>	Size in bytes of file-name list pointed to by <code>fm_buffer</code> . If <code>NFILES</code> is the number of elements in <code>fm_buffer</code> , <code>fm_buf_size</code> is <code>NFILES * sizeof(FILEINFO)</code> .
<code>fm_startline</code>	Number of lines on the display where the file-menu prompt will appear. The file list will start on the next line down. As with the Screen Services, the top line is the number <code>-3</code> . A common value is <code>-2</code> which leaves the top line free for a prompt such as "File to open."
<code>fm_startcol</code>	Leftmost column to be used by the file-menu display. A common value is <code>0</code> .
<code>fm_numlines</code>	Number of lines to be used by the file-menu display, including the file-menu-prompt line. A common value is <code>13</code> , which, together with the top line for a general prompt and the bottom two lines for softkey labels will use all lines of the display.
<code>fm_muncols</code>	Number of columns to be used by the file-menu display. A common value is <code>40</code> which uses all columns of the display.
<code>fm_filesperline</code>	Number of file names to be displayed on each line across the screen. A common value is <code>3</code> given that <code>fm_numcols</code> is <code>40</code> .

In addition, a common usage employs a file-selection prompt identifying the file to be selected (for example, "File to open:"). This prompt is inserted in the `EDITDATA` structure prior to calling `fmenu_init`.

There are four EDITDATA structure elements to be initialized as follows:

<code>prompt_window</code>	Should be set to 1.
<code>prompt_line_length</code>	Should be set to 0.
<code>message_line</code>	Should point to the desired prompt string.
<code>message_line_length</code>	The length of the prompt string.

The FILEINFO structure is:

```
typedef struct {
    char fi_attr;           /* file attribute */
    int fi_time;            /* time modified */
    int fi_date;            /* date modified */
    long fi_size;           /* file length */
    char fi_name[13]        /* file name */
} FILEINFO;
```

The FILEINFO structure is used internally by the File-Menu Services and is not intended to be manipulated directly by the application.

Applications using the File-Menu Services *must* include the header files `fmenu.h`, `edit.h`, and `interfac.h`.

fmenu_dis

```
int fmenu_dis(fmenu_data, edit_data);
FMENU far *fmenu_data;
EDITDATA far *edit_data;
```

Redisplays the current file-menu screen. *fmenu_data* and *edit_data* are pointers to the same FMENU and EDITDATA structures that were used in the call to `fmenu_init`.

fmenu_init

```
int fmenu_init(fmenu_data, edit_data, name, namelen, maxlen);
FMENU far *fmenu_data;
EDITDATA far *name;
char far *name;
int namelen;
int maxlen;
```

Initializes the File-Selection-Menu Services and displays the File-Selection menu. *fmenu_data* points to an FMENU structure that has been initialized by the application as shown above. *edit_data* points to an EDITDATA structure that is normally initialized with a file-selection prompt as described above. *name* points to a null-terminated string that can be a file name, a pattern, or a null string. If *name* is a file name or pattern, `fmenu_init` will present this name on the prompt line and will not display the names of files that match `fm_pattern`. If *name* is a null string, `fmenu_init` displays `fm_pattern` on the prompt line and displays the names of files that match this pattern. *namelen* is the length of the *name* string.

The *mazlen* must be present, but is not used. We recommend that you set *mazlen* to zero.

Returns RET_OK normally; else returns RET_BADFILE, RET_BADDIR, or RET_BADDRIVE if there is a problem with *name*. If such an error results, a common recourse is to use the null string for *name*. This causes the File-Menu Services to use the file pattern contained in *fm_pattern*, which should specify an existing directory.

fmenu_key

```
int fmenu_key(fmenu_data,edit_data,key);
FMENU far *fmenu_data;
EDITDATA far *edit_data;
int key;
```

Processes a keystroke that is entered while the File-Menu Service is active. *fmenu_data* and *edit_data* are pointers to the same FMENU and EDITDATA structures that were used in the call to *fmenu_init*.

key is the value of the keystroke as returned in *event.data* by the Event Services.

The return values defined in *fmenu.h* are:

RET_UNKNOWN	key was unknown by <i>fmenu_key</i> .
RET_OK	key was processed by <i>fmenu_key</i> , just call <i>fmenu_dis</i> .
RET_BAD	key was known by <i>fmenu_key</i> , but was invalid (for example, pgdn off list).
RET_REDISPLAY	redisplay application area before calling <i>fmenu_dis</i> .
RET_ACCEPT	user made a file selection, the filename is in the <i>edit_buffer</i> element of the EDITDATA structure pointed to by <i>edit_data</i> .
RET_ABORT	user aborted operation.

fmenu_off

```
int fmenu_off(fmenu_data,edit_data);
FMENU far *fmenu_data;
EDITDATA far *edit_data;
```

Clears the portions of the screen that were used by the File-Menu Services. *fmenu_data* and *edit_data* are pointers to the same FMENU and EDITDATA structures that were used in the call to *fmenu_init*.

Returns RET_OK.

Screen Services

Accessing the user display is done through the following functions. All row and column values given relative to the top-left corner of the application's active window; that is, (0,0) are the coordinates of the first character that may be drawn in the applications window.

In the HP 95LX, position (0,0) corresponds to the first column in the fourth row. Negative row coordinates can be used to access the first three rows of the HP 95LX screen; for example, the top row is -3.

Applications using the screen services must include the header file **interfac.h**.

m_chrattr

```
m_chrattr(buffer, len);  
char far *buffer;  
int len;
```

Returns the characters and attributes starting from the top left corner of the screen. *len* character/attribute pairs are put into *buffer*.

m_chrinvt

```
m_chrinvt(row, col, nchars);  
int row, col, nchars;
```

Invert the attributes for *nchars*, starting at the screen location specified by *row* and *col*.

m_chrrvrt

```
m_chrrvrt(row, col, buf, len);  
int row, col, len;  
char far *buf;
```

Restore attributes on display. *buf* is a pointer to an array of characters, which are the saved attributes to be restored. *len* tells how many characters are to be restored. *row* and *col* tell which characters to restore.

m_clear

```
m_clear(row, col, nrows, ncols);  
int row, col, nrows, ncols;
```

Clears a rectangular region where *row* and *col* specify the top-left corner and *nrows* and *ncols* specify the dimensions.

m_dirty_sync

```
void m_dirty_sync(void);
```

Causes an immediately redisplay of the virtual display buffer. This function is called automatically at the beginning of `m_event` (and `m_nevent`).

m_disp

```
m_disp(row,col,str,len,style,ostyle);  
int row,col,len,style,ostyle;  
char far *str;
```

Displays *str* of length *len* at *row* and *col*, with the attribute *style*. The *ostyle* parameter is required, but is not used. The valid values for *style* are:

Style	MDA Attribute	Description
0	07h	Normal
1	70h	Inverse
4	01h	Underlined
8	87h	Normal Blink
9	F0h	Inverse and Blink
12	81h	Underline and Blink

`m_disp` puts the string into a virtual display buffer and does not write the string to the physical display. The changes to the virtual display buffer are written to the physical display each time the application requests the next event. The `m_dirty_sync` function should be used if it is desired to update the physical display immediately after an `m_disp` call (as might be the case during debugging).

m_getmode

```
int m_getmode(void);
```

Returns the current display mode, 1=text or 2=graphics.

m_rows_cols

```
int m_rows_cols();
```

Returns number of rows in high byte, number of columns in low byte.

m_scroll

```
m_scroll(row,col,nrows,ncols,offset);  
int row,col,nrows,ncols,offset;
```

Scrolls vertically the rectangular region where *row* and *col* specify the top-left corner, *nrows* and *ncols* specify the dimensions, and *offset* is the number of lines to scroll.

m_setcur

```
void m_setcur(row,col);  
int row,col;
```

Sets the cursor position to the row and column coordinates specified by *row* and *col*. The cursor can be turned off by moving it to an off-screen location; for example, use `m_setcur(0,-1)`.

m_setmode

```
void m_setmode(mode);  
int mode;
```

Sets the display mode to text (*mode*=1) or graphics (*mode*=2) and clears the display.

m_xchg

```
m_xchg(row,col,nrows,ncols,region);  
int row,col,nrows,ncols;  
char far *region;
```

Copies the information in *region* into the rectangular region on the screen whose top-left coordinates are *row* and *col*, and whose dimensions are *nrows* and *ncols*.

Editing Services

The System Manager provides general purpose editing facilities that should be used by all applications. Single-line editing is done by filling a structure of type EDITDATA by calling EDIT_INI() or EDIT_TOP(), and then by repeatedly calling EDIT_KEY() and EDIT_DIS() until the editing is terminated (generally when a CR is struck), which will be signalled by EDIT_KEY().

The EDITDATA structure is defined as follows:

```
typedef struct {
    int edit_length;           /* current length of the edit buffer */
    char first_time;          /* flag for special processing on first char */
    char spec_flags;          /* bit 0 is tab handling */
    int prompt_window;        /* whether this belongs to prpt. Window*/
    char far *message_line;    /* the top line message for prompt */
    int message_line_length;   /* length of message_line */
    char far *prompt_line;    /* second line of prompt window */
    int prompt_line_length;   /* length of prompt_line */
    char edit_buffer[80]      /* work space */
    int line_array[2];        /* to be passed to mdit structure */
    MDITDATA mdit;           /* multi-line struct, to hold more info */
    int e_dispcols;
} EDITDATA;
```

The width of the edit buffer may exceed the width of the display window, up to 78 characters. Horizontal scrolling is handled by the services.

Multi-line editing is done similarly, but with calls to the mdit_*() series of functions. The multi-line editor allows the buffer to exceed the capacity of its display window, up to some length fixed at initialization.

The MDITDATA structure is defined as follows:

```
typedef struct {
    char far *m_buffer;       /* user supplied edit buffer */
    int m_length;            /* length of buffer */
    int m_pos;               /* current cursor position */
    int m_row, m_col;        /* location of edit area */
    int m_nrows, m_ncols;    /* dimensions of logical edit area */
    int m_yoff, m_disprows;  /* log.top line of display; lines of display*/
    char m_ccol;            /* cursor column */
    char m_modified;         /* 1 if buffer has been changed */
    char m_xoff;            /* 1st disp. col (for ticker fields only) */
    char wrapflag;          /* word wrap enable flag */
    int far *m_line;         /* caller supplied buffer for line starts */
                           /* must be at least m_nrows+1 long */
    int markon;             /* marking is currently active flag */
    char spec_flag;
    int markst;             /* offset of start of marked region, inclusive */
    int markend;            /* offset of end of region, inclusive */
    int m_dispcols;         /* displayable columns */
} MDITDATA;
```


The `m_nrows` field specifies the number of rows in the buffer. `m_disprows` specifies the number of rows displayed on the screen.

Applications using the editing services must include the header files `edit.h` and `interfac.h`.

edit_dis

```
int edit_dis(e);
EDITDATA far *e;
```

Displays the edit area, defined in `e`, on the screen.

Returns 0

edit_init

```
edit_init(e, ini_buf, ini_len, max_len, display_line, display_col);
EDITDATA far *e;
char far *ini_buf;
int ini_len, display_line, display_col, max_len;
```

Sets up the structure `e` for single line editing at an arbitrary location. `ini_buf` points to a string of length `ini_len`, which is the default value for the edit field. This buffer must be in the application's data segment. `max_len` indicates the maximum width of the field. `display_line` and `display_col` provides coordinates for the first character of the field.

Calls `edit_dis()` to display the field.

Returns 0.

edit_key

```
int edit_key(e, keystroke, presult);
EDITDATA far *e;
int keystroke;
int far *presult;
```

Processes `keystroke` in the context of `e`; that is, it does the work of inserting characters, etc, for the edit field. `keystroke` is the value returned in `event.data` by the Event Services.

Sets `*presult` to 1 if editing has been completed, otherwise, sets `*presult` to 0.

Returns 0.

edit_top

```
int edit_top(e, ini_buf, ini_buf_len, max_len, line1, len1, line2, len2);
EDITDATA far *e;
char far *line1;
char far *line2;
char far *ini_buf;
int ini_buf_len;
int len1;
int len2;
int max_len;
```

Sets up the structure *e* for single line editing. The field will be edited on the top line of the application's menu area if it occupies only one line; otherwise, it is edited on the second line of the menu area. The first character of the edit field will follow the last character of the prompt string (*line2*).

ini_buf points to a string of length *ini_buf_len*, which is the default value for the edit field. This buffer must be located in the application's data segment. *max_len* indicates the maximum width of the field. *line1*, of length *len1*, will be displayed as a message. *line2*, of length *len2*, will be displayed as the user prompt.

Calls *edit_dis()* to display the field.

Returns 0.

mdit_cutmark

```
void mdit_cutmark(mp);  
MDITDATA far *mp;
```

Deletes the characters included in the marked region. If an application wants to save the contents of marked region, it should copy the data prior to making the cutmark call. The offsets for the start and end of the region are maintained in the *markst* and *markend* fields of the MDITDATA structure. Applications should note that if *markend* == *m_length*, the last character should be ignored.

The marking mode is ended after successful completion of this function.

mdit_dis

```
void mdit_dis(mp);  
MDITDATA far *mp;
```

Displays the current contents of the edit field pointed to by *mp* and updates the cursor position.

mdit_fil

```
mdit_fil(mp,fp);  
MDITDATA far *mp;  
FILE far *fp;
```

Writes the buffer in *mp* to the open file associated with *fp*. No attempt is made to reposition the write pointer of the file.

Returns 0 if successful; otherwise, returns the error code reported by *m_write()*.

mdit_ini

```
void mdit_ini(mp,row,col,nrows,ncols,buf,len,wrapflag,disprows,line_array);  
MDITDATA far *mp;  
int row,col,nrows,ncols,len;  
char far *buf;  
int wrapflag;  
int disprows;  
int far *line_array;
```

Initializes the structure *mp* for multi-line editing. *row* and *col* establish the top-left corner of the edit region; *nrows* and *ncols* establish the dimensions of the logical edit region. The caller must provide the buffer through *buf*, which must be located in the application's data segment. Its length is specified in *len*, and the maximum length is 32767.

m_nrows specifies the number of rows in the edit buffer, while *m_disprows* specifies the number of screen rows that are displayed; that is, the latter defines the size of the region on the screen.

wrapflag, if nonzero, enables word wrapping.

line_array provides the mdit functions a buffer in which to store line offsets for each row. It must be at least $(m_nrows+1)*sizeof(int)$ bytes long, and must be located in the application's data segment.

To use mdit functions without vertical scrolling, *m_disprows* must equal *m_nrows*. Applications must also provide the buffer space for *line_array*.

Returns 0

mdit_ins_str

```
int mdit_ins_str(mp,str,len);  
MDITDATA far *mp;  
char far *str;  
int len;
```

Allows entry of multiple characters into the edit buffer. Screen will be updated after all characters have been inserted.

mdit_key

```
int mdit_key(mp,keystroke);  
MDITDATA far *mp;  
int keystroke;
```

Processes *keystroke* in the context of *mp*; that is, it does the work of inserting characters, etc, for the edit field. *keystroke* is the value returned in *event.data* by the Event Services.

If marking mode is in effect, only cursor movement keys will be accepted.

The cursor movement keys are as follows:

UP ARROW	Move to previous line, same column if possible.
DOWN ARROW	Move to next line, same column if possible.
LEFT ARROW	Move to previous character.
RIGHT ARROW	Move to next character.
HOME	First column of current line
END	Last column of current line
CONTROL HOME	First character of buffer.
CONTROL END	Last character of buffer.
CONTROL LEFT ARROW	Beginning of previous word.
CONTROL RIGHT ARROW	Beginning of next word.

The following keys also have special meanings:

BACKSPACE	Delete character to left of cursor
DELETE	Delete character under cursor.
CONTROL ENTER	Delete characters from cursor to end of line.
CONTROL BACKSPACE	Delete word to left of cursor.
TAB	Insert spaces from current position to next tab stop.

Returns 0.

mdit_mark

```
void mdit_mark(mp);  
MDITDATA far *mp;
```

Invokes the multi-line editor's marking mode. While marking is active, only cursor movement keys are accepted by `mdit_key()`. The current cursor position becomes the anchor, and the user may select text before or after the anchor, but the anchor cannot be moved.

An application may inspect or copy the text in the marked region by observing the current values in the `markst` and `markend` fields of the `MDITDATA` structure.

This function causes the display to be updated showing the initial marked region (1 character) in inverse video.

mdit_unmark

```
void mdit_unmark(mp);  
MDITDATA far *mp;
```

Ends the multi-line editor's marking mode. This function has no other effect on the `MDITDATA` structure, but it will redisplay the text without the marking attributes.

File Services

Operations on disk files are done through the following set of calls. Refer to the include file `fileio.h` for information on the structures and `m_error.h` for error codes used by the File Services.

Caution



The FILE Services use a FILE structure that is *not* compatible with the FILE structure defined in a Standard C Library include file, `stdio.h`. Modules using the System Manager File Services must not include `stdio.h`.

Applications may use either buffered and unbuffered operations. If only unbuffered operations are used, then the structure NBFIL can be used in place of FILE to save the RAM that would be used for the file buffer.

Applications that use File Services must include the header files `fileio.h` and `interfac.h`, and may want to include `m_error.h` for file-error codes.

m_close

```
int m_close(fp);  
FILE far *fp;
```

Closes the file associated with *fp*.

Returns 0 if successful, otherwise, returns an error code.

m_copydt

```
void m_copydt(fpsrc, fpdest);  
FILE far *fpsrc;  
FILE far *fpdest;
```

Copies the date and time modified values from *fpsrc* to *fpdest*, both of which must refer to files opened through System-Manager-file services.

m_create

```
m_create(fp, filespec, len, sys, nobuf);  
FILE far *fp;  
char far *filespec;  
int len;  
int sys;  
int nobuf;
```

Creates a new file to be identified by *filespec*. *len* is the length of *filespec*. *sys* is not used and should be zero. If *nobuf* is set, no buffering will be performed.

The file will be created and opened only if it does not already exist.

If successful, the values in *fp* will be updated and 0 will be returned. Otherwise, an error code will be returned.

m_delete

```
int m_delete(bp, len, sys);  
char far *bp;  
int len;  
int sys;
```

Deletes the file specified by *bp*. *len* is the length of the string pointed to by *bp*. *sys* is not used and should be set to zero.

Returns 0 if successful, otherwise, returns an error code.

m_fcreat

```
m_fcreat(fp, filespec, len, sys, nobuf);  
FILE far *fp;  
char far *filespec;  
int len;  
int sys;  
int nobuf;
```

Creates a new file identified by *filespec*. *len* is the length of *filespec*. *sys* is not used and should be set to zero. If *nobuf* is nonzero, no buffering will be performed.

Any existing file matching the *filespec* will be truncated.

If successful, the values in *fp* will be updated and 0 will be returned. Otherwise, an error code will be returned.

m_fdate

```
m_fdate(fp1, lp1);  
FILE far *fp1;  
char far *lp1;
```

For the file referenced by *fp1*, return the time and date values in **lp1*.

m_getattr

```
m_getattr(bp, len, sys, attr);  
char far *bp;  
int len;  
int sys;  
unsigned far *attr;
```

For the file specified by *bp*, returns in *attr* the file attributes as returned by DOS. *len* is the length of the file specification. *sys* is not used and should be set to zero.

m_getdir

```
int m_get_dir(drive,bp,lenp);  
char drive;  
char far *bp;  
int far *lenp;
```

Builds in *bp* the current directory for the specified *drive*. The returned path includes the drive designator (for example, "c:"). **lenp* contains the length of the string returned in *bp*.

Returns 0.

m_getdrv

```
int m_getdrv(drivep);  
char far *drivep;
```

Sets **drivep* to the drive letter of the current default drive. Reported values start at 'A'.

Returns 0 if successful, otherwise, returns an error code.

m_getfdt

```
void m_getfdt(fp,dateval);  
FILE far *fp;  
long far *dateval;
```

Gets the last modification date/time of the file *fp* and returns it in **dateval*. The format of *dateval* is that of the date/time stamp in a DOS directory entry.

m_get_sysdir

```
m_get_sysdir(bp);  
char far *bp;
```

Copies into **bp* the system directory path. On the HP 95LX, the system-directory path is C:_DAT\. On the Connectivity Pack, the system-directory path is controlled by the PIMS environment variable with default of C:\CPACK\.

m_ident

```
int m_ident(bp,len,sys,typep);  
char far *bp;  
int len;  
int sys;  
int far *typep;
```

Sets **typep* to indicate the file type of the file whose name is pointed to by *bp*. *len* indicates the length of the filename stored in *bp*. *sys* is not used and should be set to zero.

The file type is set to 1 for a file, set to 2 for a directory, set to 3 for a device, or set to 0 if filename is not found.

Returns 0 if successful, otherwise, returns an error code.

m_match

```
int m_match(matchp, flagp);  
MATCH far *matchp;  
int far *flagp;
```

For the *matchp* structure set up by *m_setpat()*, find the next matching file name. **flagp* is set to nonzero when a match is found; otherwise, **flagp* is set to zero.

Returns 0 under normal conditions; returns a nonzero error code with *flagp* set to 0 when an error other than no more matches is encountered.

m_mkdir

```
int m_mkdir(bp, len, sys);  
char far *bp;  
int len;  
int sys;
```

Creates the directory specified in the string *bp*, where *len* is the length of *bp*. *sys* is not used and should be set to zero.

Returns 0 if successful, otherwise, returns an error code.

m_open

```
int m_open(fp, filespec, len, sys, nobuf);  
FILE far *fp;  
char far *filespec;  
int len;  
int sys;  
int nobuf;
```

Opens an existing file identified by *filespec*. *len* is the length of *filespec*. *sys* is not used and should be set to zero. If *nobuf* is nonzero, no buffering will be performed.

If successful, the values in *fp* will be updated and 0 will be returned; otherwise, an error code will be returned.

m_openro

```
int m_openro(fp, filespec, len, sys, nobuf);  
FILE far *fp;  
char far *filespec;  
int len;  
int sys;  
int nobuf;
```

Opens an existing file identified by *filespec* in read-only mode. *len* is the length of *filespec*. *sys* is not used and should be set to 0. If *nobuf* is nonzero, no buffering will be performed.

If successful, the values in *fp* will be updated and 0 will be returned. Otherwise, an error code will be returned.

m_putfdt

```
void m_putfdt(fp,dateval);  
FILE far *fp;  
long dateval;
```

Sets the last modification date/time of the file *fp* to *dateval*. The format of *dateval* is that of the date/time stamp in a DOS directory entry.

m_read

```
m_read(fp,buffer,len,lenp);  
FILE far *fp;  
char far *buffer;  
int len;  
int far *lenp;
```

Reads up to *len* bytes of data from the I/O stream *fp* into the memory pointed to by *buffer*. **lenp* tells how many bytes were actually read. If **lenp* != *len* and the returned value is zero, the end-of-file has been reached.

Returns 0 if successful, otherwise, returns an error code.

m_rename

```
int m_rename(bp1,len1,sys1,bp2,len2,sys2);  
char far *bp1;  
char far *bp2;  
int len1;  
int sys1;  
int len2;  
int sys2;
```

Rename the file specified by the name pointed to by *bp1* to the name pointed to by *bp2*. *len1* and *len2* are the lengths of the filenames pointed to by *bp1* and *bp2* respectively. *sys1* and *sys2* are not used and should be set to zero.

Returns 0 if successful, otherwise returns an error code.

m_rmdir

```
int m_rmdir(bp,len,sys);  
char far *bp;  
int len;  
int sys;
```

Removes the directory specified in the string *bp*, where *len* is the length of *bp*. *sys* is not used and should be set to zero.

Returns 0 if successful, otherwise, returns an error code.

m_seek

```
m_seek(fp,mode,seek);  
FILE far *fp;  
int mode;  
long seek;
```

Sets the current position in the file associated with *fp*. *seek* is the signed-byte offset of where to set the position, relative to the value of *mode*. Modes defined in *fileio.h* are:

Mode	Meaning
seek_beginning	Offset from start of file.
seek_current	Offset from current position.
seek_end	Offset from end of file.

m_setattr

```
m_setattr(bp,len,sys,attr);  
char far *bp;  
int len;  
int sys;  
int attr;
```

For the file specified by *bp*, sets the file attributes to *attr*. *len* is the length of the file specification. *sys* is not used and should be set to zero.

m_setdir

```
int m_setdir(bp,len);  
char far *bp;  
int len;
```

Given the path *bp* of length *len*, sets the current directory for the drive included in the path. Returns 0 if successful; otherwise, returns an error code.

m_setdrv

```
int m_setdrv(drive);  
char drive;
```

Sets the default drive to *drive*. Drive values should start with 'A'. Returns 0 if successful, otherwise returns an error code.

m_setpat

```
m_setpat(matchp, bp, len, sys);  
MATCH far *matchp;  
char far *bp;  
int len;  
int sys;
```

Builds the MATCH structure used for searching for files with the `m_match()` function. *bp* is the pattern to match, including any path information. *len* specifies the length of *bp*. *sys* is not used and should be set to zero.

m_tell

```
int m_tell(fp, seekp);  
FILE far *fp;  
long far *seekp;
```

Sets **seekp* to the current position in the file associated with *fp*.

Returns 0 if successful, otherwise, returns an error code.

m_volume

```
void m_volume(bp, sizep);  
char far *bp;  
long far *sizep;
```

Sets *bp* to the name of the current volume and sets **sizep* to the amount of freespace available on that volume.

m_write

```
m_write(fp, buffer, len);  
FILE far *fp;  
char far *buffer;  
int len;
```

Writes *len* bytes of data to the I/O stream *fp* from the memory pointed to by *buffer*.

Returns 0 if successful, otherwise, returns an error code.

Process Management Services

m_fini

```
void m_fini(void);
```

Signals to the System Manager that the current application is done. Control is never returned to the application.

m_init

```
void m_init(void);
```

Must be called by the application's main entry point.

m_lock

```
void m_lock(void);
```

Increments the system lock count, which prevents interruption of the currently running application while it is nonzero. That is, it causes other applications' hot keys to be ignored.

This could be used, for example, to prevent a task switch until after an error message is acknowledged by the user.

m_reg_app_name

```
m_reg_app_name(appname);  
char far *appname;
```

Records the null-terminated string pointed to by *appname* as the name of the application. The application name is used in the low memory closeout screen. The third field in the APNAME.LST entry provides an initial name for external applications. Internal applications use this function to record their names.

m_spawn

```
int m_spawn(command_str, command_len, sysflag, prompt_str);  
char far *command_str;  
int command_len;  
int sysflag;  
char far *prompt_str;
```

command_str points to the string that will be passed to COMMAND.COM for execution. *command_str* must be terminated with a carriage return ('\r' = 0x0d). *command_len* is the length of *command_str*.

sysflag can be 0 or 2. Applications should set *sysflag* to 0; value 2 is reserved for internal System-Manager use. When *sysflag* is set to 0, the calling application must be the only open application. When *sysflag* is set to 2, the spawn will be attempted even if other applications are open.

prompt_str points to a null-terminated string that will be displayed by the System Manager at the top of the screen prior to invoking the program.

Returns 0 if successful; 513 if another application is active; otherwise, a DOS error code as returned by EXEC (INT 21h, AH = 4bh).

m_unlock

```
void m_unlock(void);
```

Decrements the system lock count. If calls are not nested, allows the system to take control from the current application.

Clipboard Services

The following functions provide a generic means of passing information between applications. In the current form, applications can only get or put information onto the clipboard; there is no provision for forcing information through to another application. Data on the clipboard is named, however. Applications can therefore establish regimes of communication as needed.

The default representation of data is that of "TEXT". All applications using the clipboard at all should write a "TEXT" representation, and read a "TEXT" representation if no other known representations are found. Data in the "TEXT" form should be ASCII characters, with a lone carriage return (0x0d) used to mark the end of each line.

Clipboard return codes are contained in `cbcodes.h`.

m_cb_read

```
m_cb_read(index, offset, data, length);  
char far *data;  
int index;  
unsigned int length, offset;
```

Reads *length* bytes into *data* from the representation associated with *index*. *offset* indicates the starting offset in the clipboard buffer from which data will be read.

m_cb_write

```
m_cb_write(data, length);  
char far *data;  
unsigned int length;
```

Writes *length* bytes from *data* to the representation opened by `m_new_rep()`. The maximum length of a single call to `m_cb_write` is 1024. Multiple calls may be made to `m_cb_write()` if needed.

m_close_cb

```
int m_close_cb();
```

Attempts to close the clipboard and allow other applications to claim it. Returns 0 if successful; returns -1 if the clipboard is not currently open.

m_fini_rep

```
m_fini_rep(void);
```

Signals the clipboard that no further data will be sent for the current representation.

m_new_rep

```
m_new_rep(rep_name);  
char far *rep_name;
```

Prepares the clipboard to receive a representation under the name *rep_name*. Multiple representations of the same data may be copied to the clipboard.

m_open_cb

```
int m_open_cb(void);
```

Attempts to claim the clipboard and lock out requests from other applications. Returns 0 if successfully claimed, otherwise returns a nonzero value.

m_rep_index

```
int m_rep_index(name, index, length);  
char far *name;  
int far *index;  
unsigned int far *length;
```

Gets the *index* and *length* of a representation, given a *name*. Returns 0 with pointers updated if successful; otherwise, returns an error code.

Applications attempting to respond to a Paste command should call this function with the appropriate representation name ("TEXT" as default).

m_rep_name

```
m_rep_name(index, name, length);  
int index;  
char far *name;  
unsigned int far *length;
```

Gets the *name* and *length* of a representation, given an *index*. Returns 0 with pointers updated if successful; otherwise, returns an error code.

m_reset_cb

```
m_reset_cb(author);  
char far *author;
```

Clears the contents of the clipboard and establishes the current application, specified by the string *author*, as the creator of clipboard contents. Applications should attempt to use a unique string for this value. Returns 0 if successful; returns -1 if there is an error.

Sound Services

These services are used to signal the user through the system speaker.

m_asound

```
void m_asound(index);  
int index;
```

Generates one of several various sound patterns, as specified by *index*. There are currently seven (0-6) supported patterns. The first four are used to generate tones for 1-2-3, the remaining three are more complicated alarm sounds.

m_beep

```
void m_beep(void);
```

General purpose sound for error alerts.

m_soundoff

```
void m_soundoff(void);
```

Turns off the current sound. If the speaker is on when this function is called, the sound will actually continue until the next BIOS clock tick.

m_thud

```
void m_thud(void);
```

Generally used when a user's keystroke cannot be interpreted in the current application state.

Memory Management Services

When an application is invoked, the System Manager must provide the RAM space indicated in the application's image structure. This amount of memory should be sufficient for ordinary uses of the application.

There are limited instances when more RAM is required, at which point the application calls `m_alloc()` or `m_alloc_large()` to expand the size of its data area. If successful, the application's data area may have been moved by the System Manager. Consequently, applications must be careful to *not* store the current data segment or to update these stored values after memory-allocation calls.

Dynamic memory allocation is implemented by resizing the memory block belonging to the application. The new buffer will therefore begin at the previous end of the memory block. Calls to release memory are translated into requests to shrink the memory block to the specified level. These services should not be confused with the functions `malloc()` and `free()`.

DOS memory allocation services are also available. However, it is likely that all DOS memory will have been consumed by 1-2-3, and memory will be available only by using the System-Manager's memory-management services.

m_alloc

```
void near * m_alloc(expsize);  
unsigned int expsize;
```

Attempts to expand the data space occupied by the accessory by *expsize* bytes. If successful, the return value will be the offset (near address) of the new buffer. The function returns 0 on failure. Failure may be caused by a request which would grow the applications total data space beyond 64K or by an exhaustion of system memory.

If *expsize* is not a multiple of 16 (the size of a paragraph), it will be rounded up to the next multiple.

m_alloc_large

```
void near * m_alloc(exp_paras);  
unsigned int exp_paras;
```

Attempts to expand the data space occupied by the accessory by *exp_paras* paragraphs (16 byte units). If successful, the return value will be the paragraph offset from the beginning of the application's data segment. The function returns 0 on failure.

This function must be used with care, since it is possible to claim space that cannot be accessed through the DS register (> 64K). Since the entire data space of the application may be moved during certain System Manager calls, the application must be careful either not to store any segment values that may become invalid or to update these stored values after any time that the data segment may have moved.

m_free

```
unsigned int m_free(ptr);  
void near *ptr;
```

Shrinks the data space claimed by the application by releasing all memory beyond *ptr*. Applications should make sure that the value of *ptr* is above or equal to the first value returned by `m_alloc()`.

m_free_large

```
void m_free_large(paras);  
unsigned int paras;
```

Shrinks the data space claimed by the application beginning at *paras* that is the paragraph offset from the beginning of the application's data segment. Applications must make sure that the value of *paras* is above or equal to the first value returned by `m_alloc_large()`.

Date/Time Services

There are three subsets of date and time services. Primarily, there are calls to get and set the system time and to get the system-wide display format. Secondly, there are alarm calls, which are primarily used by the Appointment Book and Watch applications. Finally, there are stopwatch functions used by the Watch.

The date and time information is maintained in the following structure:

```
typedef struct {
    char dt_order;      /* month-day-year order, 0 = MDY, 1 = DMY, 2 = YMD */
    char dt_dsep;       /* date separator */
    char dt_tsep;       /* time separator */
    cahr dt_24_hr;      /* nonzero means 24 hour time */
} DTINFO;
```

The following structure is used to schedule alarms:

```
typedef struct {
    char a_hour;         /* time of alarm */
    char a_minute;
    char a_second;
    char a_pad;          /* supplied by caller */
    int a_interval;      /* reschedule interval (seconds) */
    char a_use_seconds;  /* are seconds significant */
    char a_sound;        /* alarm sound */
    char message[ALARM_MSG_LEN]; /* message displayed when alarm goes off */
    char owner;          /* task id of owner */
    char special;        /* apps own use for sub-class */
    char extra[ALARM_EXTRA_LEN]; /* apps own use for specific data */
} ALARM;
```

This final structure is used to represent the actual time and date:

```
typedef struct {
    int dt_year;         /* year in the range of 1980, to 2116 */
    char dt_month;       /* Jan == 1 */
    char dt_date;        /* 1st == 1 */
    char dt_day;         /* Day of week, 0->6; otherwise unknown */
    char dt_hour;
    char dt_minute;
    char dt_second;
    cahr dt_hundreth;
} DTM;
```

m_alarm

```
m_alarm(alarmp,type);  
ALARM far *alarmp;  
unsigned int type;
```

Sets an alarm as defined in the ALARM structure pointed to by *alarm* and associates it with the application-determined alarm *type*.

m_dtinfo

```
m_dtinfo(ntp);  
DTINFO far *ntp;
```

Fills in the DTINFO structure pointed to by *ntp* with a special time and date information as described below. Due to the special nature of this information, *m_dtinfo()* should probably not be used.

The information returned in *ntp* does not have complete separator information. The current separator information is stored in the settings structure (see the Configuration Services section). In that structure, there are two separators for both date and time, to provide for flexibility of localization. Only the first character is copied into the DTINFO structure by *m_dtinfo()*. It is probably easier for applications to get these fields directly from the settings structure.

These values are not based on the date and time settings selected in the SETUP utility. Rather, they are part of the localization done when the user selects his country at very cold boot. The values are used exclusively to format date and time values in the Filer's directory listings. They should match what a localized version of DOS would display in response to the DIR command.

m_getdtm

```
m_getdtm(dtmp);  
DTM far *dtmp;
```

Returns the system date/time in the DTM structure pointed to by *dtmp*.

m_get_sw

```
m_get_sw(start_time,elapsed_time,onflag);  
TIME far *start_time;  
TIME far *elapsed_time;  
char far *onflag;
```

Copies contents of system manager buffers and variables into those supplied by the application.

m_get_timer

```
m_get_timer(start_time, elapsed_time, onflag);  
DTM far *start_time;  
DTM far *elapsed_time;  
char far *onflag;
```

Copies contents of System Manager buffers and variables into those supplied by the application.

m_parse_date

```
int m_parse_date(rule, input, dtm);  
int rule;  
char far *input;  
DTM far *dtm;
```

Parses a null-terminated input string according to *rule*, and fills in the day, date, week, and year fields of the DTM structure pointed to by *dtm* (the time of day fields are unaffected). Unspecified fields that do not cause parsing errors are set to 0.

The available rules (found in settings.h) are:

- 0 - rule appropriate to currently selected system date format.
- 1 - DR_DMY_LIM - date, month, year and year must be in limited range.
- 2 - DR_MDY_LIM - month, date, year and year must be in limited range.
- 3 - DR_YMD_LIM - year, month, date and year must be in limited range.
- 4 - DR_MD - just month and date.
- 5 - DR_DM - just date and month
- 6 - DR_MY_LIM - month and year and year must be in limited range.

Years with the rules above must be in the range of 1900 to 2099. Two-digit input of years is assumed to be in the range of 1980 to 2079. For parsing of arbitrary years, OR DR_ANY_YEAR (0x8) onto the rule code.

Returns 0 if parsing is successful, nonzero if input is improperly formatted or invalid value.

m_parse_time

```
int m_parse_time(rule, input, dtm);  
int rule;  
char far *input;  
DTM far *dtm;
```

Parses a null-terminated input string according to *rule*, and fills in the hour, minute, second, and hundredth fields of the DTM structure pointed to by *dtm* (the date fields are unaffected). Unspecified fields that do not cause parsing errors are set to 0.

The available rules (found in `settings.h`) are:

- 0 = rule appropriate to currently selected system time format.
- 1 - TM_H_M_S_P = 12 hour clock, with optional am/pm specifier.
- 2 - TM_H_M_S+24 = 24 hour clock.
- 3 - TM_HM_S_24 = 24 hour clock with hours and minutes together.
- 4 - TM_H_M_S_C_24 = 24 hour clock to hundredth resolution.
- 5 - TM_H_M_P = 12 hour clock, without seconds.
- 6 - TM_H_M_24 = 24 hour clock, without seconds.
- 7 - TM_HM_24 = 24 hour clock with hours minutes together, without seconds.

Returns 0 if parsing is successful, nonzero if input is improperly formatted or invalid value.

m_posttime

```
void m_posttime(void);
```

Gets the current time and writes it to the display in the standard location.

m_setdtm

```
m_setdtm(dtmp);  
DTM far *dtmp;
```

Sets the system date/time with the contents of the DTM structure pointed to by *dtmp*.

m_start_sw

```
m_start_sw(time);  
TIME far *time;
```

Saves the starting time specified in the TIME structure pointed to by *time* in a system manager buffer and sets the System-Manager-stopwatch flag to on.

m_stop_sw

```
m_stop_sw(time);  
TIME far *time;
```

Saves the current elapsed time, as calculated by an application and stored in the TIME structure pointed to by *time*, in a System-Manager buffer and sets the stopwatch flag to off.

m_start_timer

```
m_start_timer(start_dtm);  
DTM far *start_dtm;
```

Saves the starting time specified in the DTM structure pointed to by *start_dtm* in a System-Manager buffer and sets the System-Manager-timer flag to on.

m_stop_timer

```
m_stop_timer(elapsed_time);  
DTM far *elapsed_time;
```

Saves the current elapsed time, as calculated by an application and contained in the DTM structure pointed to by *elapsed_time*, in a System Manager buffer and sets the timer flag to off.

m_tell_anytime

```
char far *m_tell_anytime(content,row,col,dtinfo,dtm);  
int content;  
int row;  
int col;  
DTINFO far *dtinfo;  
DTM far *dtm;
```

Formats the date or time contained in the structure pointed to by *dtm*, according to *content*. Returns a pointer to a System Manager buffer containing the formatted time. The *content* parameter uses the same values as with *m_telltime()*. *row*, *col*, and *dtinfo* are ignored, but are currently left in to preserve existing calls.

m_telltime

```
m_telltime(content,row,col);  
int content,row,col;
```

Displays the current date or time, according to *content*, at the given row and column coordinates. The current system date and time formats are used.

The defined content values are as follows:

- 0 = date only
- 1 = time only
- 2 = date and time
- 3 = date prefixed with day of week.

m_xalarm

```
m_xalarm(type);  
unsigned int type;
```

Kills all alarms of the application-defined *type*.

Printer Services

The printer services manage communications with and translation of characters for specific printer types. The active printer is selected through SETUP. That selection is used to control access tables for special characters and print-control sequences.

m_close_printer

```
void m_close_printer(void);
```

Any remaining characters are flushed to the printer, and the printer channel is closed.

m_init_printer

```
unsigned int m_init_printer(void);
```

The printer tables are initialized for the selected printer. This call does not initialize the communications channel. It is intended to be called from 1-2-3 and SETUP only. Other applications should use `m_open_printer`. A segment value is returned which is used by 1-2-3 to access printer information in a standard format.

m_open_printer

```
void m_open_printer(void);
```

Prepares the communications channel to converse with the printer. The printer baud rate is established and the printer tables are set to the selected device.

m_trans_printer

```
int m_trans_printer(ch, bp);  
char ch;  
char far *bp;
```

This routine should be used by applications that wish to do their own printer communications. The CP850 character passed as *ch* is converted to the sequence required by the selected printer to produce the equivalent character. The resulting sequence is placed in the buffer pointed to by *bp*. This buffer must be at least 48 bytes long. The number of characters in the buffer is returned by this routine. That number will always be greater than zero.

m_write_printer

```
int m_write_printer(bp, len);  
char far *bp;  
unsigned int len;
```

The string pointed to by *bp* is sent to the printer. The string is expected to be composed of CP850 characters and of length *len*. Any special sequences required to produce the specified characters on the output are supplied by this routine.

This routine does not add a carriage return (0x0d), line feed (0x0a) pair for terminating a line so these should be included in the string as desired.

Configuration Services

Various System Manager settings are maintained in RAM in a **SETTINGS** structure. The functions in this section provide access to this information.

Applications using these services *must* include **settings.h** and **interfac.h**.

See **settings.h** for the definition of the **SETTINGS** structure as well as the manifest constants which specify the values that may be found in the structure.

m_get_settings_addr

```
SETTINGS far *m_get_settings_addr(void);
```

Returns a pointer to the System Manager's settings structure. Applications other than **SETUP** should never change any values in this structure, but should use this only to enquire about system settings.

Communications Services

The Communication Services provide access to the serial port.

Applications using the Communications Services must include **comio.h** and **interfac.h**.

ComAnswer

```
int ComAnswer(handle,mode);  
com_handle handle;  
int mode;
```

Put the modem/com line in answer mode. *handle* identifies the port to which the modem is connected. The only valid value for *mode* is COM_ANS_NOWAIT.

Returns 0 if successful; otherwise returns an error code.

ComBreak

```
int ComBreak(handle,duration);  
com_handle handle;  
int duration;
```

Sends a break to the port associated with *handle*. The break state will be held for *duration* milliseconds.

Returns 0 if successful, otherwise, returns an error code.

ComClose

```
int ComClose(handle);  
com_handle handle;
```

Closes the communication port associated with *handle*.

Returns 0 if successful, otherwise, returns an error code.

ComCommand

```
int ComCommand(handle,cmd,cmdlen);  
com_handle handle;  
char far *cmd;  
int cmdlen;
```

Sends the modem specific command pointed to by *cmd* to the modem attached to the port associated with *handle*, where *cmdlen* is the length of the command.

The modem will be put into command state and the command will be sent verbatim. No translation will occur.

The transmit queue associated with *handle* will be flushed before the command is sent. To ensure that all data is sent, use ComXmitting() before sending the command.

Returns 0 if successful, otherwise, returns an error code.

ComConfigure

```
int ComConfigure(port, IRQnum, IObase, modem, permflag) ;  
int port;  
int IRQnum;  
int IObase;  
int modem;  
int permflag;
```

Note



There is no need for this routine to be called on the HP 95LX, since there is only one COM port, and one configuration that may be used.

Configures the communications *port* where *IRQnum* is the interrupt-request number, *IObase* is the base address of the port in I/O space, and *modem* is the modem type.

If *permflag* is nonzero, the information will be entered into the COM driver's permanent database, otherwise it will not be permanent.

Returns 0 if successful, otherwise, returns an error code.

ComDial

```
int ComDial(handle, number) ;  
com_handle handle;  
char far *number;
```

Dials the number contained in *number* on the modem associated with *handle*. *number* includes punctuation and/or commas and the number will be dialed according to the modem's specifications. For example, ','s cause a delay.

Returns 0 if successful, otherwise, returns an error code.

ComForceXoff

```
int ComForceXoff(handle) ;  
com_handle handle;
```

Forces XOFF state on the port associated with *handle*.

Returns 0 if successful, otherwise, returns an error code.

ComForceXon

```
int ComForceXon(handle) ;  
com_handle handle;
```

Forces XON state on the port associated with *handle*.

Returns 0 if successful, otherwise, returns an error code.

ComHangUp

```
int ComHangUp(handle);  
com_handle handle;
```

Hangs up phone on the modem attached to the port associated with *handle*. The transmit queue associated with *handle* will be flushed before the command is sent. To ensure that all data is sent, use ComXmitting() before sending the hang-up command.

Returns 0 if successful, otherwise, returns an error code.

ComHayesCommand

```
int ComHayesCommand(handle,cmd,cmdlen);  
com_handle handle;  
char far *cmd;  
int cmdlen;
```

Sends the Hayes command pointed to by *cmd* to the modem attached to the port associated with *handle*, where *cmdlen* is the length of the command.

The transmit queue associated with *handle* will be flushed before the command is sent. To ensure that all data is sent, use ComXmitting() before sending the command.

Returns 0 if successful, otherwise, returns an error code.

ComGet

```
int ComGet(handle,settings);  
com_handle handle;  
com_settings far *settings;
```

Gets the communications settings for the port associated with *handle*. These settings are returned in the structure pointed to by *settings*.

Returns 0 if successful, otherwise, returns an error code.

ComGetModem

```
int ComGetModem(handle);  
com_handle handle;
```

If successful, returns the modem type for the port associated with *handle*, otherwise, returns an error code. If successful, the returned value is either COM_MDM_NONE or COM_MDM_HAYES indicating either no modem or Hayes-compatible modem, respectively.

ComOpen

```
int ComOpen(handle,port);  
com_handle far *handle;  
int port;
```

Opens the communication port specified by *port* and returns a handle in the location given by *handle*. The returned handle is associated with this port and is used in subsequent communications function calls.

Returns 0 if successful, otherwise, returns an error code.

ComQryErr

```
int ComQryErr(handle);  
com_handle handle;
```

Returns the error status of the port associated with *handle*. A return value of 0 indicates no error. Possible nonzero error values are E_OVERN, E_PARITY, E_FRAME, etc.

Use of ComStatus is preferred over use of ComQryErr.

ComQryRxQue

```
int ComQryRxQue(handle,size,free);  
com_handle handle;  
int far *size;  
int far *free;
```

Queries the status of the receive queue for the port associated with *handle*. Sets the location specified by *size* to the total queue size and sets the location specified by *free* to the number of free bytes in the queue.

Returns 0 if successful, otherwise, returns an error code.

ComQryTxQue

```
int ComQryTxQue(handle,size,free);  
com_handle handle;  
int far *size;  
int far *free;
```

Queries the status of the transmit queue for the port associated with *handle*. Sets the location specified by *size* to the total queue size and sets the location specified by *free* to the number of free bytes in the queue.

Returns 0 if successful, otherwise, returns an error code.

ComReceiveBytes

```
int ComReceiveBytes(handle, data, datalen);  
com_handle handle;  
char far *data;  
int far *datalen;
```

Receives bytes from the port associated with *handle*. The data is received in the location specified by *data* up to a maximum of *datalen* bytes. On return, *datalen* is set to the number of bytes actually received, which can be zero.

Returns 0 if successful, otherwise, returns an error code.

ComReset

```
int ComReset(handle, reset);  
com_handle handle;  
int reset;
```

Resets the port associated with *handle* in accordance with the reset options specified in *reset*. *reset* uses bit values to indicate what is to be reset. The constants listed below can be OR'd together to create the desired reset actions.

Constant	Bit	Action if Bit is Set
COM_RESET_LINE	0	Line is reset.
COM_RESET_TXB	1	Transmit buffer is flushed.
COM_RESET_RXB	2	Receive buffer is flushed.
COM_RESET_MODEM	3	Modem is reset.
COM_RESET_RXFLOW	4	Receiver's 'S state is reset.
COM_RESET_TXFLOW	5	Transmitter's 'S state is reset.

Always returns 0.

ComSendBytes

```
int ComSendBytes(handle, data, option, datalen);  
com_handle handle;  
char far *data;  
int option;  
int far *datalen;
```

Queues *datalen* bytes of data pointed to by *data* for transmission over the port associated with *handle*. *control* uses bit values to indicate queuing options. The constants listed below can be OR'd together to create the desired actions.

Constant	Bit	Action if Bit is Set
COM_CTL_WHOLE	0	Do not queue any data unless all <i>datalen</i> bytes will fit in the output queue.
COM_CTL_SETRCV	1	Turn receiver on after sending data.

If COM_CTL_WHOLE is not specified, partial data may be queued and several calls to ComSendBytes may be required to complete sending all the data.

If successful, returns 0 and stores the number of bytes queued for transmission in the location specified by *datalen*. If not successful, an error code is returned. In particular, E_NOFIT will be returned if COM_CTL_WHOLE was specified and there is not enough free space in the transmit queue to fit the data. In this case, the value pointed to by *datalen* will be set to zero.

ComSet

```
int ComSet(handle,settings);  
com_handle handle;  
com_settings far *settings;
```

Sets the communications settings for the port associated with *handle* to the values contained in the structure pointed to by *settings*.

Note that the values used may differ from what is specified in the structure. For instance, if IR is used, HALF-DUPLEX is forced, baud rate is limited to 2400, and FLOW control is turned off. On return the passed structure will have the settings that are actually being used.

Returns 0 if successful, otherwise, returns an error code.

ComSetDtr

```
int ComSetDtr(handle,state);  
com_handle handle;  
int state;
```

Sets the DTR state on the port associated with *handle*. If *state* is zero, DTR will be set OFF, else DTR will be set ON.

ComStatus

```
int ComStatus(handle);  
com_handle handle;
```

Returns the error status of the port associated with *handle*. A return value of 0 indicates no error. Possible nonzero error values are E_OVERN, E_PARITY, E_FRAME, etc.

ComXmitting

```
int ComXmitting(handle);  
com_handle handle;
```

Tests the transmitting status on the port associated with *handle*.

Returns 0 if nothing is being transmitted, else returns a nonzero value.

Miscellaneous Services

drawbox

```
void drawbox(box_name);  
char far *box_name;
```

May be used to display the application name on draw the double line separating the menu area from the data window. The application name to be displayed is pointed to by *box_name*. The display is cleared before the box is drawn.

m_errmsg

```
void m_errmsg(code, bp, len, lenp);  
int code;  
char far *bp;  
int len;  
int far *lenp;
```

Returns the string which serves as the error message for the code passed in *code*. String is copied into buffer pointed to by *bp*, whose maximum length is specified by *len*. The actual length of the returned string is returned in **lenp*.

If an error message is not found for the given code, the string "Error n" is generated.

Actual codes are listed in *m_error.h*.

message

```
void message(str1, len1, str2, len2);  
char far *str1;  
char far *str2;  
int len1;  
int len2;
```

Displays one or two lines of text in the menu area of the application window. The string pointed to by *str1* is displayed on the first line and the string pointed to by *str2* is displayed on the second line. *len1* and *len2* give the lengths of the strings *str1* and *str2*, respectively. Strings that are wider than the display area will be truncated.

When message is called, the screen hidden by the message box is saved by the System Manager, and will be restored by a call to *msg_off()*.

Generally speaking, applications should lock the system (*m_lock()*) until the message is cleared by the user.

message3

```
void message3(str1, len1, str2, len2, str3, len3);  
char far *str1;  
char far *str2;  
char far *str3;  
int len1;  
int len2;  
int len3;
```

Similar to `message()` except that it supports three lines of text.

msg_off

```
void msg_off(void);
```

Clears the message posted by a prior call to `message()` or `message_3()` and restores the text that was covered by the message box.

m_form_ft

```
char far *m_form_ft(dtm);  
DTM far *dtm;
```

Given the time and date in the DTM structure pointed to by *dtm*, returns a pointer to a null-terminated string in the format described above, suitable for display in a directory listing.

Note



The contents of the buffer will be destroyed by the next call to any of the date and time formatting functions (made by the caller), so it should be used immediately.

This function is used by the Filer to format the date and time used in its directory listings.

showname

```
void showname(box_name);  
char far *box_name;
```

Displays *box_name* on the menu line of the application's window.

Resource Services

The Resource Services are only applicable to the built-in applications.

Help Services

At this time, the Help Services are only applicable to the built-in applications.

Collating Services

The System Manager provides character and string comparison functions which may be used to sort items. These comparisons are more useful than those based solely on the machine's character set.

m_col_cpsearch

```
int m_col_cpsearch(sptr, slen, dptr, dlen, dir);  
char far *sptr;  
int slen;  
char far *dptr;  
int dlen;  
int dir;
```

The data string of length *dlen* pointed to by *dptr* is searched for the search string of length *slen* pointed to by *sptr*. The search is in a forward direction if *dir* = -1, and backwards if *dir* = +1. The search comparison follows the same rules as for *m_col_cpstr*.

If found, returns the offset of the beginning of the search string within the data string. If not found, returns 0.

m_col_cpstr

```
int m_col_cpstr(str1, len1, str2, len2);  
char far *str1;  
int len1;  
char far *str2;  
int len2;
```

Compares the supplied code page 850 string of length *len1* pointed to by *str1* with the code page 850 string of length *len2* pointed to by *str2*. Comparison is case- and accent-insensitive: characters a, A, and ä are considered equivalent. Graphical characters also all evaluate together.

Returns:

```
0   if the strings are equal  
+1  if string 1 precedes string 2  
-1  if string 2 precedes string 1
```

m_col_init

```
void m_col_init(void);
```

Initializes internal pointers in accordance with the global sort selection. These are used by the LICS comparison routines.

m_col_tolower

```
void m_col_tolower(sptr, len);  
char far *sptr;  
int len;
```

The code page 850 string of length *len* pointed to by *sptr* is scanned through its length, and any upper case characters found are converted to their lower-case equivalents.

m_col_toupper

```
void m_col_toupper(sptr, len);  
char far *sptr;  
int len;
```

The code page 850 string of length *len* pointed to by *sptr* is scanned through its length, and any lower case characters found are converted to their upper case equivalents.

1-2-3 Bridge Services

The System Manager includes a bridge function, with a set of subfunctions, which allow applications to communicate with 1-2-3.

Applications using the bridge services must include **bridge.h**.

Applications prepare a bridge parameter block (see below) and make a call to the bridge service. The system manager confirms that 1-2-3 is loaded and performs a context-switch so that the applicaiton is deactivated (but not notified, since it is generating the bridge request) and 1-2-3 is reactivated with a Bridge-Request Event, that contains a pointer to the application-supplied parameter block.

1-2-3 then performs the requested action and signals the System Manager that the service has been completed. 1-2-3 is fully responsible for the values returned in the parameter block (except for -1 in the retcode that indicates that 1-2-3 is not loaded). The System Manager then switches control back to the application.

In general, the application should repaint its screen when it regains control.

The bridge parameter block definition is:

```
typedef BRIDGE_BP {
    int bpb_funcode;           /* function code */
    int bpb_retcode;          /* return value */
    char bpb_rangename[16];    /* ASCIIZ name of range */
    int bpb_startcol;         /* column coordinates of range start */
    int bpb_startrow;         /* row coordinates of range start */
    int bpb_endcol;           /* column coordinates of range end */
    int bpb_endrow;           /* row coordinates of range end */
    int bpb_order;            /* order (row or col first) of range data */
    int bpb_bufsize;          /* size in bytes of supplied buffer */
    char near *bpb_buffer;     /* buffer for prompt and range data (must be
                                located in same seg. as the struct) */
} BRIDGE_BP;
```

Range specification (used even for a single cell) is stored in the bpb_startcol, bpb_startrow, bpb_endcol, and bpb_endrow fields of the bridge-parameter block. Coordinates are zero based so that the cell A1 is stored as COL=0, ROW=0.

If the range is to be accessed by name, the name is stored as an ASCIIZ string in the bpb_rangename field.

Applications and 1-2-3 exchange cell contents using a buffer composed of variable length cell records. The records contain a single byte type followed by a type specific body. The cells are in the order declared by a flag passed with the get/set functions.

The following cell types are defined:

Type Code	Type	Body Length
B	Blank	0
I	Integer	2
N	Float	8
S	ASCII String	variable
F	Formula	variable

bridge_serv

```
void bridge_serv(bpb);
BRIDGE_BP far *bpb;
```

The bridge services are used by setting a function code in the bridge parameter block and then calling `bridge_serv`. The valid function codes are described below.

bridge_test

This function should be called before using any other services to ensure that 1-2-3 is loaded and ready to receive bridge calls.

INPUT:

```
bpb_funcode    BRIDGE_TEST (0)
```

OUTPUT:

```
bpb_retcode    1 if 1-2-3 is loaded and safe for bridge services
               0 if 1-2-3 is loaded, but busy.
               -1 if 1-2-3 is not loaded.
```

bridge_getrange

This function switches to 1-2-3 POINT mode, for user to select range. Range can be entered by painting, or by name. Pressing the NAMES key (F3) will switch to NAMES mode, allowing the user to select by name.

INPUT:

```
bpb_funcode    BRIDGE_GETRANGE (1)
bpb_buffer     ASCII string containing user prompt.
bpb_order      The range edit is controlled by the following
                bitflags:
                1 = BRIDGE_GETRANGE_EDITOLD
                2 = BRIDGE_GETRANGE_STARTANCHORED
                4 = BRIDGE_GETRANGE_SHOWHIDDEN
```

If directed to EDITOLD, then getrange edits the range in startcol..., else starts at the current cursor position.

If STARTANCHORED then getrange begins with an anchored range, else single cell.

If SHOWHIDDEN, then 1-2-3 will show hidden columns hidden with /wch, else will hide those columns.

OUTPUT:

bpb_retcode	1 if successful 0 if user entered ESC, thus aborting selection. -1 if 1-2-3 is not loaded.
bpb_rangename	ASCIIIZ string is used selected by name, else '\0'
bpb_startcol	Coordinates of selected range.
bpb_startrow	
bpb_endcol	
bpb_endrow	

bridge_getrange_addr

This function returns the current coordinates for a named range.

INPUT:

bpb_funcode	BRIDGE_GETRANGE_ADDR (2)
bpb_rangename	ASCIIIZ string containing range name.

OUTPUT:

bpb_retcode	1 if successful 0 if range name not found. -1 if 1-2-3 is not loaded.
bpb_startcol	Coordinates of selected range.
bpb_startrow	
bpb_endcol	
bpb_endrow	

bridge_setrange_addr

This function sets the coordinates for a named range. If the range name already exists, its coordinates will be replaced. If the range does not already exist, it will be created.

INPUT:

bpb_funcode	BRIDGE_SETRANGE_ADDR (3)
bpb_rangename	ASCIIZ string containing range name.
bpb_startcol	Coordinates of selected range.
bpb_startrow	
bpb_endcol	
bpb_endrow	

OUTPUT:

bpb_retcode	1 if successful
	0 if failed (not enough room in 123 or invalid values)
	-1 if 1-2-3 is not loaded.

bridge_getrange_data

This function gets the data associated with a range of cells, which must be specified through coordinates. The data is exported from 1-2-3 into the client supplied buffer (bpb_buffer). Cells are copied until the entire range is exported, or the buffer overflows. The data is a stream of cell records that must be parsed by the client.

INPUT:

bpb_funcode	BRIDGE_GETRANGE_DATA (4)
bpb_startcol	Coordinates of range.
bpb_startrow	
bpb_endcol	
bpb_endrow	
bpb_order	Row order = 0 (a1,b1,a2,b2,etc) Column order = 1 (a1,a2,b1,b2,etc)
bpb_bufsize	Size in bytes of client supplied buffer.
bpb_buffer	Pointer to client supplied buffer. Segment must be the same as that of the bpb structure itself, which must be in the client's data segment.

OUTPUT:

bpb_retcode	Number of cells returned in buffer. 0 may indicate error in argument specification. -1 if 1-2-3 not loaded,
bpb_buffer	Stream of cell records.

bridge_setrange_data

This function sets the contents of a cell or range. The data is imported to 1-2-3 from the bpb_buffer field. The entire range must be contained in the RDB. The data is a stream of cell records which will be parsed by 1-2-3.

INPUT:

bpb_funcode	BRIDGE_SETRANGE_DATA (5)
bpb_startcol	Coordinates of range.
bpb_startrow	
bpb_endcol	
bpb_endrow	
bpb_order	Row order = 0 (a1,a2,b1,b2,etc) column order = 1 (a1,b1,a2,b2,etc) .
bpb_bufsize	Size in bytes of the cell stream found in bpb_buffer.
bpb_buffer	Pointer to client supplied buffer. Segment must be the same as that of the bpb structure itself, which must be in the client's data segment. Data must be a stream of cells to be parsed by 1-2-3.

OUTPUT:

bpb_retcode	number of cells successfully entered. Any number less than the full range indicates error, probably inadequate memory or invalid range specification. -1 if 1-2-3 not loaded.
-------------	--

bridge_recalc

This function tells 1-2-3 to recalculate the current worksheet.

INPUT:

bpb_funcode	BRIDGE_RECALC (6)
-------------	-------------------

OUTPUT:

bpb_retcode	1 if successful 0 if error -1 if 1-2-3 is not loaded.
-------------	---

bridge_get_cursor

This function gets the cell coordinates of the cursor.

INPUT:

bpb_funcode	BRIDGE_GET_CURSOR (7)
-------------	-----------------------

OUTPUT:

bpb_startcol	Coordinates of cursor.
bpb_startrow	
bpb_retcode	1 if successful. 0 if error.
	-1 if 1-2-3 is not loaded.

bridge_set_cursor

This function sets the cell coordinates for the cursor.

INPUT:

bpb_funcode	BRIDGE_SET_CURSOR (8)
bpb_startcol	Coordinates for cursor.
bpb_startrow	

OUTPUT:

bpb_retcode	1 if successful 0 if error -1 if 1-2-3 is not loaded.
-------------	---

bridge_redisplay

This function causes 1-2-3 to redisplay the worksheet, but does not redisplay the control panel or status area.

INPUT:

bpb_funcode	BRIDGE_REDISPLAY (9)
-------------	----------------------

OUTPUT:

bpb_retcode	1 if successful 0 if error -1 if 1-2-3 is not loaded.
-------------	---

bridge_celltype

This function returns the cell type of the specified cell.

INPUT:

bpb_funcode	BRIDGE_CELLTYPE (10)
bpb_startcol	Coordinates of cell.
bpb_startrow	

OUTPUT:

bpb_retcode	'B' if blank
	'I' if integer
	'N' if float
	'S' if ASCII string
	'F' if formula

bridge_calctype

This function returns the current calc type of 1-2-3.

INPUT:

bpb_funcode	BRIDGE_CALCYTPE(11)
--------------------	----------------------------

OUTPUT:

bpb_retcode	0 if 1-2-3 in manual mode.
	-1 if 1-2-3 in automatic mode.

From Software Design to Ordering ROM Cards

HP Supplied Development Tools

The purpose of this section is to introduce some tools Hewlett Packard is making available to help with the development of HP 95LX software. The tools are discussed in the order they would be used during the software development process. This process includes the following steps:

1. Prepare the program and data files.
2. If the program is to be system management compliant, the RAM resident portion needs to be converted from .EXE form to .EXM form.
3. If the program is to execute in place in ROM, the .EXM form will need to be converted to .XIP form and an .EXM loader will need to be created.
4. Finally, all the files needed for the ROM card will need to be combined into a ROM image file.

This is the all-inclusive process, and many applications can be prepared using only a subset of the steps. For example, if the application is to run under DOS and not be System-Manager-compliant then it does not need to set up an event handler that uses System-Manager routines. Or if your application will run in RAM like most PC applications then you will not need to convert your .EXE file to a .XIP file.

Power-Saving Suggestions

The HP 95LX has a low-power mode that greatly extends battery life. This is an idle state that the computer enters whenever you invoke INT 16h to test for a keypress (services 1 or 11h) or GET key (services 0 or 10h).

Idle State

When TEST for a keypress is called, the computer enters the idle state until the next hardware interrupt. Normally, this is the 18.2 Hz timer, that interrupts every 55 msecs. However, it could be any other hardware interrupt. Assuming that the timer interrupt is what causes the computer to exit from its idle state, the TEST for key routine can last at most for 55 msecs.

If an application program has a main loop that does some processing and then TESTs for a key down, the computer will automatically enter its idle state on each pass through the loop. The percentage of time the HP 95LX spends in the idle state depends on the amount processing apart from the TEST for key. To optimize battery life, you should minimize processing time in the program's main loop.

If GET key is called and no key is in the buffer, the HP 95LX enters its idle state almost continuously until a key is pressed.

Caution

The idle state is suppressed for 110 msec (two timer ticks) after a screen cursor movement or BIOS call to write to the display. Consequently, do not include these operations in the main loop of your program if you desire low-power mode.

Serial Port Power

HP 95LX battery life can be extended by approximately 10% by simply turning off power to the serial port—use Interrupt 15h service 4Ah. When serial port power is turned off, the HP 95LX can receive characters from the serial port, but it can not send characters.

Prepare the program and data files.

The first steps of a software project involve defining a project that will meet the users needs and will be compatible with the constraints of the target hardware. After the project has been defined and the design has been developed you will want to start writing the software. For small applications that will run in the HP 95LX's RAM you can proceed by compiling and linking software on your PC and downloading the .EXE file to your HP 95LX. For larger programs that will eventually run XIP from a ROM card, you will want to use a product like Soft-ICE from Nu-Mega and emulate the software on your PC. The output from this process will be an .EXE file and whatever datafiles are needed to run the program.

For system manager compliant software, you should take time to develop a good design to handle keyboard events, memory management and display output. All this activity should go through the system manager to DOS and the BIOS. It should be done by in a central top level area of your program. The *smhello.c* program included with the developer's software demonstrates one such method of interfacing the system manager to your application.

Important HP 95LX Considerations

1. Since the RAM of the HP 95 LX is limited, avoid linking in libraries that include unused entries.
2. Many C libraries cause startup or initialization code to be included in the final program. System manager application writers should avoid such libraries since such initialization code is usually incompatible with the HP 95LX system manager.

Converting from .EXE to .EXM

Standard .EXE files must be converted to standard .EXM files before they can be successfully loaded and run by the system manager. This conversion can be most easily made by using the *E2M.EXE* available in the HP 95LX Software Developer's package.

In order for the system manager to link your application to a hot key, you must include a line in your *apname.lst* file that points to the appropriate .EXM file. The format for each entry in your *apname.lst* file is:

path with file name, hot key scan code, identification string

For example, you could use the following entry to associate the *smhello* program with the Alt-S hot key:

c:\smhello.exm,1400,Hello Program

Converting Files from .EXM or .EXE to .XIP

If your application is to execute in place on a ROM card it must be converted to a standard .XIP file. Hewlett Packard is providing tools to support this conversion. For system manager compliant applications this tool is called *exe*; for DOS applications it is called *.exe*. These tools are also available in the software developer's package.

In addition to the .XIP file these tools create the source code for a stub program. The stub program needs to be included on your ROM card. The stub program will need to be edited to include the eventual ROM address of your .XIP program. Then it will need to be compiled and linked to create an .EXE file. For system manager compliant applications it should be converted to an .EXM file. The purposes of the .EXE or .EXM file are to provide these services:

1. File manager (.EXM case) or DOS (.EXE case) run your application by starting up the stub program. In particular, it is the stub program that should be listed in the *apname.lst* file.
2. The stub program will bank switch the .XIP file into high memory and set registers so the .XIP program will run successfully.
3. For system manager compliant applications the stub program should be the top level handler of keyboard, memory allocation, and display events.

The constraints on the .XIP program include:

1. The .XIP file must start on a 64K byte boundary on the ROM card. As mentioned the ROM address needs to be coded in the stub program. It also needs to be communicated to the ROM CARD File Ssystem Builder described in the next section.
2. The .XIP file cannot exceed 192K bytes.

ROM Image Builder

Purpose

The ROM Card File System Builder (ROMCFS) creates an image of a file system that can be placed on a Plug-in ROM card. The card can then be used as a file system or drive when plugged into the HP 95LX.

ROMCFS is invoked: ROMCFS *directory*

Where *directory* contains a configuration file (ROMIMAGE.CFG) and all files to be placed on the ROM card. <directory> may contain nested subdirectories that will also be added to the ROM card.

Glossary of Terms

Term	Explanation
PCMCIA	Personal Computer Memory Card International Association PCMCIA provides standards for memory card header information (see CIS in this glossary).
CIS	Card Information Structure. First sector on a device containing information based on the PCMCIA standard.
constrained file	A file that needs to start on a certain boundary. Usually one that will be mapped into RAM.
device	The hardware being created. For example, EPROM or OTP.
ROMCFS	ROM Card File System. An image of a file system to be placed on a device. The device can then be treated as a file system or another drive on certain products.

Overview

ROMCFS reads a configuration file. Using the configuration information it creates a file that is the image of a file system. Through subsequent means the image can be placed on a programmable device.

The directory specified on the command-line (see USAGE above) becomes the root directory on the file system. Any sub-directories are also added to the file system. When complete, the file system will contain:

1. PCMCIA CIS
2. boot sector
3. two File Allocation Tables (FAT)
4. root directory

5. files and sub-directories

Subsequent sections in this document may imply that the output of ROMCFS is a device with a file system on it; however, ROMCFS only creates a DOS file which is an image of a file system. The newly created file is suitable for transferring to a programmable device using other means.

The Configuration File for ROMCFS

The configuration file contains keywords and parameters. The keyword and associated parameter are separated by one or more spaces or tabs. Some reasonability checking is done on the parameters. Incongruities are listed on stdout. By design most of these are flagged as WARNING and processing continues. This allows the creation of some non-standard devices. Any error marked ERROR causes termination of the process.

Keywords and Parameters

Keyword	Description	Default	Required
ROM_SIZE	Size in Kbytes of the ROM being created.	0	R
RAM_SIZE	Size in Kbytes of the RAM being created.	0	
RAM_OFFSET	Offset to the beginning of RAM from 0.	ROM_SIZE	
DEVICE_SIZE	Size in Kbytes of the device being created; e.g., 1024 for a 1MB device.	ROM_SIZE+ RAM_SIZE	
ROM_NAME	Variable length name to be placed in the CIS.		
DEVICE_TYPE Kind of device being created: EPROM, OTR etc.		R	
DEVICE_SPEED	Speed of the device being created: e.g., 250 ns		R
VENDOR	Vendor's name		
MACHINE_CODE	Code for the machine for which the device is being created.	0	
<file> <bounds>	Name of a file that is to be placed on a specific boundary.		

Required Keywords and Parameters

The following keywords and parameters are required as the very minimum.

ROM_SIZE *n*

where *n* is the size of the ROM in Kbytes. Minimum value according to PCMCIA standards is 512. The maximum is 256MB (specified 262144 for Kbytes). The HP 95LX accepts a maximum size of 2MB.

DEVICE_SPEED *s*

If the speed is	then <i>s</i> is
250 nsec	1
200 nsec	2
150 nsec	3
100 nsec	4

DEVICE_TYPE *t*

If the device is a	then <i>t</i> is
masked ROM	1
OTP (One-time Programmable ROM)	2
EPROM	3
EEPROM	4
Flash EPROM	5

Optional Keywords and Parameters

DEVICE_SIZE *n*

n is specified in Kbytes. DEVICE_SIZE is assumed to be the sum of ROM_SIZE and RAM_SIZE. RAM_SIZE *n* *n* is the size in Kbytes of RAM to be added to the device. The RAM is not actually added; the value is placed in the CIS. RAM_OFFSET *n* *n* is the offset from the beginning of the device to the start of RAM. For RAM_SIZE > 0, RAM is assumed to start immediately following ROM. ROM_NAME *string* *string* is the name of the ROM to be placed in the CIS. Any ASCII character is valid. Maximum length is 60. VENDOR *string* *string* is the name of the company or individual producing the ROM code. Maximum size is 25.

MACHINE_CODE *c*

If the machine is	then <i>c</i> is
HP95LX	0

The default machine code is: 0.

filename *b*

filename is the name of a file in the root directory; *b* is the boundary specified in Kbytes on which *filename* is to be located on the device. *filename* is placed at the first appropriate boundary found which has enough contiguous clusters for the entire file. Multiple filename-boundary combinations can occur (maximum 64) in the configuration file.

These constrained files are placed on the device before any others.

Processing Specifics

The ROM Card File System builder uses the configuration file ROMIMAGE.CFG and user files to build ROMIMAGE.ROM: a DOS file containing the image of a file system.

ROMCFS changes directories to the one specified on the command-line. It expects to find the configuration file ROMIMAGE.CFG.

Some validation on the keywords and associated parameters is done. Incongruities result in either warnings or errors written to stdout. Warning messages are prefaced by "WARNING" and are non-fatal. Warnings usually result in the software making an assumption. Review these messages to insure that the desired value has been obtained. Error messages are preceded by "ERROR" and cause ROMCFS to terminate.

Appropriate sizes are calculated for the root directory and FATs based on the contents of the directory to be processed.

Any constrained files found in the configuration must exist in the current directory. At this point the current directory will become the root directory on the new file system. The constrained files are sorted by boundary in descending sequence. Each constrained file is written to the file system at the first appropriate boundary and in contiguous clusters.

After processing constrained files, ROMCFS does a "find" on the current directory. All files excluding ROMIMAGE.CFG, ROMIMAGE.ROM and constrained files are added to the file system. These files and sub-directories may be fragmented into 512-byte clusters to optimize the file system space.

Once the contents of the current directory have been added to the file system, contents of sub-directories are added.

ROMCFS extends the file system to the specified ROM_SIZE padding with zeros.

A checksum is computed on the entire ROM image and added to the CIS header.

Each time ROMCFS is run, it will re-build ROMIMAGE.ROM. All other contents of the user-supplied directory are unaltered.

Ordering ROM's from Epson

It is the ROM file image that needs to be transferred to ROM cards. Epson America should be contacted directly for instructions on ordering ROM's. Their Hewlett Packard ISV coordinator is:

David Rifkin
Epson America
20770 Madrona Avenue
Torrence, California 90509-2842 Telephone: (213) 782-5315

'HOPPER'
95LX SYSTEM CONTROLLER
External Reference Specification
Version 1.01

HEWLETT-PACKARD COMPANY

April 30, 1991

CONTENTS

INTRODUCTION.....	1
BLOCK DIAGRAM.....	3
PIN OUT.....	4
3.1 PIN Description.....	4
I/O ADDRESS SUMMARY.....	8
4.1 Programmable Interrupt Controller (8259 Compatible).....	8
4.2 Programmable Interval Timer (8254 Compatible).....	11
4.3 Programmable Peripheral Interface (Emulates 8255).....	12
4.4 MDA Adapter (PC MDA Compatible).....	13
4.5 Serial Port (8250 Compatible UART).....	14
4.6 Display Controller (HOPPER specific).....	17
4.7 Real Time Clock (HOPPER specific).....	17
4.8 Miscellaneous Registers (HOPPER specific).....	18
4.9 HOPPER Memory Configuration and Bank Switching.....	22
CPU BUS INTERFACE.....	24
5.1 Display Address Remapping.....	25
5.2 Internal Chip Selects.....	25
5.3 External Chip Selects.....	26
5.3.1 Wait State Registers 27	
5.3.2 Start Address Registers 27	
5.4 Write Enable Bits.....	27
5.4.1 Device Size Registers 28	
5.5 Bank Switching.....	28
5.5.1 BANK C and BANK D 29	
5.5.2 BANK E0 through BANK E3 29	
5.5.3 Limitations 30	
5.5.4 Bank Control Registers 30	
5.5.5 Example 31	
5.6 Take Over ROM/EPROMs.....	31
HOPPER DISPLAY CONTROLLER.....	32
6.1 Display Timing and Control Registers.....	32
6.1.1 MDA Registers 32	
6.1.2 HOPPER Display Control Registers 35	
6.2 System Memory Interface.....	37
6.2.1 Display Memory Organization 37	
6.3 Display Data Generation.....	38
6.4 Cursor Generation.....	39
6.5 Display Blank Mode.....	39
KEYBOARD CONTROL.....	41
7.1 Keyboard Hardware.....	41
7.2 Software Control.....	41

7.3	Hardware Reset.....	42
CARD DETECTION.....		43
8.1	Card Detect Register.....	43
8.2	Special Considerations.....	44
SERIAL COMMUNICATION.....		45
9.1	UART.....	45
9.2	IR Communication.....	45
	9.2.1 REDEYE format	48
	9.2.2 Software Controlled Mode	49
	9.2.3 Modulated Mode	49
	9.2.4 IR UART Modes	49
INTERRUPT CONTROL.....		51
10.1	8259 Interrupt Sources.....	51
10.2	Non-Maskable Interrupt.....	52
10.3	Interrupt Source Register.....	52
10.4	Wake-Ups.....	53
10.5	Enabling Interrupts.....	53
10.6	Special Considerations.....	53
PROGRAMMABLE INTERVAL TIMER.....		54
11.1	TIMER0.....	54
11.2	TIMER1.....	54
11.3	TIMER2.....	55
11.4	Timer Operation in Light Sleep.....	55
REAL TIME CLOCK TIMER.....		56
12.1	Functional Description.....	56
12.2	Special Considerations.....	56
12.3	Pre-Divider Outputs.....	56
PC COMPATABLE I/O REGISTERS.....		57
CRYSTAL OSCILLATORS.....		58
14.1	Low Frequency Oscillator.....	58
14.2	High Frequency Oscillator.....	58
CLOCK GENERATOR.....		59
TOUCH PANEL CONTROLLER.....		61
16.1	The A/D Converter Interface (ADCONT).....	61
16.2	Touch Panel Control (TPCONT).....	62
TONE GENERATOR.....		65
17.1	Hardware Description.....	65
17.2	Software Control.....	66
POWER MANAGEMENT.....		67
18.1	Static Test Condition.....	67

18.2	Deep Sleep.....	67
18.3	Light Sleep.....	67
18.4	Operating.....	68
18.5	Backup.....	68
18.6	System Resets.....	68
18.7	System Control Register.....	69
CONTRAST CONTROL VOLTAGE GENERATOR.....		72
SPECIAL HARDWARE CONSIDERATIONS AND HOPPER REV C.....		73
20.1	Pseudo-Static RAMs and the CPU Halt Instruction.....	73
20.2	Pseudo-Static RAMs and Hardware Reset.....	74
20.3	Display Cursor.....	74
20.4	TX Output.....	75
20.5	Timer Wakeups.....	75
20.6	Interrupt Source Register (ISR).....	75
20.7	The ON Key.....	76
20.8	RDY Timing.....	76
20.9	Speaker Power-Down.....	76
20.10	Keyboard Precharge and Reset.....	76
20.11	UP8250 Lockup.....	77
20.12	UP8250 Parity Enable.....	77
20.13	UP8250 Interrupts and Interrupt ID Register.....	77
20.14	UP8250 Receiver Error Bits.....	78
20.15	UP8250 Modem Status Register.....	78
20.16	UP8250 Break Reset.....	78
20.17	UP8250 Line Status Register.....	78
20.18	UP8250 Receiver Buffer Register.....	78
TESTING.....		80

LIST OF FIGURES

Figure 2.1.	HOPPER Block Diagram 0.12.....	3
Figure 5.1.	Bus Interface Block Diagram.....	24
Figure 15.1.	Clock Generator Block Diagram.....	59

CHAPTER 1 INTRODUCTION

This document contains a description of the 'HOPPER' IC. This chip is designed to be used in conjunction with an 80C88 CPU, a LCD module, and ROM/RAM chips, to implement a system with a moderate degree of PC compatibility. This chip is divided into the following functional areas:

1. An 8088 CPU Bus Interface. This block contains the functionality of the 8288 BUS CONTROLLER plus address latches, address/data transceivers, and internal/external chip enable decoding. The internal chip enable decoder supports programable address configuration for 5 external ROM/RAM devices.
2. A Display Controller capable of controlling an LCD module of up to 16 lines by 40 characters. The Display Controller supports a movable window into the 25 line by 80 character MDA standard. It also supports non-standard bit-mapped graphics of up to 128 rows by 240 columns.
3. A Keyboard Controller that interfaces directly to a keyboard switch matrix of up to 8 rows by 16 columns plus a separate, dedicated ON key. PC compatible key codes are generated by the BIOS.
4. A Card Detect Register that contains information on the presence and write protect status of up to 2 plug-in cards.
5. RS232 and infrared (IR) serial ports supporting communication via an 8250 compatible UART. IR communication with the REDEYE printer and HP calculators is also supported.
6. An 8259 compatible Programmable Interrupt Controller (PIC).
7. An 8254 compatible Programmable Interval Timer (PIT).
8. A Real Time Clock timer used by software to keep track of time and date and to implement alarms.
9. A set of PC compatible I/O registers for system configuration.

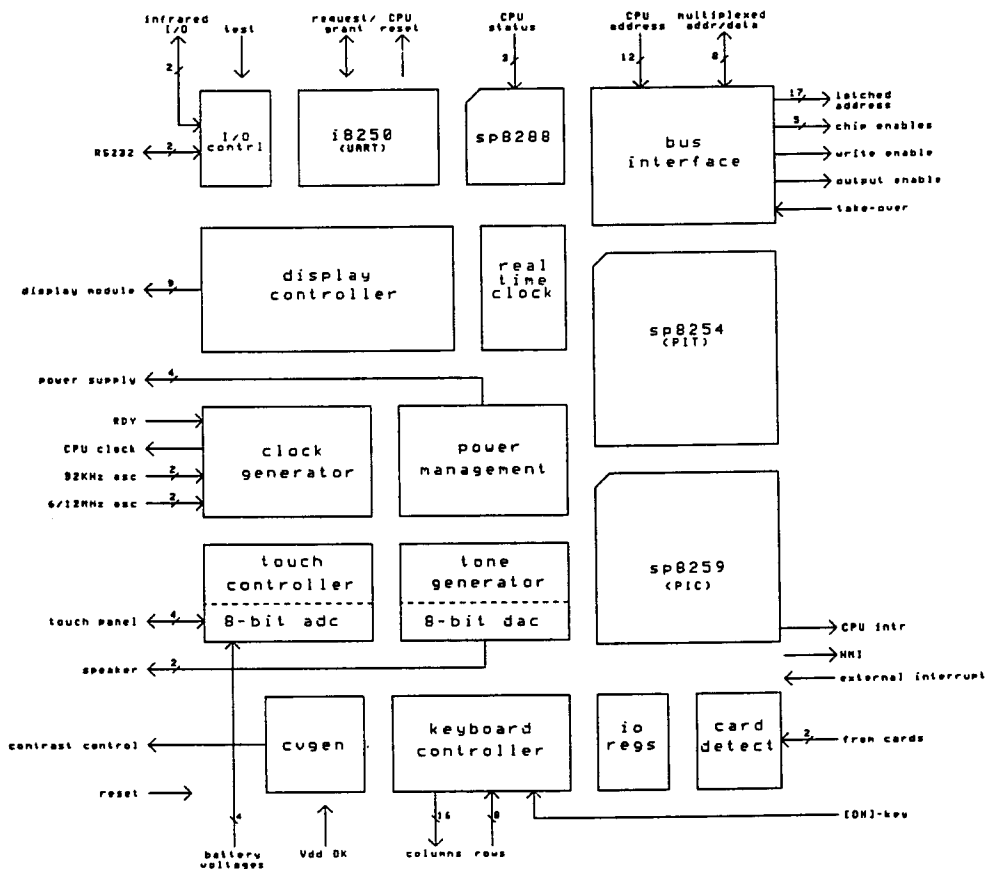
10. A low frequency (32.768KHz) quartz-crystal oscillator.
11. A high frequency (5.37MHz or 10.74MHz) quartz-crystal oscillator.
12. A Clock Generator that uses the output of the high frequency oscillator (HFO) to generate clocks for the CPU, the PIT, and the UART.
13. A Touch Panel Controller which includes an 8-bit A/D converter. This A/D converter is also used to monitor the voltage levels of up to 4 battery inputs.
14. An analog Tone Generator which includes an 8-bit D/A converter.
15. A Power Management circuit that controls the external switching power supply regulator. This external supply operates in low power, high power, and backup power modes. The power management block also includes a Power-On Reset (POR) circuit.
16. A Contrast Control Voltage generator to control the external LCD voltage generation. This circuit consists simple resistive network used to impliment a 4 bit D/A.

A chapter has been dedicated to each of these functional areas. Also included are chapters describing the chip BLOCK DIAGRAM, the chip PIN OUT, the I/O ADDRESSING, and TESTING.

CHAPTER 2 BLOCK DIAGRAM

The following is a simplified block diagram of the HOPPER chip.

Figure 2.1. HOPPER Block Diagram 0.12



<p>CHAPTER 3 PIN OUT</p>

3.1 PIN Description

PIN	#	TYPE	DESCRIPTION
-----	---	------	-------------

Power Supply:

VDD	7	I	Power Supply: 4.5 to 5.5 Volts.
GND	4	I	Ground: 0 Volts.
AVDD	2	I	Analog Power Supply.
AGND	2	I	Analog ground.

80C88 Interface:

CLK	1	I/O	System clock.
NS[0:2]	3	I	Processor status.
AD[0:7]	8	I/O	Multiplexed address/data lines.
MA[8:11]	4	I/O	System address lines. Source either from CPU or from HOPPER.
AP[12:15]	4	I	Processor address lines.
AS[16:19]	4	I	Multiplexed address/status lines.
INTR	1	O	Interrupt request.
NMI	1	O	Non-Maskable interrupt. Caused by either NVDDOK = 1 or a change on the CDT[0:1] inputs.
NRGO	1	I/O	Request/Grant pin for bus hold requests (required by display controller to take control of AD[0:7] and MA[8:11]).
PRES	1	O	Processor reset.

System RAM/ROM Interface:

MA[0:7]	8	O	LSB's of system address; normally latched from AD[0:7].
MA[12:20]	9	O	MSB's of system address. Four of these 8 bits are required to remap 4K portions of the system RAM, the others are latched from AS[16:19].
NWE	1	O	Read/Write enable; active low.
NCE[0]	1	O	Internal ROM chip enable; active low.
NCE[1]	1	O	Internal RAM chip enable; active low.
NCE[2]	1	O	Internal ROM/RAM chip enable; active low.
NCE[4:5]	2	O	Plug-in ROM/RAM chip enables; active low.
NOE	1	O	Output enable; active low. This pin also provides refresh timing for pseudo-static RAMs.
RDY	1	I	High true ready input from external memory devices. If this pad is pulled low during a memory access, CLK will be held high. There is a resistive pullup on this pad.
CDT[0:1]	2	I/O	Card detects; sense NWPOUT (Write protect output) of plug-in cards.

External LCD Row/Col Drivers:

DVEN	1	O	Display blanking output. This signal is actually a general purpose output used to control the display module's high voltage supply in JAGUAR.
YD	1	O	Data for row driver.
LOAD	1	O	Parallel load pulse.
DF	1	O	Frame inversion signal.
CP	1	O	Clock pulse. Data is shifted on the fall of this signal.
DD[0:3]	4	O	Data to display column drivers.

Keyboard:

KBO[0:15]	16	O	Keyboard drive.
KBI[0:7]	8	I	Keyboard sense.
ON	1	I	ON key input.

Other:

BAT[0:3]	4	I	Analog battery voltage inputs (main batteries, backup battery, and 2 card batteries).
NVDDOK	1	I	Low true Vdd OK input from external power supply. If this line goes high in operating or light sleep a NMI will be issued. If it is high at any other time it will force the system into backup mode.
ACIN	1	I	Input from the external power supply indicating that the AC adapter is plugged in.
BUV	1	O	Backup; driven high to force the external power supply into backup mode.
SOFF	1	O	System power off; driven high to turn off power to CPU, Display, ROM, and Card.
IOFF	1	O	I/O power off; driven high to turn off power to IR receiver chip.
TPH[0:1]	2	I/O	Touch panel lines. Driven high or float.
TPL[0:1]	2	I/O	Touch panel lines. Driven low or float.
SPK[0:1]	2	O	Differential speaker driver pins.
CCV	1	O	Contrast control voltage. This is a high impedance analog voltage output between GND and VDD.
IRO	1	O	LED output driver.
IRI	1	I	Infrared input sensor.
RX	1	I	Serial in.
TX	1	O	Serial out.

HXI, HXO	2	I/O	5.369318MHz quartz crystal connections.
LXI, LXO	2	I/O	32.768KHz quartz crystal connections.
XINT	1	I	External interrupt; pulling XINT high will wake the system if asleep and cause an interrupt. This is a wired-or input with a passive pulldown.
NTKO	1	I	Take over; if NTKO is shorted to ground by a plug-in card, all system ROM accesses (NCE[0]) will be redirected to NCE[4].
NTEST	1	I	Test mode input; pulled low to force the part into test mode.
NRES	1	I	Reset; active low.

TOTAL	132		
-------	-----	--	--

CHAPTER 4 I/O ADDRESS SUMMARY

The following is a summary of the HOPPER I/O addresses. Where at all possible the I/O addresses are compatible with the original IBM PC.

4.1 Programmable Interrupt Controller (8259 Compatible)

I/O Address	R/W Mode	Description
-------------	----------	-------------

0020h	R	PIC Interrupt Request/In-Service Registers programmed by Operation Command Word 3 (OCW3): Interrupt Request Register, where: bits 7-0 =0 no active request for the corresponding interrupt line =1 active request for the corresponding interrupt line Interrupt In-Service Register, where: bits 7-0 =0 the corresponding interrupt line is not currently being serviced =1 the corresponding interrupt line is currently being serviced
0020h	W	PIC Initialization Command Word 1 (ICW1) when bit 4 is one: bits 7-5 =0 not used bit 4 =1 required to select this command word bit 3 =0 edge triggered mode bit 2 =1 not used bit 1 =1 single mode (no ICW3 needed) bit 0 =1 ICW4 needed

I/O	R/W	Address	Mode	Description
-----	-----	---------	------	-------------

0021h	W	PIC ICW2 and ICW4 in sequential order after ICW1 written to Port 0020h:		
-------	---	---	--	--

ICW2, where:				
bit 7-3	=00001	address lines A7-A3 of base vector address for interrupt controller		
bit 2-0	=0	reserved		

ICW4, where:				
bits 7-5	=0	not used		
bits 4	=0	no special fully nested mode		
bits 3-2	=11	buffered mode/master		
bit 1	=0	normal EOI		
bit 0	=1	8086/8088 mode		

0021h	R/W	PIC interrupt mask register (OCW1), where:		
-------	-----	--	--	--

bit 7	=0	enable HOPPER RTC interrupt		
bit 6	=0	enable HOPPER XINT pin interrupt		
bit 5	=0	enable HOPPER IR input interrupt		
bit 4	=0	enable UART interrupt		
bit 3	=0	enable HOPPER keyboard and touch panel interrupts		
bit 2	=0	enable miscellaneous HOPPER interrupts		
bit 1	=0	enable PC compatible keyboard interrupt (checked by BIOS)		
bit 0	=0	enable timer0 interrupt		

0020h	W	PIC OCW2 when bit 4 is zero and bit 3 is zero, where:		
-------	---	---	--	--

bits 7-5	=000	rotate in automatic EOI mode (clear)		
	=001	non-specific EOI		
	=010	no operation		
	=011	specific EOI		
	=100	rotate in automatic EOI command (set)		
	=101	rotate on non-specific EOI command		
	=110	set priority command		
	=111	rotate on specific EOI command		
bits 3-4	=00	required to select this command word		
bits 2-0		interrupt request to which the command applies		

I/O Address	R/W Mode	Description
----------------	-------------	-------------

0020h	W	PIC OCW3 when bit 4 is zero and bit 3 is one, where:
-------	---	---

bit 7	=0	reserved
bits 6-5	=00	no operation
	=01	no operation
	=10	reset special mask
	=11	set special mask
bits 4-3	=01	required to select this command word
bit 2	=0	no poll command
	=1	poll command
bits 1-0	=00	no operation
	=01	no operation
	=10	read interrupt request register on next read at Port 20h
bits 1-0	=11	read interrupt in-service register on next read at Port 20h

4.2 Programmable Interval Timer (8254 Compatible)

I/O Address	R/W Mode	Description
0040h	R/W	PIT counter 0
0041h	R/W	PIT counter 1
0042h	R/W	PIT counter 2
0043h	W	PIT control word, where: bits 7-6 =00 select Counter 0 =01 select counter 1 =10 select counter 2 =11 read back command bits 5-4 =00 counter latch command =01 read/write counter bits 0-7 only =10 read/write counter bits 8-15 only =11 read/write counter bits 0-7 first, then bits 8-15 bits 3-1 =000 mode 0 select =001 mode 1 select =x10 mode 2 select =x11 mode 3 select =100 mode 4 select =101 mode 5 select bit 0 =0 binary counter 16 bits =1 BCD counter

4.3 Programmable Peripheral Interface (Emulates 8255)

The HOPPER chip does not contain a PPI as such. Instead it contains a group of 3 I/O registers that are configured to behave as the PC's PPI. These registers are referred to as the PC Compatible I/O Registers. The definitions of the configuration switch bits should be chosen to be PC compatible.

I/O Address	R/W Mode	Description
-------------	----------	-------------

0060h R/W PPI Input Port A:

If port 0061h bit7=0:
bits 7-0 scratch location for keyboard scan code

If port 0061h bit7=1:
bits 7-0 scratch location for SW1 configuration switch settings

0061h R/W PPI Output Port B:

bit 7	=0 read/write keyboard scratch byte (0060h)
	=1 read/write SW1 scratch byte (0060h)
bit 6	=0 disable keyboard
bit 5	=0 ignored, reads 0 (enable I/O check)
bit 4	=0 ignored, reads 0 (enable RAM parity check)
bit 3	=0 read high switch (0062h)
	=1 read low switch (0062h)
bit 2	=0 ignored, always reads 0
bit 1	=1 enable speaker data
bit 0	=1 enable timer 2 gate (to speaker)

0062h R/W PPI Input Port C:

bits 7-6 =0 unused (read only)
bit 5 timer 2 output (read only)
bit 4 =0 unused (read only)

If port 0061h bit 3=0:
bits 3-0 scratch location for 4 MSBs of SW2 configuration switches

If port 0061h bit 3=1:
bits 3-0 scratch location for 4 LSBs of SW2 configuration switches

4.4 MDA Adapter (PC MDA Compatible)

I/O Address	R/W Mode	Description
03B4h	W	MDA Index register (only 0Ah, 0Eh, and 0Fh supported)
03B5h	R/W	Indexed MDA registers
03B8h	R/W	MDA mode control register
03BAh	R	MDA status register

4.5 Serial Port (8250 Compatible UART)

I/O		R/W	Address Mode Description	
03F8h	W		UART transmitter holding register, which contains the character to be sent. Bit 0, the least significant bit, is sent first.	
			bits 7-0	contains data bits 7-0 when Divisor Latch Access Bit (DLAB) = 0 (03FBh)
03F8h	R		UART receiver buffer register, which contains the received character.	
			bits 7-0	contains data bits 7-0 when DLAB=0
03F8h	R/W		UART divisor latch, low byte. Both divisor latch registers store the baud rate divisor.	
			bits 7-0	bits 7-0 of divisor when DLAB=1
03F9h	R/W		UART divisor latch, high byte, where:	
			bits 7-0	bits 15-8 of divisor, when DLAB=1
03F9h	R/W		UART interrupt enable register when DLAB = 0. Allows the four controller interrupts to enable the chip interrupt output signal.	
			bits 7-4	=0 reserved
			bit 3	=1 not used (modem status interrupt enable)
			bit 2	=1 receiver line status interrupt enable
			bit 1	=1 transmitter holding register empty interrupt enable
			bit 0	=1 received data available interrupt enable

I/O Address	R/W Mode	Description
-------------	----------	-------------

03FAh	R	UART interrupt ID register. Information about a pending interrupt is stored here. When ID register is addressed, the highest priority interrupt is held and no other interrupts are acknowledged until the CPU services that interrupt.
-------	---	---

bits 7-3 =0 reserved

bits 2-1 Identity of the pending interrupt with the highest priority

=11	receiver line status interrupt: highest priority
=10	received data available: second priority
=01	transmitter holding register: third priority
=00	invalid (modem status interrupt)

bit 0 =0 interrupt pending, contents of register can be used as a pointer to the appropriate interrupt service routine

=1 no interrupt pending

03FBh	R/W	UART Line Control Register, where:
-------	-----	------------------------------------

bit 7	=0	Receiver buffer, transmitter holding or interrupt enable register access (DLAB)
bit 7	=1	divisor latch access
bit 6	=1	set break enabled (output = space)
bit 5		stick parity
bit 4	=0	odd parity
	=1	even parity
bit 3	=1	parity enable
bit 2	=0	1 stop bit
	=1	1.5 stop bits if bits 1-0 = 00, else 2 stop bits
bits 1-0	=00	5 bit word length
	=01	6 bit word length
	=10	7 bit word length
	=11	8 bit word length

I/O Address	R/W Mode	Description
03FCh	R/W	UART Modem Control Register (modem control is not implemented) bits 7-4 =0 reserved bit 3 =1 enable UART interrupt bits 2-0 =0 reserved
03FDh	R	UART Line Status Register, where: bit 7 =0 reserved bit 6 =1 transmitter shift and holding registers empty bit 5 =1 transmitter holding register is empty bit 4 =1 break interrupt bit 3 =1 framing error bit 2 =1 parity error bit 1 =1 overrun error bit 0 =1 data ready
03FEh	R	UART Modem Status Register (modem status is not implemented, these bits are hard-programmed to always return the values shown) bit 7 =0 Carrier Detect (changed to read a one on Rev C) bit 6 =0 bit 5 =0 Data-Set-Ready (changed to read a one on Rev C) bit 4 =0 Clear-To-Send (changed to read a one on Rev C) bits 3-0 =0
03FFh	R/W	UART scratch pad register

4.6 Display Controller (HOPPER specific)

I/O Address	R/W Mode	Description
D300h	R/W	bits 7-0 LSBs of display window start address pointer
D301h	R/W	bits 7-4 unused bits 3-0 MSBs of display window start address pointer
D302h	W	bit 7 unused bits 6-0 window row size offset
D303h	W	bits 7-6 unused bits 5-0 horizontal window size
D304h	W	bit 7 unused bits 6-0 vertical window size
D305h	R/W	bits 7-4 row time control bits 3 unused bit 2 =0 display blank bit 1 =1 display on bit 0 =0 alpha mode =1 graphics mode

4.7 Real-Time Clock (HOPPER specific)

I/O Address	R/W Mode	Description
D306h	R/W	bits 7-0 bits 7-0 of the 16-bit counter value
D307h	R/W	bits 7-0 bits 15-8 of the 16-bit counter value

4.8 Miscellaneous Registers (HOPPER specific)

I/O Address	R/W Mode	Description
E300h	R/W	<p>Test Mode Register</p> <p>bit 7 reserved</p> <p>bit 6 state of AMPPD for test (read only) (Rev C only)</p> <p>bit 5 =1 enable watchdog circuit (Rev C only)</p> <p>bit 4 state of ACIN pad (read only)</p> <p>bit 3 =1 force PIT clock to run in light sleep (normally enabled only when CPU is running)</p> <p>bit 2 =1 force HOPPER output disable test mode</p> <p>bit 1 should be set if a 10.74MHz crystal is used (not supported at this time)</p> <p>bit 0 =0 reset has occurred; this bit will be cleared by a system reset; it is initialized to a one by a warmstart</p>
E301h	R/W	<p>First Byte of System Control Register</p> <p>bit 7 =1 RTC interrupt/wakeup enable</p> <p>bit 6 =1 XINT pad interrupt/wakeup enable</p> <p>bit 5 =1 IR circuit interrupt/wakeup enable</p> <p>bit 4 =1 UART interrupt/wakeup enable</p> <p>bit 3 unused</p> <p>bit 2 =1 I/O on</p> <p>bit 1 =1 CPU shutdown (set only)</p> <p>bit 0 =1 writing a one twice in succession will cause a reset (write only)</p>
E302h	R/W	<p>Second Byte of System Control Register</p> <p>bit 7 =1 touch panel interrupt/wakeup enable</p> <p>bit 6 =1 keyboard interrupt/wakeup enable</p> <p>bit 5 =1 low voltage RX pad interrupt/wakeup enable</p> <p>bit 4 =1 display cursor interrupt/wakeup enable</p> <p>bit 3 =1 module pulled interrupt/wakeup enable</p> <p>bit 2 =1 low power interrupt/wakeup enable</p> <p>bit 1 =1 timer 1 interrupt/wakeup enable</p> <p>bit 0 =1 timer 0 interrupt/wakeup enable</p>

I/O Address	R/W Mode	Description
E303h	R/W	Interrupt Source Register (read/clear only) bit 7 =1 touch panel interrupt bit 6 =1 keyboard service request bit 5 =1 low voltage RX interrupt bit 4 =1 display cursor update request bit 3 =1 module pulled interrupt bit 2 =1 low power interrupt bit 1 =1 timer 1 interrupt bit 0 unused
E304h	R/W	Card Detect Register bit 7 =1 port 1 has card inserted (read only) bit 6 =1 port 0 has card inserted (read only) bit 5 =1 port 1 is write enabled (read only) bit 4 =1 port 0 is write enabled (read only) bits 3-2 unused bit 1 =1 update bits 7-4 every millisecond (write only) bit 0 =1 update bits 7-4 continuously (write only)
E305h	W	Tone Buffer Register (D/A value)
E306h	R	A/D Value Register (0 = 0 Volts, 255 = 5 Volts)
E307h	R/W	A/D Control Register (ADCONT) bits 7-6 unused bit 5 =1 conversion completed (read only) bit 4 =1 start conversion (write only) bit 3 =0 power down mode (write only) bits 2-0 Input Voltage Channel Select, where: =000 system battery =001 backup battery =010 port 1 battery =011 port 2 battery =100 touch pad x-axis low side =101 touch pad x-axis high side =110 touch pad y-axis low side =111 touch pad y-axis high side

I/O Address	R/W Mode	Description
E308h	R/W	<p>Touch Panel Control Register (TPCONT)</p> <p>bits 7-6 unused</p> <p>bit 5 =1 indicates pad has been touched in standby mode (read only)</p> <p>bit 4 =1 touch pad interrupt enabled</p> <p>bit 3 =1 unselected axis is precharged to ground</p> <p>bit 2 =0 select y-axis</p> <p> =1 select x-axis</p> <p>bit 1 =1 touch panel precharge mode</p> <p>bit 0 =1 touch panel standby mode</p>
E309h	R/W	<p>Tone Control / Contrast Register (TCCONT)</p> <p>bits 4-7 D/A value for contrast control voltage pin</p> <p>bits 3-2 digital beeper volumn control (0 = min, 3 = max)</p> <p>bit 1 =0 tone circuit power down mode</p> <p>bit 0 =1 tone buffer empty (read only)</p>
E30Ah	R/W	<p>IR Format Register (IRFMAT)</p> <p>bit 7 =1 disables IR pad current regulation</p> <p>bit 6 =1 enable LED buffer empty interrupt</p> <p>bit 5 =1 LED buffer full (read only)</p> <p>bit 4 =0 single pulse transmission mode</p> <p> =1 multiple pulse transmission mode</p> <p>bit 3 =0 modulate using 32.768KHz</p> <p> =1 modulate using baud rate generator</p> <p>bit 2 =1 IR UART communication mode</p> <p>bit 1 =1 modulated communication mode</p> <p>bit 0 =1 REDEYE transmit mode</p>
E30Bh	R/W	<p>IR Transmit / Receive Register</p> <p>bit 7 state of the IRI pin (read only)</p> <p>bit 6 =1 enable IR interrupt on IRI = 1</p> <p>bit 5 =1 IR event has occurred, must be cleared by software</p> <p>bit 4-3 unused</p> <p>bit 2 =1 gate output when in modulated communication mode</p> <p>bit 1 =0 transmit "off" half-bit in REDEYE format</p> <p> =1 transmit "on" half-bit in REDEYE format</p> <p>bit 0 =1 turn on LED driver, used for software controlled transmissions</p>

I/O Address	R/W Mode	Description
E30Dh	W	Writing anything to this location will end keyboard precharge
E30Eh	W	Low byte of keyboard output register (KBO[0-7])
E30Fh	W	High byte of keyboard output register (KBO[8-15])
E30Eh	R	Keyboard input register (KBI[0:7])
E30Fh	R	bits 7-1 =0 unused bit 0 state of [ON] key

4.9 HOPPER Memory Configuration and Bank Switching

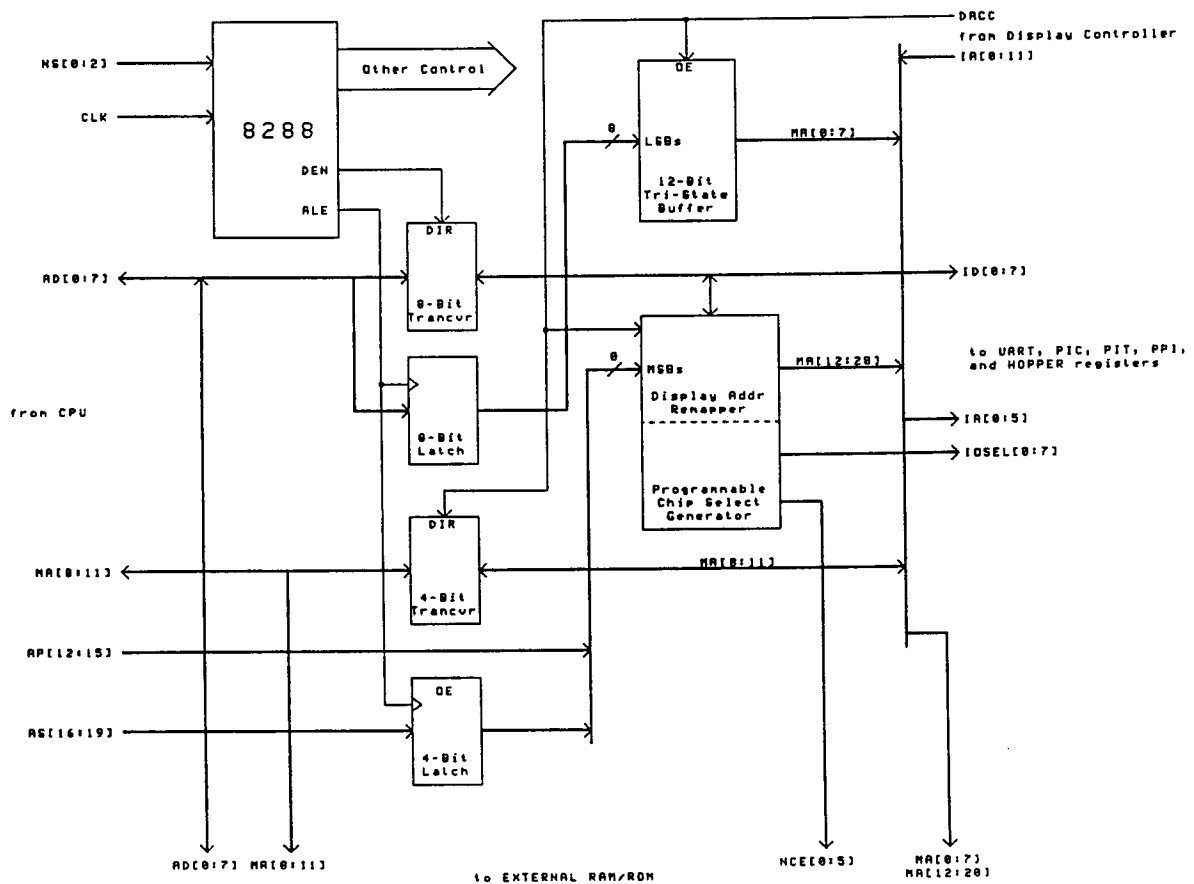
I/O Address	R/W Mode	Description	
F300h	R/W	bit 7 bits 6-5 bits 4-3 bits 2-0	unused 2-bit NCE[2] wait state value 2-bit NCE[1] wait state value 3-bit NCE[0] wait state value
F301h	R/W	bits 7-5 bits 4-2 bits 1-0	3-bit NCE[5] wait state value 3-bit NCE[4] wait state value reserved
F302h	R/W	bits 7-1 bit 0	NCE[2] starting address (MSBs) unused
F303h	R/W	bits 7-1 bit 0	reserved unused
F304h	R/W	bits 7-1 bit 0	NCE[4] starting address (MSBs) unused
F305h	R/W	bits 7-1 bit 0	NCE[5] starting address (MSBs) unused
F308h	R/W	bit 7 bits 6-0	=0 NCE[0] write protected (default) unused
F309h	R/W	bit 7 bits 6-1 bit 0	=1 NCE[1] write enabled (default) NCE[1] size register unused
F30Ah	R/W	bit 7 bits 6-1 bit 0	=0 NCE[2] write protected (default) NCE[2] size register unused
F30Bh	R/W	bits 7-1 bit 0	reserved unused
F30Ch	R/W	bit 7 bits 6-1 bit 0	unused NCE[4] size register unused
F30Dh	R/W	bit 7 bits 6-1 bit 0	unused NCE[5] size register unused

I/O Address	R/W Mode	Description	
F310h	R/W	bits 7-1 bit 0	Bank E0 frame select (MSBs of device address) unused
F311h	R/W	bits 7-3 bits 2-0	unused Bank E0 device select =000 select NCE[0] (ROM) =001 select NCE[1] (system RAM) =010 select NCE[2] (system RAM/ROM) =011 reserved =100 select NCE[4] (slot 0) =101 select NCE[5] (slot 1)
F312h	R/W	bits 7-1 bit 0	Bank E1 frame select (MSBs of device address) unused
F313h	R/W	bits 7-3 bits 2-0	unused Bank E1 device select (see F311h)
F314h	R/W	bits 7-1 bit 0	Bank E2 frame select (MSBs of device address) unused
F315h	R/W	bits 7-3 bits 2-0	unused Bank E2 device select (see F311h)
F316h	R/W	bits 7-1 bit 0	Bank E3 frame select (MSBs of device address) unused
F317h	R/W	bits 7-3 bits 2-0	unused Bank E3 device select (see F311h)
F318h	R/W	bits 7-3 bits 2-0	Bank C frame select (MSBs of device address) Bank C device select (see F311h)
F319h	R/W	bits 7-3 bits 2-0	BANK D frame select (MSBs of device address) BANK D device select (see F311h)
F31Fh	R/W	bits 7-0	MDA buffer address mapping register (MSBs of NCE[1] device address)

CHAPTER 5 CPU BUS INTERFACE

The HOPPER chip includes a CPU Bus Interface block that generates internal address and data busses, internal and external device select signals, as well as read/write and other control signals. This section includes an Intel 8288 compatible Bus Controller. A block diagram of the Bus Interface follows.

Figure 5.1. Bus Interface Block Diagram



The 8288 compatible bus controller latches and decodes the status lines, S[0:2], from the 80C88 CPU. Its outputs include memory read/write signals, I/O register read/write signals, as well as control signals for the address latches, data bus transceivers, and PIC.

The address lines can be sourced from either the CPU or from the Display Controller. The Display Controller will gain control of the CPU address bus using the Grant/Request sequence. It will then act as a bus controller to read up to 80 bytes of display data from system memory into its refresh buffer.

5.1 Display Address Remapping

The data for the display is stored in the device connected to NCE[1] (system RAM) at the device address 1nnXXXh, where "nn" is the value stored in the 8-bit MDA buffer address register located at I/O location F31Fh. The display data can be accessed at either the MDA address space or at it's actual address.

When the 5 most significant bits of the CPU address are pointing into the 32KB MDA address space starting at B0000h, the display address remapper replaces these address bits with the contents of its 8-bit display address register (with MA[20]=1). The contents of this register is always used when addresses are being sourced from the Display Controller. The display address register must be initialized to point into a 4KB section of RAM (NCE[1]) as defined by the external chip select configuration.

5.2 Internal Chip Selects

The I/O Select logic decodes 6 device select signals for the internal I/O registers. The addresses within which these select signals are active are hard programmed as follows:

SIGNAL	I/O ADDRESS	DESCRIPTION
IOSEL[0]	0020h-0021h	PIC (8259)
IOSEL[1]	0040h-0043h	PIT (8254)
IOSEL[2]	0060h-0063h	Miscellaneous PC compatible IO registers.
IOSEL[3]	03B0h-03BFh	MDA registers
IOSEL[4]	03F8h-03FFh	UART (8250)
IOSEL[5]	D300h-D307h	HOPPER display control and real time clock.
IOSEL[6]	E300h-E30Fh	Miscellaneous HOPPER control/status.
IOSEL[7]	F300h-F31Fh	HOPPER memory configuration and bank switching.

5.3 External Chip Selects

SIGNAL	FUNCTIONAL DESCRIPTION
NCE[0]	ROM chip select. It is hard-addressed (non-programmable) at two separate address ranges of 64KB each. The top 64KB of the ROM chip is addressed at F0000h to FFFFFh. The next 64KB of the ROM chip is addressed at A0000h to AFFFFh. The remaining portion of a ROM that is larger than 128KB can be accessed using bank switching as described in the next section.
NCE[1]	RAM chip select with a starting address of 00000h. The size is programmable from 8KB to 512KB (6-bits) and defaults to 8KB on reset. The 4KB display data buffer is stored in this device.
NCE[2]	RAM/ROM chip select intended for built-in memory. The size registers are programmable to any power of 2 from 8KB to 512KB (6-bits). The 7-bit starting addresses must be programmed to an address that is a multiple of the size.
NCE[3]	Reserved for future expansion.
NCE[4:5]	RAM/ROM chip selects intended for either built-in or plug-in memory. The size registers associated with these chip selects are programmable to any power of 2 from 8KB to 512KB (6-bits). The 7-bit starting addresses may be programmed to any 8KB boundary. This requires 2 7-bit adders.

Unused upper-order address lines are driven high during an access. This means that the accesses will be made to the upper portion of devices that are larger than their configured size. The remaining portion can be accessed via bank switching as described in the following section. The upper portion was chosen since plug-in's with both RAM and ROM configure the RAM in upper addresses. This RAM can be used to extend system memory and the ROM can then be accessed via bank switching.

If 2 chip select configurations are programmed to overlap at some addresses, the lowered number chip select takes precedence. Therefore, in order for NCE[2:5] to be generated, the starting address register for each must be configured (written to) after a system reset.

5.3.1 Wait State Registers

The number of wait states required by each device can be programmed using the Wait State Registers. This register defaults to the maximum number of wait states and should be programmed to maximize the system performance allowed by the access speeds of the devices present.

Addr	Bits	Signal	Reset Value	Comments
F300h	[0-2]	NCE[0]	111b	0 to 7 wait states
	[3-4]	NCE[1]	11b	0 to 3 wait states
	[5-6]	NCE[2]	11b	0 to 3 wait states
	[7]	unused	x	
F301h	[0-1]	unused	xx	reserved
	[2-4]	NCE[4]	111b	0 to 7 wait states
	[5-7]	NCE[5]	111b	0 to 7 wait states

5.3.2 Start Address Registers

The Start Address Registers specify the 7 MSBs of the CPU address (A13 to A19) at which the device address 0 is mapped. Address bit 20 is always a one except when using bank switching as described later in this chapter. The LSB of these registers is not used.

Addr	Signal	Reset Value	Comments
-	NCE[0]	E0h	non-programable ROM start
-	NCE[1]	00h	non-programable RAM start
F302h	NCE[2]	00h	must be a multiple of the size
F303h	none	XXh	reserved
F304h	NCE[4]	00h	any 8KB boundary
F305h	NCE[5]	00h	any 8KB boundary

5.4 Write Enable Bits

The MSB of the the Device Size Registers described in the following section serve as write enable bits for the NCE[0] through NCE[2] signals. NCE[1] defaults to one (enabled). NCE[0] and NCE[2] default to zero (disabled). The write enable for NCE[4] and NCE[5] is controlled by the card detect circuitry. The write enable bits should be set for RAM and clear for ROM.

5.4.1 Device Size Registers

The Device Size Registers specify the size of the device that is mapped into CPU addresses. The 6-bit size may be configured to any power of 2 between 8KB and 512KB. The size value can be calculated as a 7-bit value as follows:

$$\text{Size Register Value} = (\text{Device Size} / 4096) - 2$$

This formula will yield even values for valid size values. This is necessary since the LSB is unused. The MSB is used as a write enable flag as described in the previous section.

Addr	Signal	Reset Value	Comments
-	NCE[0]	1Eh	non-programable (ROM size = 128KB)
F309h	NCE[1]	80h	8KB default with write enable bit set
F30Ah	NCE[2]	00h	any power of 2 from 8KB to 512KB
F30Bh	none	XXh	reserved
F30Ch	NCE[4]	00h	any power of 2 from 8KB to 512KB
F30Dh	NCE[5]	00h	any power of 2 from 8KB to 512KB

If a device is configured with a size that is larger than its actual size several images of the device will appear in the allotted address space.

NOTE: After a reset, NCE[1:5] are all configured to a size of 8KB starting at 00000h. With this configuration, only NCE[1] will be accessed in this address range since it will take precedence over the others.

5.5 Bank Switching

HOPPER implements a bank switching scheme that allows up to 2 MByte of memory to be addressed on each of the 6 chip enable signals. This allows up to 12MByte to be accessed in 1MByte address space of the 80C88 CPU.

The bank switching scheme supports two 64KB banks and four 16KB banks. It is intended to allow support of LIM EMS 3.2 and also to support the need for several 64KB pages of ROM code to be swapped in and out of upper memory. Bank switching is totally independent of the external chip select configuration defined above.

Each of the 6 banks is accessed at a fixed location in the CPU address space. There are IO registers associated with each bank. These registers must be initialized to specify the section of memory that will be mapped into the bank.

5.5.1 BANK C and BANK D

Pages C and D of the CPU address space (CPU addresses C0000h to CFFFFh and D0000 to DFFFFh) are each set up as a 64KB bank. These banks are referred to as BANKS C and D. Each has a single 8-bit IO register used to select the memory mapped into the bank. The 3 LSBs (bits 0-2) of each register contain the "Chip Select (CS)" value that is used to select the device to be addressed. The 5 MSBs contain the "Frame Select (FS)" value that specifies the 5 MSBs of the 21-bit address to the device, thereby selecting a 64KB section.

The 3-bit CS values of BANK C and D can select any of the 6 possible external memory devices. The CS value is decoded into a chip select (NCE) as follows:

CS VALUE	DEVICE SELECTED
000	NCE[0] - Built-In system ROM.
001	NCE[1] - Built in RAM hard-configured to start at 00000h.
010	NCE[2] - Built-In ROM or RAM.
011	none - Reserved
100	NCE[4] - Plug-In slot 0.
101	NCE[5] - Plug-In slot 1.

5.5.2 BANK E0 through BANK E3

Page "E" of the CPU address space (E0000h to EFFFFh) is divided into four 16K banks. These banks are referred to as BANK E0 through BANK E3. Each of these banks has 2 IO registers that select the memory mapped into the bank. One register contains the 3 bit "Chip Select (CS)" value. The other contains the 7 bit "Frame Select (FS)" value that specifies the 7 MSBs of the 21-bit address to the device, thereby selecting a 16KB section.

The 3-bit CS values of BANK E0-3 are defined as for BANK C and D above.

5.5.3 Limitations

Devices larger than 2MB (21 address bits) are not supported by bank switching. There is also no provision to prevent one section of memory to be accessed both in the normal CPU address space and via bank switching.

5.5.4 Bank Control Registers

The Bank Control Registers are read/write registers. The function of these registers is disabled until the register has been initialized.

Addr	Name	Bits	Comments
F310h	E0 Frame Select	[1-7]	7 MSBs of device address
F311h	E0 Chip Select	[0-2]	3-bit device select code
F312h	E1 Frame Select	[1-7]	7 MSBs of device address
F313h	E1 Chip Select	[0-2]	3-bit device select code
F314h	E2 Frame Select	[1-7]	7 MSBs of device address
F315h	E2 Chip Select	[0-2]	3-bit device select code
F316h	E3 Frame Select	[1-7]	7 MSBs of device address
F317h	E3 Chip Select	[0-2]	3-bit device select code
F318h	Bank C	[3-7] [0-2]	5 MSBs of device address 3-bit device select
F319h	Bank D	[3-7] [0-2]	5 MSBs of device address 3-bit device select
F31Fh	MDA Buffer	[0-7]	A[12:19] of NCE[1] device address. A[20]=1.

5.5.5 Example

As an example of bank switching, assume that a 512KB ROM is tied to NCE[0]. The last pages of this ROM (ROM addresses 70000h to 7FFFFh) would be configured at F0000h to FFFFFh. The second to the last page (ROM addresses 60000h to 6FFFFh) would be configured at A0000h to AFFFFh. If an application needs 2 contiguous pages beginning at ROM address 20000h, it can configure those into CPU pages C and D. The following register settings would accomplish this plus map the starting addresses of a plug-in in slot 0 into the 4 sections of BANK E.

```
Bank C = 10h      (FS=00010, CS=000)
Bank D = 18h      (FS=00011, CS=000)
```

```
E0-3 Chip Select = 4h
E0 Frame Select   = 0h
E1 Frame Select   = 1h
E2 Frame Select   = 2h
E3 Frame Select   = 3h
```

5.6 Take Over ROM/EPROMs

The NTKO pin in the card port is normally left floating. A ROM card or EPROM card may be manufactured as a take over device if the NTKO pin is connected to ground. This pin is sampled each time the system is turned on. If it is low it will cause a swap in functionality between the NCE[0] and NCE[4] pads. This makes the plug-in card look like the system ROM, and the system ROM look like plug-in port 0.

The following rules apply to write protection :

1. The write protect switch on the plug-in card always write protects the card (NCE[4]).
2. The write protect bit at address F308h is associated with NCE[0] in normal operation. In take-over mode this bit is associated with NCE[4].
3. NCE[0] can not be hardware write-protected in take-over mode.

CHAPTER 6

HOPPER DISPLAY CONTROLLER

The display controller in the HOPPER chip interfaces to off-the-shelf LCD drivers to drive a dot matrix LCD up to 240x128 dots in size. The actual display size to be driven is software programmable.

The display controller is designed to be compatible with the IBM Monochrome Display Adapter when operating in alpha mode. The IBM MDA will display 25 lines of 80 characters each. The HOPPER display controller supports displays up to 16 lines of 40 characters. A means is provided to move this smaller (16x40) window around within the full (25x80) display. The HOPPER display controller also supports a non-standard bit-mapped graphics mode.

The major functional areas of the display controller are described in this chapter.

6.1 Display Timing and Control Registers

This area contains all of the I/O registers necessary to be compatible with the IBM MDA. It also contains the I/O registers necessary for setting the display width and height, window location in MDA memory, and control (such as text vs. graphics mode). The I/O registers are defined as follows:

6.1.1 MDA Registers

The following MDA registers have identical functions as those in an IBM MDA.

I/O Address	R/W Mode	Description
-------------	----------	-------------

03B4h	W	Address index register for 6845 CRTIC used in MDA mode.
-------	---	---

03B5h	W	6845 data register for transfer of data to internal 6845 register pointed to by the address index register.
03B8h	W	MDA CRT control register:
		bit 3 Video enable bit
		bit 5 Enable blink bit
03BAh	R	MDA CRT status register:
		bit 0 Horizontal drive status
		bit 3 Video data status

The following internal 6845 registers are implemented. Except where noted, their function is identical to the original 6845 registers.

6845	R/W	
Register	Mode	Description

R10	W	Cursor definition register:
		bits 0-4 Cursor start: Defines the starting scan line of the alpha cursor. If value is equal to or greater than 7, cursor is an underline. Cursor end register is not implemented. Cursor end is always row 7.
		bits 5-6 Cursor control: Enables or disables cursor.
		bit 7 not used
R14	R/W	Cursor address register:
		bits 0-5 high order bits of cursor address
		bits 6-7 not used
R15	R/W	Cursor address register:
		bits 0-7 low order bits of cursor address

6.1.2 HOPPER Display Control Registers

The following are registers unique to the HOPPER display controller.

I/O Address	R/W Mode	Description
D300h	R/W	Window Start Address bits 7-0 LSBs of display window start address pointer. Window start address points to the character in MDA memory which is to appear in the upper, left corner of the HOPPER display.
D301h	R/W	Window Start Address bits 7-4 unused bits 3-0 MSBs of display window start address pointer. Window start address points to the character in MDA memory which is to appear in the upper, left corner of the HOPPER display.
D302h	W	Window Row Size Offset bit 7 unused bits 6-0 Text Mode: For 40 character display, value should be 80. This will add 40 characters (81 bytes) to end of current row address to obtain address for beginning of next row. Graphics Mode: Can be set to any desired number of bytes. If contiguous memory is desired, value should be one.
D303h	W	Horizontal Window Size bits 7-6 unused bits 5-0 Text Mode: Contains the number of characters (minus 1) to be displayed. For a 40 character display, value should be 39. Graphics Mode: Contains the number of 16-bit words (minus 1) to be displayed. For a 240 column display, value should be 14.

I/O Address	R/W Mode	Description
-------------	----------	-------------

D304h	W	Vertical Window Size bit 7 unused bits 6-0 Contains number of scan lines (minus 1) in display. For a 128 line (16 character) display, value should be 127.
-------	---	--

D305h	R/W	Row Time Control bits 7-4 Value in register determines the time to update one row in the display. The combination of the value in this register and the vertical window size register determines the refresh rate of the display. The contents of bits 7-4 make up a constant named RT. The system LFO clock (32768 Hz) is divided by RT to obtain a start of new row clock. The value of RT is determined as follows: Bit 7 is the most significant bit, bit 5 is the least significant bit. If bit 4 is set, 1/2 will be added to the value of Bits 7-5 to determine the value of RT. RT is set to 3.5 after a hard reset. bit 3 unused bit 2 =1 Display is active and not blanked. =0 Display is active but blanked. bit 1 =1 Display refresh is on. =0 Display refresh is off. bit 0 =0 alpha mode =1 graphics mode
-------	-----	--

6.2 System Memory Interface

The display memory is physically located in system memory. This section of the display controller acts like a dedicated DMA controller in that it will request the system bus, drive the system address bus, buffer one row of display refresh data, and then release the bus.

At the beginning of each row time, the display controller will request the system bus from the cpu if necessary. In graphics mode this occurs every row. In alpha mode, this is only necessary every 8 rows since the data represents ascii characters which are implemented in an 8-row high font. After the bus is granted, the display controller will begin accessing system memory and filling the refresh buffer until the horizontal window size is reached. The display controller will then release the bus back to the cpu.

At this time, the data will be read from the refresh buffer and shifted to the external column drivers.

The major function of the system memory interface is to generate the system memory addresses from which the display data is obtained. For the first row in the display, the first address is loaded from the window start address register. This address is incremented for each successive memory access until the horizontal window size is reached. The window row size offset is then added to the current address to obtain the first address for the next row. Each of the remaining rows in the display are refreshed in the same manner.

In order to keep the degradation of system performance down, the system memory interface runs at the HFO clock frequency. The memory access time is two HFO clock cycles.

6.2.1 Display Memory Organization

There are 4K bytes of system memory reserved for display memory. The display memory is organized differently depending on whether the display is in alpha or graphics mode.

In alpha mode, the memory is organized into character cells containing two bytes each. Byte 0 of each cell contains an 8 bit character code and byte 1 contains attribute information for that character. The character cells are arranged in rows of 80 cells (160 bytes) each with a total of 25 rows.

In graphics mode, each bit in display memory corresponds to a

pixel on the display. The first byte corresponds to the 8 pixels on the left side of the first row of the display. The next byte corresponds to the next 8 pixels on the first row, etc. Within a byte, bit 7 maps to the left most pixel and bit 0 maps to the right most pixel.

6.3 Display Data Generation

Once the display refresh buffer is filled, the display data must be generated from the buffer.

In alpha mode, the first byte of data in a word is used to address a character rom containing 256 characters in a 6x8 character cell. The output of the character rom is the 6 bits of data used to build the current row of the addressed character. The second byte in the buffer contains attribute information and modifies the character as follows:

BIT:7-0	DESCRIPTION
b000x000	No display
b000x001	Underline character
b000x111	Normal characters.
b111x000	Inverse video characters.
b:	If set, character will blink.
x:	Intensity bit, ignored.

The character blink (and cursor blink) rate is set by the VSCLK (Very Slow CLK) signal from the timer block. The blink rate is not software adjustable.

In graphics mode, the data in the refresh buffer is shifted out to the column pads with no modification. No attributes of any kind are supported.

6.4 Cursor Generation

In alpha mode, the cursor is controlled and generated in the same way that the 6845 CRTC generates a cursor. The cursor address is contained in two registers. This address corresponds to the character location where the cursor will appear. The starting scan line for the cursor can be adjusted by writing to the cursor start register. Since the character cell is 8 scan lines deep, this register can contain a value from 0 to 7. The 6845 also has a register to set the cursor end. In the HOPPER display controller, the cursor end is always set in hardware to line 7 or the bottom line in the character cell.

Cursor blink is enabled/disabled by two bits in the cursor start register as described above.

6.5 Display Blank Mode

The display blank bit in HOPPER at address D305h goes directly to a pad on HOPPER. In the 95LX system, it is being used to provide a way to blank the display by turning off the external display drivers. The display controller is still active and refreshing the display. The purpose of this bit is to provide a way to power-up

and power-down the display module while preventing any high-voltage dc bias on the LCD. To power-up the module, software should wait at least 1ms after turning on the DON bit before enabling this bit. On power-down, the display should be blanked at least 1ms before the DON bit is cleared.

This mode can also be used to save some power during data communications or other activities where the CPU must be powered on but the display need not be visible to the user.

CHAPTER 7 KEYBOARD CONTROL

The HOPPER IC contains a keyboard control block. This block along with the proper software control will enable a PC compatible keyboard to be implemented. The keyboard consists of a 16 by 8 matrix of output lines and input lines. When a key is depressed, one input line is shorted to one output line. To sense this connection the output line must be driven high and the input line pulled low by a weak pull down. The connected input line will then be pulled high by the output line. All control of keyboard scans is done by software.

7.1 Keyboard Hardware

The keyboard hardware contained in the HOPPER chip consists of an input register, output register, and precharge control circuit. Each bit in the output register controls one output keyboard line. When this bit is set to one, the keyboard line must be pulled high by a strong pullup. If a zero output is desired, the keyboard line must first be precharged low and then held there by a weak pulldown. The pulldown must be weak to limit the current that can occur if two output lines become shorted together. This could happen if two keys on the same input line are pressed at the same time. The input lines must first be precharged low by the strong pulldown and then held there by the pulldown resistor. If a key is pressed, the line will then be pulled high by the corresponding output line. The weak pulldown in the input pad can be disabled by setting OD to a 1 for test purposes.

7.2 Software Control

The output register bits are located at register addresses E30Eh(low order 8 bits) and E30Fh(high order 8 bits). The input register bits are located at register address E30Eh. The status of the ON key can be read as bit 0 of register address F30Fh.

A keyboard precharge will occur anytime the output register is written. This precharge lasts until ended by a write to address E30Dh.

When all keys are up, the keyboard should be placed in standby

mode. To do this, software should write the appropriate value, normally all ones, to the output register; wait an appropriate amount of time for precharge; then write any value to address E30Dh to end precharge. Once in standby mode, an interrupt will occur when a key is pressed.

After a key is pressed, a keyboard scan must be executed. To do this, each output bit must be driven high one at a time and the state of the input register checked at each bit. A software keyboard interrupt (09h) must then be generated after the proper keycode has been placed in the keycode register (address 60h). The keyboard must be continually scanned at a determined interval until all keys have been released. This must be done to determine if more keys have been depressed or if a key has been released. If a key has been pressed or released, a proper interrupt must be generated as before. After all keys are released, the keyboard again can be placed in Standby mode.

Keyboard interrupts must be enabled by setting bit 6 of 061h in the PPI.

7.3 Hardware Reset

The keyboard hardware will also enable a hardware reset of the system. To initiate a hardware reset, three input lines, KBI[6], KBI[7], and ON, must be pulled high at the same time. These three lines will be debounced in hardware to prevent stray resets from occurring. Software will have no control over this feature, although they will be able to read these keys separately. When these keys are all pressed simultaneously, the machine will be reset.

CHAPTER 8 CARD DETECTION

The Card Detect Circuit (CDR block) uses the CDT[0:1] pins to sample the write protect outputs of plug-in cards. It can determine if a card is plugged in and if it is writable. A low level on the CDT pins indicates that a card is present and writable, a high indicates that a card is present and write protected, and a float indicates that no card is present. When enabled the card detect circuit cycles through floating the CTD pads, passively driving them low, and passively driving them high to determine the states of the pads.

8.1 Card Detect Register

A one byte I/O register located AT E304h controls the circuit and shows its status:

Bit	Reset Value	R/W Mode	Description
[0]	-	W	RCDT - Run Card Detect. If RCDT and ECDT are set then the card detect logic runs continuously, updating its status every 122uS. This mode is intended to be used while the CPU is running. In this mode the CDT pads are fighting with a cards write protect output 30uS every 122uS.
[1]	-	W	ECDT - Enable Card detect. If only ECDT is set then the card detect logic runs updating its status every 1mS. This mode is inteded to be used in light sleep. In this mode the CDT pads are fighting with a cards write protect output no more then 30uS every millisecond.
[2]	-	W	RSTMP - Reset Module Pulled. Writing a one to this bit will reset the module pulled condition. This should be done each time a module pulled interrupt occurs.
[3]	X	-	unused

[4]	0	R	POW - High when port 0 is writable. When low all writes to NCE[4] are disabled.
[5]	0	R	P1W - High when port 1 is writable. When low all writes to NCE[5] are disabled.
[6]	0	R	POC - High when port 0 has a card inserted.
[7]	0	R	P1C - High when port 1 has a card inserted.

The moduled pulled interrupt status bit, MPI, is located in the interrupt status register. When enabled, the card detect logic will set this bit and cause an IRQ2 interrupt when it senses any change in the card status (i.e. any change in the states of POW, P1W, POC, or P1C bits). Since this interrupt is shared with other sections, the MPI bit must be read to determine if the card detect was the source of the interrupt. The MPI bit must be cleared before another interrupt will occur.

8.2 Special Considerations

1. The status bits (P1C etc) are initialized to zero. Therefore, a spurious module pulled interrupt will be generated if a card is already plugged in when the circuit is first enabled.
2. When the system enters deep sleep power is removed from the card ports. This will cause the the CDT pad to transition from pulled high to pulled low if connected to a card that is write protected. A spurious interrupt will occur if the card detect circuit is enabled.
3. If a single card contains both RAM and ROM, the RAM chip enable should be tied to the pin normally used for card enable. Thus writes can be disabled to the RAM. However, since an auxillary chip enable pin must be used for the ROM chip enable, ROM writes must be disabled using the write enable bit for the corresponding chip enable.

CHAPTER 9 SERIAL COMMUNICATION

The serial communication portion of the HOPPER IC contains both wired RS232 and IR transmit and receive capabilities. The RS232 UART ports use an 8250 cell to control transmit and receive. The IRO output port can be used for two types of communication, IR transmit and REDEYE. REDEYE is used to transmit data to an infrared REDEYE printer port. The IR communications portion uses both the IRO output to transmit and the IRI input pin to receive data. These ports are used for wireless communication using infrared light. The 8250 and the IR ports can be used simultaneously to implement a wireless infrared UART mode.

9.1 UART

The serial UART block is implemented using an 8250 compatible macro cell. A 1.8 MHz clock will be supplied for operation of this block. The UART is addressed from 3F8h to 3FFh. RS232 drive and receive circuits are provided off chip.

The 1.8MHz UART clock is available in operating and light sleep modes. The UART clock should be disabled to save power when the UART is not in use. This is done by setting the 8250 baud rate divisor to zero.

9.2 IR Communication

The IR communication block enables the HOPPER IC to have wireless communication using an external infrared LED and IR receive circuit. The IR communication block supports 5 separate communication formats. These formats are REDEYE, Software controlled communication, Modulated communication, IR UART with both single pulse and multiple pulse communication. To control the IR transmission of these formats, the hardware uses 2 control registers, the IRCNT register and the IRFMAT register. The IRFMAT register at I/O register address E30Ah contains control bits that choose which format is chosen for IR communication. These bits are as follows:

Bit	Name	Description
0	RED	This bit when set activates REDEYE transmit mode. It turns on the REDEYE transmit hardware and sets it to a state where it is waiting for input from software.
1	MDLTE	This bit when set activates Modulated communication mode. It turns on the modulation source and allows the MDLD bit in the IRCNT register to control the output of a modulated waveform.
2	IRURT	This bit when set activates IR UART communication mode. This bit disconnects the 8250 from the RS232 pins and connects it to the IR communication block. Software after setting the PMOD and MDSEL bits in this register, just transmits and receives using 8250 as though it were connected to the RS232 ports.
3	MDSEL	This bit is used to select the modulation source for both IR UART mode and Modulation communication mode. If this bit is 0, the 32kHz low frequency clock is chosen as the modulation source. If it is set to 1, the 8250 baud rate generator 16x clock is used for the modulation source. This allows the modulation source to be set a 38kHz to be remote control compatible.
4	PMOD	This bit is used in IR UART mode to select between single pulse transmission and multiple pulse transmission. If it is set to 0, a single pulse of duration equal to a half cycle time of the modulation source will be transmitted for a 0 output bit. If it is set to 1, a pulse train of the modulation frequency will be used to transmit a 0.
5	LBF	Led Buffer Full. This bit is used in REDEYE mode to indicate that the contents of the LBR bit have not yet been transmitted and should not be written at this time. Writing to the LBR automatically sets this bit. This bit is cleared when the LBR is transferred to the REDEYE formatter.

6	ELBE	Enable Interrupt on LBR bit Empty (LBF clear). If this bit is set and LBF is clear, an IR interrupt will occur.
7	UNREG	This bit is used to test the HPIRO pad. Setting it high disables regulation of the pad current.

The IRCNT register at I/O register address E30Bh contains bits that are used to transmit a bit or waveform out on the IRO pad. The IRCNT register also allows software to receive IR data in any of the formats that can be transmitted. The contents are as follows:

Bit	Name	Description
0	LED	This bit is used to turn on the IR LED connected to the HPIRO pad. It is used for software controlled IR transmission. When it is set to a 1, the IR LED is turned on.
1	LBR	This bit contains the half-bit to be transmitted in REDEYE format. Write a one to send an "on" half-bit or write a zero to send an "off" half-bit.
2	MDLD	This bit is used for software to output a serial waveform to be modulated by the chosen modulation source. This allows compatibility with remote control format.
5	IRE	IR Event. This bit is set by a logic low voltage on the IRI pin. It is set to indicate that an IR event has occurred. Once set, software must reset this bit.
6	EIRI	Enable IR interrupt. An IR interrupt will occur if this bit and the IRE bit are both set.
7	IRI	IR Input pin. This bit allows software to monitor the state of the IRI pin. It is a read only bit.

Using the last three bits, software can receive each of the transmission formats described later. Also, if IR UART mode is set, software can receive data using the 8250 the same as it would in RS232 mode.

9.2.1 REDEYE format

The REDEYE portion consists of the RED, LBF, and ELBE bits in the IRFMAT register, the LBR bit in the IRCNT register, the REDEYE formatter, and the IRO LED pin. The LED pin has an open drain device and thus may be driven low or tristated only. When driven low the drain current is somewhat regulated by a feedback circuit. The LBF and LBR bits form a double buffered handshake mechanism that allow automatic REDEYE half-bit formatting and pacing. An interrupt mechanism is provided to indicate completion of each half-bit.

The REDEYE printer requires 15-bit frames of a precise format. Each bit of the frame consists of two half-bits. The duration of each half-bit is 14 cycles of the 32768 kHz crystal oscillator. The half-bit is considered to be "on" if the LED is pulsed 6-8 times (out of the 14 possible) at the 32768 kHz rate. HOPPER's REDEYE port uses 8 pulses. The format of a complete REDEYE frame is shown below:

Start-bits	Three half-bits "on-on-on".
Hamming-bits	Four pairs of half-bits.
Data-bits	Eight pairs of half-bits. Each of the four hamming and eight data bits are encoded with two half-bits. A "one" data or hamming bit is encoded by "on-off" and a zero is encoded by "off-on".
Stop-bits	Three half-bits "off-off-off". This is the minimum idle time required between frames.

The ELBE, RED, LBF, and LBR bits are cleared at reset. The REDEYE port also uses a formatter which is turned off whenever RED is cleared. Software initiates a half-bit transmission by writing a bit to LBR. This automatically sets the LBF flags in IRFMAT register and starts the state machine. The state machine transfers the bit from LBR into the formatter and clears LBF. If ELBE is set, this will cause an IR interrupt indicating that it is safe to write the next half-bit to LBR. The state machine then times the half-bit for 14 counts of the 32768 Hz crystal oscillator. If the bit in the formatter is a one, the LED is pulsed for the first eight of the 14 counts. Otherwise the LED is left off. If after the 14 counts LBF is clear, the state machine will return to its idle state of waiting for LBF. Otherwise it will immediately transfer the next half-bit and start timing it.

When LBF is clear and ELBE is set, an IR interrupt will occur. When the state machine clears LBF, software has 13 counts of the oscillator to write the next bit to LBR. Otherwise the length of the half-bits will not be correct.

Through-put:

$32768 / 14 = 2340.6$ baud (half-bits/sec)

$32768 / 28 = 1170.3$ bps (bits/sec)

REDEYE Frame Length:

$1.5 \text{ start} + 4 \text{ Hamming} + 8 \text{ data} + 1.5 \text{ stop} = 15 \text{ bits}$

REDEYE Thru-put:

$1170.3 / 15 = 78.02$ cps

9.2.2 Software Controlled Mode

The LED bit in IRCNT register is provided for software generated IR formats. This bit is OR-ed with the output of the REDEYE formatter, and the other IR format outputs. Therefore, two IR formats may not be used simultaneously.

Due to LED current limitations, the LED output driver duty cycle must be limited to a time average of 29%. The duty cycle is automatically limited to $1/2 \times 8/14$ or 28.6% by the REDEYE formatter. The format of a full REDEYE frame yields a duty-cycle of only 14.3%. If a different format is used (by using the LED bit) software must limit the duty-cycle.

9.2.3 Modulated Mode

The MDLD bit in the IRCNT register can be used by software to output any custom modulated waveform desired. To output a waveform, software must first set the MDLTE bit in the IRFMAT register and choose the modulation source using the MDSEL bit. If the 8250 baud rate 16x clock is chosen, its frequency must be set to the desired modulation frequency. Once this has been accomplished, software can set and clear the MDLD bit at desired to emulate the envelope of the output waveform. Whenever MDLD is one, pulses will be output of a 50% duty cycle for the given modulation source. As before, care must be taken not to exceed the 29% communication duty cycle.

9.2.4 IR UART Modes

The 8250 may be used for half duplex IR communication of limited baud rate. To use this mode the IRURT bit must be set in the IRFMAT register. When this bit is set, the 8250 is disconnected from the RS232 ports and connected to the IR communication block. When using this mode, software must first choose the transmission format. The two possible formats are single pulse mode and multiple pulse mode. If the PMOD bit is 0, single pulse mode is chosen. In this mode, a single pulse of one half cycle of the

modulation source is transmitted for a 0. In multiple pulse mode, a train of pulses of the modulation source is transmitted for a 0. In both modes, a 1 is transmitted as no pulses. As in Modulated mode, the modulation source again must be chosen. If the baud rate of 2400 baud is chosen, choosing the 8250 16x clock will give you a modulation rate of 38 kHz.

After this is set up, software can use the 8250 to communicate as though it were still connected to the RS232 ports.

CHAPTER 10 INTERRUPT CONTROL

The HOPPER interrupt control circuitry includes a Intel 8259 compatible Programmable Interrupt Controller (PIC) and additional support circuitry. The support circuitry is located in the PWR block and provides wakeup timing and interrupt enable bits. For additional information on these features please see the chapter entitled "POWER MANAGEMENT".

10.1 8259 Interrupt Sources

The 8259 PIC supports 8 vectored priority interrupts. These interrupt sources are defined as follows:

INPUT	INT	SOURCE
IRQ0	08h	PIT Timer 0 (same as PC).
IRQ1	09h	Unused (PC Keyboard). INT09h calls are made by the BIOS for PC compatibility.
IRQ2	0Ah	Timer 1, display cursor update request, low voltage RX pad input (PC reserved).
IRQ3	0Bh	Keyboard and Touch Panel (PC COM2).
IRQ4	0Ch	UART (PC COM1).
IRQ5	0Dh	IR input (XT fixed disk).
IRQ6	0Eh	External XINT pin (PC diskette).
IRQ7	0Fh	Real Time Clock timer underflow (PC LPT1). The 8259 will default to this interrupt vector if the signal requesting interrupt has gone away before the interrupt acknowledge cycle. These spurious interrupts should be handled gracefully.

10.2 Non-Maskable Interrupt

Low Power Interrupt (LPI) and Module Pulled Interrupt (MPI) will cause a CPU NMI to occur. The NMI service code must do a deep sleep shutdown immediately if the LPI bit of the Interrupt Source Register is set. If the MPI bit is set, it should be immediately cleared to re-enable NMIs. This is necessary since the NMI occurs on a rising edge.

10.3 Interrupt Source Register

The interrupt source register (ISR) is provided in order to allow software to individually identify and acknowledge interrupts that are shared among several sources.

The individual bits of the ISR will be set when an interrupt is requested from its corresponding interrupt source. Another interrupt will not occur from this source until the bit has been cleared by writing a zero.

Writing a one to an ISR bit will have no effect. This is an important feature. To avoid missing interrupts when writing to the ISR, all bits that are not to be affected should be written to a one.

The interrupt source register is physically located in the PWR block and is mapped at I/O address E303h.

Bit	Reset Value	R/W Mode	Description
[0]	x	-	unused
[1]	0	R/W	TI1 - TIMER 1 interrupt.
[2]	0	R/W	LPI - Low power interrupt.
[3]	0	R/W	MPI - Module pulled interrupt.
[4]	0	R/W	DCI - Display cursor update request.
[5]	0	R/W	RXI - Low voltage RX pad interrupt.
[6]	0	R/W	KBI - Keyboard service request.
[7]	0	R/W	TPI - Touch Panel interrupt.

10.4 Wake-Ups

All interrupt sources that are active in deep sleep or light sleep can cause the CPU to wake up to service the interrupt. This feature is discussed in the chapter entitled "POWER MANAGEMENT".

10.5 Enabling Interrupts

Each interrupt source has a separate enable bit. This is discussed in the chapter entitled "POWER MANAGEMENT".

10.6 Special Considerations

1. The XINT pad can be enabled to cause a wake-up. However this signal must still be high when the CPU is up and running (50ms hardware delay plus software delay) in order to be recognized. This constraint actually applies to all non-shared interrupts, IRQ4 through IRQ7, since they are not latched by the ISR.

<p>CHAPTER 11 PROGRAMMABLE INTERVAL TIMER</p>

The HOPPER Programable Interval Timer (PIT) is completely compatible with the Intel 8254. It contains 3 independent 16-bit counters.

11.1 TIMER0

The PC dedicates TIMER0 to generating time of day interrupts every 54.9 milliseconds. The HOPPER implimentation of TIMER0 is the same as on the PC.

SIGNAL	CONNECTION

CLK0	1.193182MHz (nominal)
GATE0	VDD (always enabled)
OUT0	IRQ0

11.2 TIMER1

The PC dedicates TIMER1 to generating DMA requests for dynamic RAM refresh. This is not needed in a HOPPER system. In the HOPPER system TIMER1 is a general purpose timer. It's output is or'ed with other sources to cause an IRQ2 interrupt. It is intended to be used for generating keyscan interrupts (see the chapter entitled KEYBOARD).

SIGNAL	CONNECTION

CLK1	1.193182MHz (nominal)
GATE1	VDD (always enabled)
OUT1	causes IRQ2 interrupt

11.3 TIMER2

The HOPPER connection of TIMER2 is the same as on the PC.

SIGNAL	CONNECTION

CLK2	1.193182MHz (nominal)
GATE2	Bit-0 of port 61h
OUT2	Tone generation circuit and bit-5 of port 62h

11.4 Timer Operation in Light Sleep

In order to save power, the 1.19MHz timer clock is normally stopped anytime the system is in light sleep. This can be avoided by setting bit 3 of E300h, LST. The timer clock does not operate in deep sleep.

<p>CHAPTER 12 REAL TIME CLOCK TIMER</p>

12.1 Functional Description

The REAL TIME CLOCK (RTC) TIMER consists of a 16 bit read/writeable counter that is decremented once per second. The HOPPER 32.768KHz crystal oscillator generates frequency is divided by a 15 bit pre-divider to produce the 1Hz timer clock. This clock is accurate to approximately 2 minutes per month.

The RTC timer is always enabled to run. A reset will clear the timer value and pre-divider to all zeros.

A level 7 interrupt (INT 0Fh) will occur anytime the timer's most significant bit (MSB) is a one. This will occur when the timer underflows. A wakeup will occur prior to the interrupt if the system is in light or deep sleep. The timer will continue to decrement after underflow. The maximum time that can be set is 2^{15} seconds or 9.1 hours.

12.2 Special Considerations

Special care is taken in the circuit design to prevent timer values from being corrupted by a decrement occurring during reads and writes. Even so, it is recommended that all read values be verified against a second read, and all write values be verified by a read.

12.3 Pre-Divider Outputs

The following signals are taken off of the timer pre-divider:

1. F16KHZ - Interrupt timing in IRCOM, KBD, and TOUCH. D/A sample output control in TONE.
2. F1KHZ - State timing in CDR (Card Detect).
3. F128HZ - Power-up delay in PWR and debounce in KBD.
4. F16HZ - Amp power down control in TONE
5. F1HZ - Cursor blink in DISP.

<p style="text-align: center;">CHAPTER 13 PC COMPATABLE I/O REGISTERS</p>

The I/O Register block contains 3-bytes of PC compatible I/O registers (8255 PPI). These registers are defined as follows:

PORT	BIT(S)	MODE	DESCRIPTION
60h	0-7	R/W	If port 61h bit7=0 : Scratch location for keyboard scan code (see chapter entitled KEYBOARD).
	0-7	R/W	If port 61h bit7=1 : Scratch location for SW1 configuration switch settings.
61h	0	R/W	TIMER 2 gate.
	1	R/W	Speaker data (see chapter entitled TONE GENERATOR).
	2	R/W	Ignored, always reads 0.
	3	R/W	Select source/destination for port 62h bits 0-3
	4-5	R/W	Ignored, always read 0.
	6	R/W	0 = disable keyboard
	7	R/W	Select source/destination for port 60h
62h	0-3	R/W	If port 61 bit3=0 : Scratch location for 4 MSBs of SW2 switch settings.
	0-3	R/W	If port 61 bit3=1 : Scratch location for 4 LSBs of SW2 switch settings.
	4	R	Unused (reads 0).
	5	R	TIMER 2 output.
	6-7	R	Unused (read 0).

CHAPTER 14 CRYSTAL OSCILLATORS

The HOPPER chip includes a 2 crystal oscillators. Each require a pair of 20pF loading capacitors as well as a quartz crystal of the appropriate frequency.

14.1 Low Frequency Oscillator

The low frequency oscillator operates at 32.768KHz and is used to generate timing for the real time clock, the card detect circuit, and the display controller. The low frequency oscillator can not be disabled.

14.2 High Frequency Oscillator

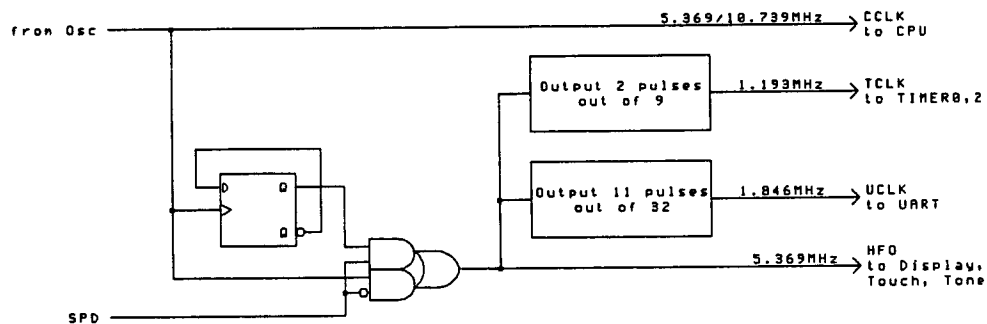
The high frequency oscillator is used to generate system timing for the CPU and peripherals. It will accept either a 5.37MHz or a 10.74MHz quartz crystal. The high frequency oscillator is active in light sleep and in operating mode.

NOTE: Operation with a 10.74MHz crystal is not supported at this time.

CHAPTER 15 CLOCK GENERATOR

The Clock Generator takes as its inputs the output of the High Frequency Oscillator, HFO, and also the speed control bit, SPD. It outputs several clock frequencies that are used by different portions of the HOPPER chip. The following is a block diagram of the clock generator circuitry.

Figure 15.1. Clock Generator Block Diagram



Signal	Where Used	Hopper Freq	PC Freq	Error
CCLK	CPU, DISPlay, Bus InterFace	5.369318MHz / 10.738636MHz	4.7727MHz +125%	+12.5% /
TCLK	TIMER0,2	1.193182MHz	1.193182MHz	0%
UCLK	UART	1.845703MHz	1.8432MHz	+0.14%
HFO	DISPlay, TOUCH, TONE	5.369318MHz	n/a	
LFO	RTC	32.768KHz	n/a	

The HOPPER chip will accept either a 5.369318MHz (5.37MHz) or a 10.738636MHz (10.74MHz) crystal. The SPD bit must be initialized to a "zero" for a 5.37MHz crystal or a "one" for a 10.74MHz crystal. This initialization must occur before using the display, timers, UART, touch, or tone circuits.

NOTE: Operation with a 10.74MHz crystal is not supported at this time.

CHAPTER 16

TOUCH PANEL CONTROLLER

The HOPPER IC includes an analog touch panel interface that consists of an A/D converter and touch panel control registers. In order to provide a simple and flexible circuit, the burden of controlling the touch panel and the A/D converter is placed on software.

The touch panel control circuit consists of 2 resistive sheets of material. The x-axis sheet is connected on the left and right ends and the y-axis sheet is connected on the top and bottom. These 4 connections are wired to the TPL[0:1] and TPH[0:1] pads of the HOPPER IC. The x-axis position of a finger or stylus that is touching the panel is determined by driving one side of the x-axis sheet to VDD and the other to ground. Both sides of the y-axis sheet are precharged to ground and then released. The voltage that appears on the y-axis sheet is measured using the A/D converter and corresponds to the x-axis position of the touch. The y-axis position is determined in a similar manner by driving the y-axis and using the x-axis to sense voltage.

16.1 The A/D Converter Interface (ADCONT)

Software is given control of the A/D converter through the use of the control bits in the ADCONT register at I/O address E307h. The contents of the register are as follows:

Signal	Bits	Type	Description
CHS0,1,2	0,1,2	R/W	Analog Input Channel Select
PWUP	3 R/W		Power up the A/D
START	4 R/W		Start an A/D conversion
STA	5 R		A/D converter Status

Before using the A/D, it should be turned on by setting the PWUP bit to 1. Using the CHS0-2 bits, software can choose between the 8 possible analog input channels to do the A/D conversion on. To start an A/D conversion, the START bit must be set. During the first twenty clock cycles after the START bit is set, the A/D does a sample and hold of the chosen input channel. The START bit should be cleared previously when the channel is selected. The STA bit will be set when the conversion is finished. The conversion including the sample and hold takes a total of 130

clock cycles. The digital output value can then be read at I/O address E306h. When the A/D is not being used, it should be powered down by resetting the PWUP bit to 0. When the system is reset, the A/D will also be reset and all of the bits in the ADCONT register set to 0.

The analog input channels are as follows:

Signal	Channel	Description
BAT[0]	0	System Battery Voltage
BAT[1]	1	Backup Battery Voltage
BAT[2]	2	Card Battery Voltage
BAT[3]	3	Card 2 Battery Voltage
TPL[0]	4	Touch pad X-axis low side
TPH[0]	5	Touch pad X-axis high side
TPL[1]	6	Touch pad Y-axis low side
TPH[1]	7	Touch pad Y-axis high side

16.2 Touch Panel Control (TPCONT)

The touch panel control circuit has 6 possible modes. They are: 1) off; 2) standby; 3) precharge x-axis; 4) precharge y-axis; 5) measure x-axis; and 6) measure y-axis. These modes are completely under software control and are set by the TPCONT register at I/O address E308h. The individual bits of this register are defined as follows:

Bit	Name	Description
0	STDBY	Touch panel standby mode. When this bit is set the entire touch panel is treated as a single switch. In this mode the TUCH status bit will go high if the panel is touched. This will also cause an interrupt if the ETINT bit is set. This is a zero-power state as long as the panel is not being touched. The panel should be precharged by setting both the STDBY bit and the TPPC bit for several microseconds before entering standby mode. The SELX bit has no effect on this mode.

1	TPPC	Touch panel precharge mode. If the STDBY bit is not set, the axis chosen to be measured by the SELX bit is connected from VDD to Ground. The panel must be precharged for several microseconds before making an A/D measurement. In STDBY mode, this bit pulls one axis high and the other low for precharge.
2	SELX	Select X bit. The x-axis is selected for measurement if this bit is set. The y-axis is selected if it is clear.
3	PULL0	This bit allows the axis opposite that chosen for measurement to be pulled to ground. This could be used to decide if a reading is valid, ie. a (0,0) reading is a no touch reading. This bit only has effect in an active precharge mode.
4	ETINT	Enable Touch Pad Interrupt. If this bit is set and the touch pad is in Standby mode, an interrupt will occur if the pad is touched.
5	TUCH	Read Only. Indicates that the pad has been touched in Standby mode.

When the pad is not in use it should be shut down by writing XXXX00 to the TPCONT register. Before taking a touch pad reading, the SELX bit must be set to the appropriate value for the axis that is desired. The Touch pad should then be precharged for a predetermined amount of time by setting the TPPC bit to one. The ADCONT register must also be set up to read the appropriate analog input channel by setting the CHS bits. After the precharge time has passed, the START bit in the ADCONT register must be set to start the A/D conversion. Twenty clock cycles after the START bit is set and the A/D is finished with its sample and hold, the TPPC is automatically cleared shutting off the Touch pad. This is accomplished with the use of a 5-bit counter in the TPCONT register and allows the A/D time for the sample and hold portion of the conversion process.

When not actively reading the Touch pad, software can wait for the pad to be touched by placing the pad in Standby mode. Before entering Standby mode, a Standby precharge must first occur. To do this, software must set both the STDBY and TPPC bits in the TPCONT register. After precharging for a given amount of time, the TPPC bit is cleared and the pad is now in Standby mode. When

the pad is touched, the TUCH bit will be set to a one, and if the ETINT bit is set, a Touch pad interrupt will occur. Software can then take a Touch pad reading.

Warning! - Because the Touchpad uses a considerable amount of power, it should be shut off when not in use.

CHAPTER 17 TONE GENERATOR

17.1 Hardware Description

The tone block on the HOPPER IC consists of digital control circuits and registers to control a D/A converter and a differential output amplifier. This block allows the HOPPER chip to output custom sounds and speech using an external piezo speaker. To output a sound waveform, software must write a series of digital values representing the waveform to the D/A. The D/A converts the digital values to analog voltages at a conversion rate of 16kHz. This allows the waveforms to contain frequencies up to 8kHz which is satisfactory for speech. After the D/A has used a value it signals software that it is ready for another value by setting the DAMT bit. The output voltage is converted to a differential voltage by the output amplifier and output to the external piezo. The D/A converter and output amplifier can be powered down when not in use by resetting the PWRDN bit to save on current consumption. The hardware will detect a digital beep to power up the amplifier and will power it down after the beep finishes.

There is also a digitally produced sound signal that is multiplexed with the analog speaker outputs. This will be used as a backup plan in case the analog portion of hopper is unusable. This signal will be the output of counter 2 ORed with port B1. The signal to be output to the piezo is selected by default. If the D/A converter is powered up by setting the PWRDN bit, the analog waveform is chosen. The volume of the digital signal can be controlled using the DVAL bits in the TCCONT register. These two bits allow 4 levels of volume control for the digital output signals, 3 being the loudest. The analog waveform can be controlled by scaling the analog values that are written to the D/A buffer.

The DAMT and PWRDN are located in the TCCONT register at I/O address E309h. The DVAL bits to control the digital volume are also located in this register as well as the CVAL bits for the CVGEN block. The register contents are as follows:

Name	Bit	Description
------	-----	-------------

DAMT	0	Set to 1 when D/A Buffer is empty, read only
PWRDN	1	Powers down the D/A and analog buffer when reset to 0
DVAL[0:1]	2,3	Volume control for the digital tone signal
CVAL[0:3]	4-7	Voltage selector value for CVGEN

17.2 Software Control

During tone output generation, software must write one sample value to the D/A buffer register every 61 μ s. This buffer is located at I/O address E305h. This timing restriction may require that all values be stored in RAM beforehand and just rewritten to the D/A buffer during tone output. The DAMT bit will be set to 1 when the hardware is ready for another sample value and otherwise be set to 0. Software must poll the DAMT bit before it writes to the D/A buffer.

Before outputting an analog waveform software should write a 7Fh to the D/A buffer, a differential 0, and then set the PWRDN bit to turn the D/A on. Digital sample values for the analog signal can then be written to the D/A. A 1 second sound requires 16384 bytes to be written to the D/A buffer sequentially. At the end of each tone, the PWRDN bit must be reset to turn off the D/A converter.

CHAPTER 18 POWER MANAGEMENT

The power management circuitry controls the transitions between the following system modes:

Mode	SUPPLY	EXTERNAL CPU	DISPLAY	TIMER	HFO	LFO
Static	-	OFF	OFF	OFF	OFF	OFF
Deep Sleep	Low Power	OFF	OFF	ON	OFF	ON
Light Sleep	Hi Power	OFF	ON	ON	ON	ON
Operating	Hi Power	ON	ON	ON	ON	ON
Backup	Backup	OFF	OFF	OFF	OFF	ON

18.1 Static Test Condition

The static test condition is entered only during test. This mode is used to measure static leakage current. In this mode the part has been reset and no clocks are being driven. The low frequency oscillator (LFO) is held inactive externally.

18.2 Deep Sleep

Deep sleep is entered from operating mode when software sets the SHT bit with the DON bit cleared. The DON bit is described in the chapter entitled "HOPPER DISPLAY CONTROLLER". In deep sleep the SOFF pad is driven high causing the external supply to operate in low power mode. The system will exit deep sleep and begin operating when an enabled interrupt is received from either: 1) the ON key; 2) Keyboard input; 3) RTC underflow; 4) the external interrupt pad, XINT, being pulled high; or 5) a rising edge on the RX pad (UART interrupt input).

18.3 Light Sleep

Light sleep is entered from operating mode when software sets the SHT bit with the DON bit set. The external supply continues to operate in high power mode in light sleep. Any enabled interrupt source will exit light sleep to operating.

18.4 Operating

Operating mode can be entered from either of the other two modes. If it is being entered from deep sleep then the HOPPER chip will hold off the CPU in reset for 35ms after requesting high power (SOFF=0) from the external supply. The 35ms delay is generated in the RTC pre-divider. Following this delay the CPU will begin operation at the reset vector (ffff0h) with interrupts disabled by the CPU reset.

When software is finished processing, it may exit operating mode and enter either light sleep or deep sleep. Light sleep is entered by setting the SHT bit. Deep sleep is entered by first clearing the DON bit, then setting the SHT bit. The CPU will prefetch several instructions following the instruction that sets the SHT bit. This requires that several NOP instructions follow setting the SHT bit. When operating mode is again entered, hardware will clear the SHT bit. It is up to software to control the state of the DON bit.

18.5 Backup

When the external power supply senses that VDD is too low it will drive the NVDDOK line high. This will cause an NMI if the system is in light sleep or operating modes. Software will use this NMI to quickly shutdown the system. When the system is in deep sleep or in the process of powering up with NVDDOK high it will immediately enter backup mode. In backup mode the BUP output is driven high switching the external supply to the backup battery. The system will remain in backup mode until the ON key is pressed.

If NVDDOK does not go low within 50ms of entering backup mode, or if NVDDOK goes low at some time following the 50ms delay, the BUP line will be driven low switching the external supply back to the main batteries. The system is still in backup mode however and will stay there until NVDDOK goes low.

The hopper chip will ignore the state of the NVDDOK signal if the ACIN signal is high indicating an AC adapter is plugged in. Both NVDDOK and ACIN are deglitched internally using the 32KHz oscillator.

18.6 System Resets

The HOPPER chip will be reset when:

1. Power is first applied (Power-On-Reset (POR) circuit), or a logic low level is driven on the NRES pad. This reset will be held until NVDDOK=0.

2. The RST bit is set twice in succession.
3. The keyboard circuitry senses that the ON pad, KBI[6], and KBI[7] are all high for a 7.8ms debounce time (ON/SHIFT/CNTRL keys depressed).

A reset will cause the system to:

1. Execute the power-up sequence if asleep,
2. Reset the CPU, and
3. Enter operating mode.

18.7 System Control Register

The 2-byte System Control Register controls the system mode and also contains the interrupt enable bits that allow individual interrupt sources to be enabled or disabled. If an interrupt is disabled it will not cause a CPU interrupt and will not cause a wakeup. If enabled, all interrupt sources will cause a wakeup from light sleep or deep sleep. In order to cause a wakeup, the wakeup source must be valid for 50us. After a wakeup, the system will ignore shutdown attempts for 15us.

The first byte of the System Control Register is located at E301h and is defined as follows:

Bit	Reset Value	R/W Mode	Description
[0]	-	W	RST - Reset. Writing a one to this bit twice in succession will cause the system to reset.
[1]	0	W	SHT - CPU shutdown. Setting this bit will stop clocks to the CPU. The system will enter light sleep mode if the DON bit is set and will enter deep sleep mode if the DON bit is clear. This bit is set only, it is automatically cleared by a wakeup.
[2]	0	R/W	ION - I/O On. This bit controls the power to the IR receiver chip and possibly to the RS232 transmitter.
[3]	X	-	unused
[4]	0	R/W	UTE - UART interrupt/wakeup enable.

[5]	0	R/W	IRE - IR circuit interrupt/wakeup enable.
[6]	0	R/W	EXE - External (XINT pad) interrupt/wakeup enable.
[7]	0	R/W	RTCE - Real Time Clock interrupt/wakeup enable.

The second byte of the System Control Register is located at E302h and is defined as follows:

Bit	Reset Value	R/W Mode	Description
[0]	0	R/W	T0E - Timer 0 interrupt/wakeup enable.
[1]	0	R/W	T1E - Timer 1 interrupt/wakeup enable.
[2]	0	R/W	LPE - Low power interrupt/wakeup enable.
[3]	0	R/W	MPE - Module pulled interrupt enable.
[4]	0	R/W	DCE - Display cursor update request interrupt enable.
[5]	0	R/W	RXE - RX pad low voltage interrupt/wakeup enable. This bit should be set when the supply voltage is reduced in deep sleep and a wakeup is desired when activity is detected on the serial port. It is necessary to select an input device with the proper threshold on the RX pad.
[6]	0	R/W	KBE - Keyboard interrupt/wakeup enable.
[7]	0	R/W	TPE - Touch panel interrupt/wakeup enable.

CHAPTER 19

CONTRAST CONTROL VOLTAGE GENERATOR

The Contrast Control Voltage (CCV) Generator is a simple 4-bit D/A converter. An analog conversion of the 4-bit CCV register value is output on the CCV pin anytime that the display is on (DON=1). This analog voltage is used by the external LCD voltage generation circuitry to control the drive levels on the display panel. Variations in these drive levels affect the apparent darkness or "contrast" of the LCD to compensate for variations in the LCD material and for differing viewing angles. Thus the CCV allows the user to adjust the contrast of the display under software control.

The CCV Generator circuit consists of a simple resistive ladder. The output pin is high impedance and should not drive a DC load.

CHAPTER 20 SPECIAL HARDWARE CONSIDERATIONS AND HOPPER REV C
--

This chapter is intended to detail several hardware features that need special consideration when using the HOPPER IC. Most of the shortcomings discussed here have been addressed in a revised version of the HOPPER IC. This revised version is referred to as Rev C. It is scheduled to be phased into production in the fall of 1991.

20.1 Pseudo-Static RAMs and the CPU Halt Instruction

When the CPU is being clocked (run mode), HOPPER issues an automatic refresh cycle during the start of each CPU cycle. This refresh cycle is necessary for pseudo-static RAMs to maintain data. Unfortunately this means that if the CPU stops issuing new cycles, as it does when a halt instruction is executed, no refresh cycles will occur. The RAM specification states that 2048 refresh cycles must occur each 32ms. In normal operation, 2048 refresh cycles will occur about every 7ms (this figure was calculated assuming that the average instruction is 16 clock cycles long). Therefore, if the CPU is halted for more than about 25ms, the RAM refresh specification will be violated.

In actual practice, our experience has been that most pseudo-static RAM parts will maintain data for several seconds with no refresh cycles. However, this cannot be guaranteed. It is preferable that no halt instructions be used. If they are used, the halt must be ended with an interrupt well under the 25ms time limit.

Rev C of the HOPPER IC includes a "WATCHDOG" circuit. When enabled by setting bit 5 of IO location E300h, this circuit will attempt a system shutdown if no refresh cycle occurs in 3.5 to 4.5 ms. If this shutdown is successful, it will put the PSRAMs into self-refresh. The only known way for this to occur is if refreshes were suspended by the execution of a HALT instruction. This shutdown will be transparent to software since HOPPER will wake-up on the first enabled interrupt. This interrupt will also terminate the HALT.

After attempting to set the shutdown bit, the WATCHDOG circuit continues to watch for PSRAM refresh. If a refresh does not occur in another 2 ms, a hardware reset will be initiated. Holding the RDY line low for several milliseconds is the only known way of

causing this reset. Holding the RDY line low will not allow a successful shutdown since CLK is stopped.

20.2 Pseudo-Static RAMs and Hardware Reset

HOPPER is defined such that if the KBI[6:7] and ON pads are all driven high a hardware reset will occur. This corresponds to hitting the [SHIFT] [CTRL] [ON] keys on the 95LX. A hardware reset will also occur if the NRST pad is pulled low.

The hardware resets are asynchronous with HOPPER chip-enable/output-enable timing and therefore can cause glitches on these signals. This may corrupt user memory since pseudo-static RAMs are sensitive to glitches on either chip-enable or output-enable.

In actual use, the user RAM appears to be fairly insensitive to corruption caused by hardware resets. The RAM based operating system seems to be much more sensitive, probably because it is being chip enabled more often. But since the output-enable signal is common to all RAMs, a part can be corrupted even if it is not being accessed when a reset occurs.

The ability to cause a hardware reset from the keyboard is a very useful and powerful feature. But users should be warned that it is only to be used as a last resort, and it may corrupt memory. Software should never be written that will intentionally hang the system and require the user to do a hardware reset.

Rev C of the HOPPER IC includes circuitry that will attempt to synchronize the start of most hardware resets. This circuitry provides a window of 60us during which a synchronous reset will be initiated if memory timing allows. If memory timing is such that a synchronous reset did not occur during this window, an asynchronous reset will be initiated. Asynchronous resets should only occur in deep sleep. This change will prevent corruption of PSRAMs due to all hardware resets except for those initiated by pulling the NRES pad low.

20.3 Display Cursor

The display cursor will disappear in an area of dots that are all on. This is because the cursor is defined as turning dots on. HOPPER Rev C will define the cursor as inverting the normal dot data. This will allow it to be always visible.

HOPPER Rev C also has a small change to provide MDA compatibility for some illegal attribute bytes.

20.4 TX Output

The TX output is normally driven high even during deep sleep. (TX is the invert of TXD). This causes a high current condition in the 95LX since the power to the TX buffer is removed in deep sleep. In order to prevent this, software is required to send a break after turning IO off and before deep sleep shutdown. This break is never seen outside of the product since turning IO off tristates the TX buffer. On wakeup, the break should be removed before turning IO on. On HOPPER Rev C this problem is solved in hardware by forcing TX low in deep sleep.

Another problem with the TX output is that it is forced low when IR UART mode is selected. This will be interpreted as a break by any device that may be connected to the serial port. Some printers will print a garbage character in response to a break. There is no software workaround for this problem. HOPPER Rev C will hold TX high when IR UART mode is selected.

20.5 Timer Wakeups

A standard PC BIOS will set up TIMER0 in Mode 3. This causes the timer output to be a square wave. If this timer output is enabled to cause wakeups, the HOPPER system will be unable to shutdown whenever the timer output is high, which is 50% of the time. This causes a large increase in current while the system is idle. The same problem exists with TIMER1 wakeups.

The 95LX BIOS works around this problem by setting up TIMER0 in Mode 0. When the 55ms count expires, the interrupt routine calculates a new count value and reloads TIMER0. This works, but causes problems for PC applications that expect TIMER0 to be in Mode 3.

HOPPER Rev C will solve this problem by making the TIMER0/TIMER1 wakeups edge sensitive. Edge sensitive in this case is defined as meaning that timer wakeups are disabled if they are active when a wakeup occurs. They are re-enabled when the request goes inactive.

20.6 Interrupt Source Register (ISR)

Bits in the ISR (E303h) are set only after the 45ms power-up delay. This makes it possible for a wakeup source such as the serial port to cause a wakeup and then go inactive before setting the appropriate bit in the ISR. It is therefore sometimes very difficult for software to determine the cause of the wakeup. HOPPER Rev C solves this problem by allowing bits in the ISR to be set immediately.

20.7 The ON Key

It is impossible to shutdown while the ON key is being held down. This is considered a problem only for low-power conditions. There is no software workaround for this problem. HOPPER Rev C solves this problem by making ON key wakeups edge sensitive.

Another problem with the ON key is that it may cause a wakeup, then be released before software can read its status due to the 45ms power-up delay. The Rev C fix to the ISR discussed above does not solve this problem since keyboard wakeups must be disabled in deep sleep. When keyboard wakeups are disabled, the ON key is the only key that will cause wakeup. This problem is solved on Rev C by latching the ONKEY bit (E30Fh bit 0). This bit is cleared only if the ON key is up when it is read.

20.8 RDY Timing

A timing problem in the HOPPER IC requires that the RDY pin be constrained to go inactive (low) after the fall of one of the chip enables. HOPPER Rev C fixes this timing problem and will allow RDY to go inactive at any time.

20.9 Speaker Power-Down

The HOPPER IC includes circuitry that will automatically power-up the beeper amplifiers whenever activity is sensed on the PC compatible speaker output. The amplifiers are powered-down after up to 125ms of inactivity. If a shutdown occurs while the amplifiers are powered-up, the system will be left in a very high power mode. This problem requires software to delay 125ms after the ON key is pressed before shutting down.

HOPPER Rev C forces the amplifiers off in deep sleep.

20.10 Keyboard Precharge and Reset

The keyboard reset feature ([ON]-[SHIFT]-[CTRL]) is disabled while the keyboard is in precharge. Keyboard precharge is initiated anytime the keyboard output register is written (E30Eh-E30Fh). It is terminated by a write to E30Dh.

HOPPER Rev C does not address this problem.

20.11 UP8250 Lockup

Writes to the UP8250 Line Control Register (LCR) during receive and transmit operations can result in lockup and/or incorrect data reception or transmission. To prevent this, software must follow these steps:

1. Clear IR UART communication mode (E30Ah bit 2 \leftarrow 0).
2. If the baud rate divisor is zero, change it to a non-zero value.
3. Wait for transmitter empty (03FDh bit 6 = 1).
4. Enable RX pad interrupts (E302h bit 5 \leftarrow 1). Then wait for at least one full frame time with no RX interrupts. This frame time will vary according to the number of data bits, parity, and stop bits. Worst case is 12 bits at 300 baud = 40ms. If an RX interrupt is received during this time, the wait time should start over. (The RX interrupt is on IRQ2 and will set E303h bit 5).
5. The Line Control register (03FBh) must now be written within 5 bit times. Worst case is 115.2K baud = 43us.

HOPPER Rev C solves this problem by buffering new values that are written to the LCR. These new values become active only on word boundaries.

20.12 UP8250 Parity Enable

Changing UP8250 parity from disabled to enabled will cause the receiver to corrupt the next character as well as the error flags associated it. This problem is fixed on Rev C HOPPER.

20.13 UP8250 Interrupts and Interrupt ID Register

The Interrupt ID Register (IIR) is synchronized with the 16X UART clock. This causes delays in the setting and clearing of the IIR bits. This in turn causes numerous problems for software that is interfacing with the IIR register. There is also a bug that will allow a false indication of receiver line status interrupt (110).

It is recommended that software avoid using bits 1 and 2 of the IIR to determine the interrupt source. The Line Status Register should be used instead to determine the source of interrupts. Also, in order to avoid missing interrupts, the interrupt handling routine should not return until bit 0 of the IIR is set indicating no pending interrupts.

The IRR and interrupt logic has been redesigned for Rev C HOPPER. The redesigned circuit is compatible with a standard 8250. During this redesign, it was decided not to attempt to issue new edges on the interrupt signal each time there is a new interrupt source. This is compatible with both the Intel 8250A and the National 16450. It requires that the interrupt handling routine service all interrupts before returning.

20.14 UP8250 Receiver Error Bits

The receiver error bits (OE, FE, BI, and PE) are reset by the next received word. This is not compatible with the 8250A. On Rev C HOPPER these bits are reset only when the Line Status Register (LCR) is read.

Also, the parity error bit (PE) is updated when the parity bit is received. On Rev C HOPPER, this bit is updated on the stop bit along with the other LCR bits.

20.15 UP8250 Modem Status Register

The Modem Status Register always reads 00h. It was determined that bits 4 (clear-to-send), 5 (data-set-ready), and 7 (receive line signal detect) should be high. Jagaur's DOS service routines pretend that these bits are always high. Software that reads the hardware directly, however, may not work. On Rev C HOPPER these bits will always be set.

20.16 UP8250 Break Reset

When a BREAK condition occurs, the UART receiver is shutdown until the Line Status Register is read clearing the BI bit. On a standard 8250, the receiver will receive a new character regardless of whether or not the BI bit has been serviced. Rev C HOPPER will fix this compatibility problem.

20.17 UP8250 Line Status Register

Continuous polling of the Line Status Register may cause receiver errors to be missed. Rev C HOPPER eliminates this possibility.

20.18 UP8250 Receiver Buffer Register

Reading the Receiver Buffer Register (RBR) after the full word has been received, but before the stop bit has been received will result in loss of the previous data with no Overrun Error

reported. HOPPER Rev C will fix this problem by not updating the RBR until the stop bit is received.

CHAPTER 21 TESTING

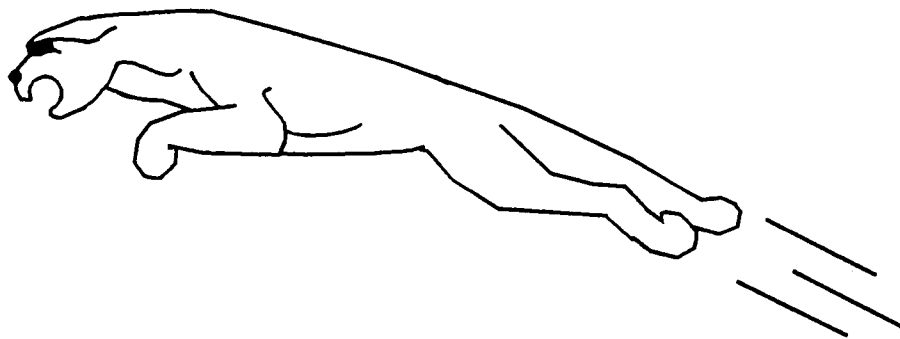
A test pad has been included to force the part into test mode. The following sections are implemented as macro-cells and are tested using the appropriate off-the-shelf test program:

- 8288 compatible BUS CONTROLLER
- 8254 compatible PIT
- 8259 compatible PIC

The remaining blocks are tested using custom test programs.

A output disable (OD) mode is implemented for testing of input leakage on pads with resistive pullups/pulldowns. When the OD bit has been set, these pins will resistive elements are disabled.

HP95LX
WIRED SERIAL AND INFRARED I/O
EXTERNAL REFERENCE SPECIFICATION



HEWLETT-PACKARD

April 11, 1991

CONTENTS

INTRODUCTION.....	1
RS232 COMPATIBLE SERIAL PORT.....	2
Protocol.....	2
Hardware	2
Software	3
BIOS control	3
Direct Register Control.....	6
IR I/O SERIAL PORT	10
Protocol.....	10
Hardware	10
Software	11
OTHER IR COMMUNICATION.....	12
REDEYE Format	13
Software Controlled Mode.....	14
Modulated Output Mode.....	14
Modulated IR UART Output Mode	15

INTRODUCTION

The *HP95LX* contains several I/O capabilities. These include an RS232 compatible wired serial port and a two way wireless infrared port. This document contains a brief description of the hardware, the supporting software, and the communications formats used. This document is not a complete description of all of the *HP95LX* features. It is not guaranteed to be 100% accurate as hardware and software changes may be made after its release. It will, although, address most of the issues involved in using the *HP95LX* I/O capabilities.

RS232 COMPATIBLE SERIAL PORT

Protocol

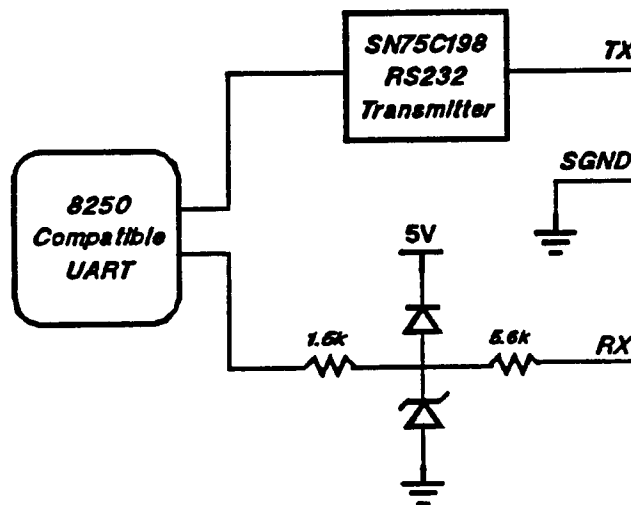
The serial protocol used in *HP95LX* is the standard asynchronous serial protocol used in the PC. It consists of three signal connections; RXD, Receive Data, TXD, Transmit Data, and SGND, Signal Ground. Each data word is preceded by a start bit, which is a spacing condition for one serial bit time. Each word is followed by at least 1 stop bit, which is a marking condition for one bit time. When the transmitter is idle, the transmit line is held at a marking condition.

The wired serial port voltage levels are -4.5 Volts for a logic level '1' or marking condition and 4.5 volts for a logic level '0' or spacing condition. These are consistent with the RS232 standard for wired communication. When powered down, the serial transmit line will be held to ground. *HP95LX* interprets this as a marking condition as will most RS232 receivers.

Hardware

The *HP95LX* serial port hardware includes an 8250 compatible UART, an SN75C198 transmitter chip, and a discrete level shifting receiver portion. A block diagram of the serial hardware is shown below. The SN75C198 is an RS232 transmitter that converts a CMOS logic level to the appropriate positive and negative voltages used for the serial port. The receiver section is used to convert the positive and negative RS232 voltages to CMOS compatible voltage levels. The 8250 compatible UART controls all serial communication and interfaces to the rest of the *HP95LX* system. Using the UART, the baud rate is set, the word length of 5, 6, 7, or 8 bits is chosen, and parity error detection can be used. Although the hardware will support baud rates up to 115.2k baud, the wired link is not guaranteed to work for all applications above 20k baud. For optimum performance, a short cable, less than 3 meters, should be used.

Serial Hardware



Software

Software control of the serial port can be accomplished by writing to the appropriate hardware registers or through BIOS routines.

BIOS control

There are several software interrupt functions provided in the BIOS for serial port control. It is recommended that applications use the BIOS routines to control the serial port. Here is a summary of these functions.

Int 15h

Set Serial Channel

This function sets the HP95LX serial channel to either Wired or IR. If IR mode is selected, the UART baud rate is changed to 2400 baud.

Input: AH = 49h
 AL = 0h : Wired
 = 1h : IR
 > 1h : No Operation

Output: None

Registers modified: None

Set RS232 Power

This function turns power to the RS232 transmitter and IR receiver on or off.

Input: AH = 4Ah
 AL = 0h : Power off
 = 1h : Power on
 > 1h : No Operation

Output: None

Registers modified: None

INT 14h

Initialize Serial Port Parameters

This service sets the baud rate, parity, number of stop bits, and character frame size for the specified serial port.

Input: AH = 00h
 DX = Serial port number (Always 0 in HP95LX)
 AL = Parameters shown below

Bit	Description
7-5	Baud rate 000 = 110 baud 001 = 150 baud 010 = 300 baud 011 = 600 baud 100 = 1200 baud 101 = 2400 baud 110 = 4800 baud 111 = 9600 baud
4-3	Parity 00 = None 01 = Odd 10 = None 11 = Even
2	Stop bits 0 = One stop bit 1 = Two stop bits
1-0	Frame size 10 = 7 bit word 11 = 8 bit word

Output: AH = Serial port status

Bit	Description
7	Timeout
6	Transmit shift register empty
5	Transmit buffer register empty
4	Break detect
3	Framing error
2	Parity error
1	Overrun error
0	Data ready

AL = Modem Status (Modem Control not implemented
in HP95LX)
This will always be returned as B0h.

Bit	Description
7	Receive Line Signal Detect
6	Ring Indicator
5	Data Set Ready
4	Clear to Send
3	Delta Receive Line Signal detect
2	Trailing edge Ring Detect
1	Delta Data Set Ready
0	Delta Clear Send

Registers modified: AX

Transmit One Character

This function transmits one character through the serial port. It waits until the UART transmit buffer is empty, then transmits the given character.

Input: AH = 01h
AL = Character to transmit
DX = Serial port number (Always 0 in HP95LX)

Output: AH = Serial port status (same a previous function)

Registers modified: AH

Receive One Character

This functions returns the character received by the serial port. It waits until the serial port reports that a character has been received, then reads character and the serial port status.

Input AH = 02h
DX = Serial port number (Always 0 in HP95LX)

Output: AH = Serial port status (only bits 7, 4, 3, 2, 1 as before)
AL = Character received

Registers modified: AX

Get Serial Port Status

This function reports the status of the serial port.

Input: AH = 03h
 DX = Serial port number (Always 0 in HP95LX)

Output: AH = Serial port status (same as Initialize function)

Registers modified: AX

Direct Register Control

Serial communication in *HP95LX* can also be accomplished by writing directly to the hardware registers that control the communication functions. The first important register is located at I/O address E301h. Bit 2 of this System Control Register is used to turn the RS232 transmitter and the IR receiver on and off. By setting bit 2, power is turned on to these devices. The other bits in this register control various other system functions so bit 2 should be ORed with the contents of the register. When the *HP95LX* is in DOS, the FILER, or COMM application, bit 2 of E301h will be set. To chose between IR mode and wired mode, the register at I/O address E30Ah is used. To select wired serial communication, a 0h must be written to this register.

The rest of the serial registers are used to control the 8250 compatible UART. There functions are equivalent to those for the UART in the PC. The registers are as follows:

I/O Address 3F8h

DLAB = 0

W UART Transmitter Holding Register
 Contains the character to be sent.

R UART Receiver Buffer Register
 Contains received character

DLAB = 1

R/W UART Divisor Latch- Low Byte
 Contains the low order byte of the baud rate divisor

I/O Address 3F9h

DLAB = 0

R/W UART Interrupt Enable Register
Allows enable and disable of UART interrupt sources

bit 7-4	=0	reserved
bit 3	=1	not used
bit 2	=1	Enable receiver line status interrupt
	=0	Disable
bit 1	=1	Enable transmitter holding register empty interrupt
	=0	Disable
bit 0	=1	Enable received data available interrupt
	=0	Disable

DLAB = 1

R/W UART Divisor Latch- High Byte
Contains high-order byte of UART baud rate divisor.

I/O Address 3FAh

R UART Interrupt ID Register
Contains information about interrupts pending. Only the highest priority interrupt is indicated.

bits 7-3	=0	Reserved
bits 2-1	=11	Receiver line status interrupt: Highest priority
	=10	Received data available: Second priority
	=01	Transmitter holding register empty: Third priority
	=00	Invalid
bit 0	=1	No interrupt pending
	=0	Interrupt pending

I/O Address 3FBh

R/W UART Line Control Register

bit 7	= 1	DLAB: select divisor latch access
	= 0	select receiver buffer, transmit holding reg, and interrupt enable reg access
bit 6	= 1	Set break enabled
bit 5	= 1	Stick parity enabled
bit 4	= 1	Even parity
	= 0	Odd parity
bit 3	= 1	Parity enabled
bit 2	= 1	2 stop bits, 1.5 if 5 bit word
	= 0	1 stop bit
bits 1-0	= 11	8 bit word length
	= 10	7 bit word length
	= 01	6 bit word length
	= 00	5 bit word length

I/O Address 3FCh

R/W UART Modem Control Register Modem control is not implemented

bit 7-4	= 0	reserved
bit 3	= 1	Enable UART interrupt
bits 2-0	= 0	reserved

I/O Address 3FDh

R UART Line Status Register

bit 7	= 0	reserved
bit 6	= 1	Transmitter empty
bit 5	= 1	Transmitter holding register empty
bit 4	= 1	Break interrupt
bit 3	= 1	Framing error
bit 2	= 1	Parity error
bit 1	= 1	Overrun error
bit 0	= 1	Data ready

I/O Address 3FEh

R UART Modem Status Register Modem control is not implemented

bits 7-0	= B0h	reserved REV C Hopper
	= 00h	reserved REV B Hopper-(Early production <i>HP95LX</i>)

I/O Address 3FFh

R/W UART Scratch Pad Register

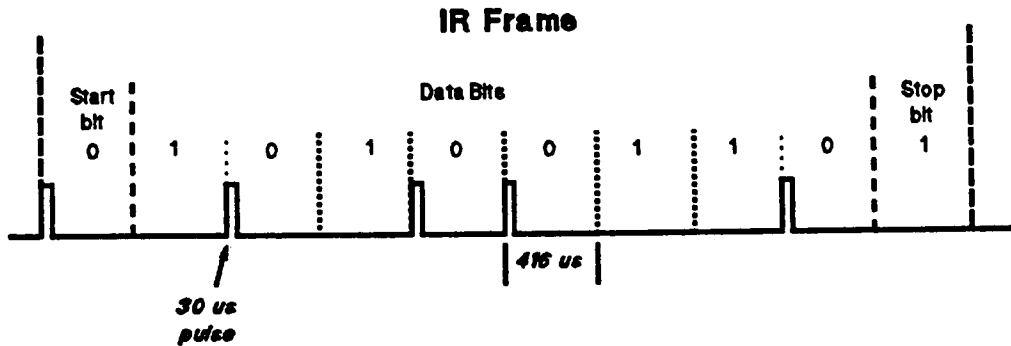
The baud rate divisor should be chosen as follows:

Baud rate	Divisor
300	180h
600	C0h
1200	60h
2400	30h
4800	18h
9600	0Ch
19200	06h

IR I/O SERIAL PORT

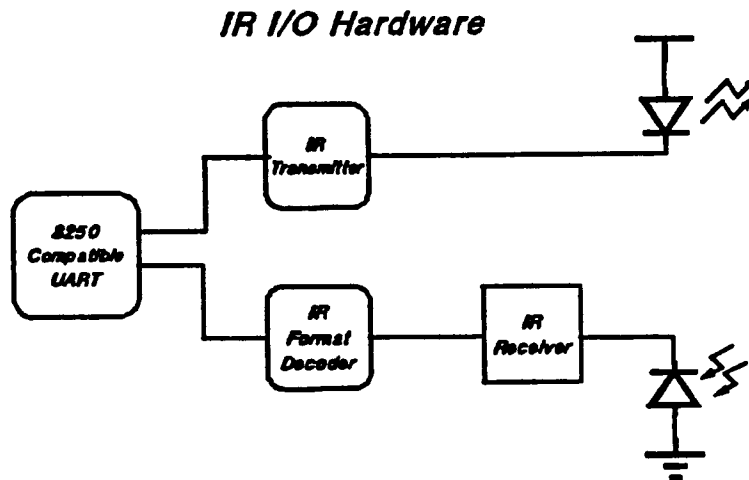
Protocol

The *HP95LX* IR I/O port is a half-duplex serial port using infrared light as a communication channel. The port is capable of communicating at 2400 baud using any of the available word modes used for the wired serial port. The mark state, logic level 1, is indicated by no transmission. The space state, logic level 0, is transmitted by a single 30 msecond IR pulse per bit. A bit time at 2400 baud is 416 mseconds. An example IR I/O frame is shown below.



Hardware

The *HP95LX* IR I/O hardware block diagram is shown below.



The UART used is the same UART used for wired serial communication. When IR I/O mode is chosen, the UART is disconnected from the wired serial port and connected to the IR I/O port. The wired TX line will be held in the marking condition, spacing condition in early production *HP95LXs*, to indicate that the serial port is idle. The wired serial RX line is just ignored. The UART is used to

transmit and receive the serial data streams. The output of the UART is converted to the IR output protocol and then converted to light by turning an infrared LED on and off. The LED transmits at a wavelength of 940nm. The infrared receiver uses a photo diode to detect the incoming IR light. The light pulse is transformed into a CMOS level digital pulse by the IR receiver. The IR format decoder then generates the appropriate serial bit stream which is sent to the UART. Due to reflections, the IR serial port receives everything that it transmits. This must be taken into account in software.

Software

Software control of the *HP95LX* IR I/O port is very similar to the wired serial port. All of the BIOS routines support IR as well as serial. The baud rate must be set to 2400 as higher baud rates are not functional. To turn on power to the IR receiver either Int 15h can be used as shown previously, or bit 2 of the register at I/O address E301h can be set directly. Again, the other bits of this register should be left alone. To select the IR I/O port, software Int 15h can be used as shown previously. Again, it is recommended that the BIOS routines be used to set up the port. The IR port can also be chosen by writing a 04h directly to the register at I/O address E30Ah.

After the port is chosen and powered up, the UART can be used to transmit and receive as it was used in the RS232 wired serial port. The only issue to keep in mind is that the IR I/O port will receive everything it sends.

OTHER IR COMMUNICATION

The IR communication hardware of *HP95LX* also supports three other IR communication modes. These modes are REDEYE, software controlled communication, and modulated output mode. These formats are all software controlled using the IRFMAT register and the IRCNT register. The IRFMAT register is located at I/O address E30Ah and contains control bits to choose the IR communication format. The IRCNT register is located at I/O address E30Bh and is used to transmit a bit or wave form to the LED or monitor the IR receiver. The registers contents are as follows:

IRFMAT Register
I/O Address E30Ah

Bit	Name	Description
0	RED	This bit activates REDEYE transmit mode when set.
1	MDLTE	This bit activates Modulated output mode when set.
2	IRURT	This bit chooses the IR I/O port when set.
3	MDSEL	This bit chooses whether the 32kHz clock or the Baud rate divisor 16x clock is used as a modulation source for both Modulated output and UART modes. If set, the baud rate divisor 16x clock is chosen.
4	PMOD	This bit chooses whether or not the UART mode uses a single pulse or modulated output. If set, the output from the UART sends out 6 pulses instead of 1 for a space or logic 0 bit.
5	LBF	LED Buffer Full. This bit is used in REDEYE mode to indicate that the contents of LBR have not yet been transmitted. Writing to LBR sets this bit.
6	ELBE	Enable Interrupt on LBR bit Empty. If this bit is set, an interrupt will occur if LBF is cleared
7	UNREG	This bit should never be set. It may cause hardware damage.

NOTE: Only one of bits 0-2 should be set at a time.

IRCNT Register
I/O Address E30Bh

Bit	Name	Description
0	LED	This bit is used to turn on the IR LED. When set the LED is turned on. Care should be taken to avoid setting this bit for long periods of time or hardware damage may occur.
1	LBR	This bit contains the half-bit to be transmitted in REDEYE format. A one will send an 'on' half-bit
2	MDLD	This bit is used to turn on and off the modulated LED output in Modulated output mode (MDLTE=1). When set, a modulated waveform is output.
3-4	Undefined	
5	IRE	IR Event. This bit is set by an input pulse from the IR receiver to indicate that an IR event has occurred. It must be reset by writing a '0' to it.
6	EIRI	Enable IR Interrupt. If this bit is set, an IR interrupt will occur when the IRI bit is set.
7	IRI	IR Input pin. This allow the output of the IR receiver to be monitored. It is a read only bit

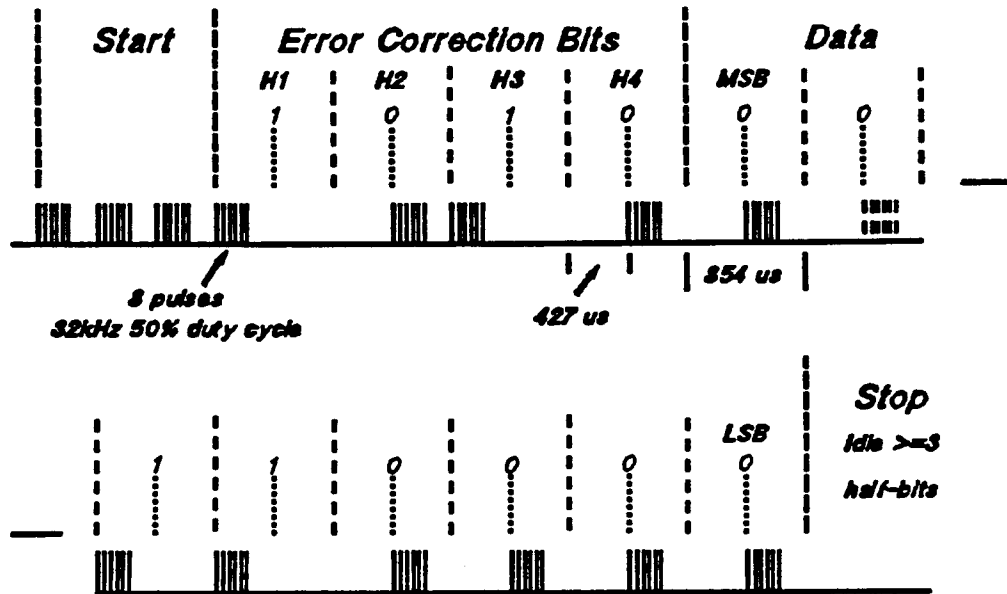
REDEYE Format

The REDEYE portion of the *HP95LX* had not yet been fully tested as of the writing of this document and therefore is not guaranteed to function completely. The hardware consists of the RED, LBF and ELBE bits in the IRFMAT register and the LBR bit in the IRCNT register. The REDEYE format consists of 15 bit frames of a precise format. Each bit of the frame consists of two half-bits. The duration of each half-bit is 427 mseconds. The half-bit is considered 'on' if the LED is pulsed 6-8 times during the half-bit time. *HP95LX* uses 8 pulses. The pulses and all of the timing is generated by using the 32,768 Hz oscillator. Each bit is encoded for transmission. A one is encoded as two half-bits, the first one 'on' and the second 'off'. A zero is transmitted by an 'off-on' sequence. Each frame contains:

Start-bits	Three half-bits 'on-on-on'
Hamming bits	Four bits for error correction
Data bits	Eight data bits
Stop bits	Three half-bits 'off-off-off'

The format of a complete REDEYE frame is as follows:

REDEYE Frame



To enter REDEYE mode, the RED bit must be set. All REDEYE frames must be generated by software. All that the hardware is designed to do is output half-bits. REDEYE transmission is initiated by a write to the LBR bit. This sets the LBF flag and starts the half-bit transmission. The bit in LBR is transferred to the formatter and the LBF bit is cleared. This indicates that it is safe to write another half-bit to the LBR. If the ELBE bit is set, this will cause an interrupt. After a full half-bit transmission time, if the LBF is clear, the output will remain idle. Otherwise, the next half-bit will immediately be transmitted. After LBF is cleared, there are 396 mseconds in which to write the next half-bit to LBR avoid REDEYE transmission errors.

Software Controlled Mode

The LED bit in the IRCNT is provided for software generated IR output formats. Due to LED current limitations, the duty cycle of this wave form should be limited to a time average of 29%.

Software can also monitor the IR receiver to receive incoming data. This can be done using the IRE, IRI, and EIRI bits in the IRCNT register. The IR pulses into the receiver can be stretched by as much as 300 ms from the end of light transmission, so this must be taken into account by the software.

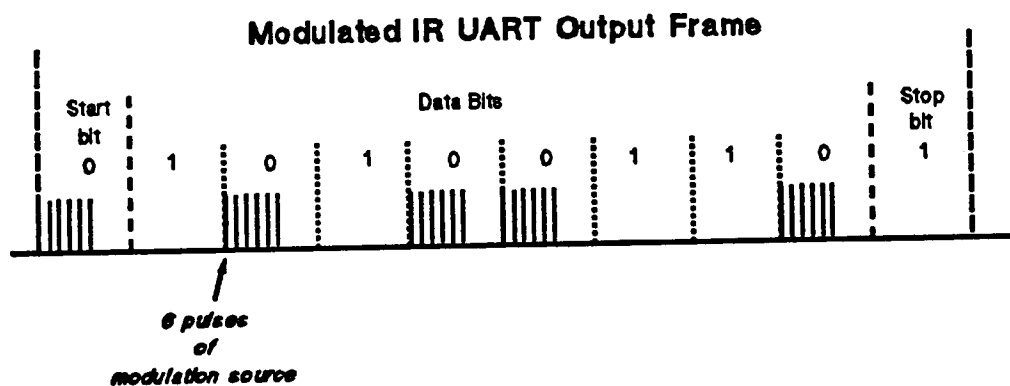
Modulated Output Mode

A Modulated output mode is provided and can be used to output any custom modulated wave form desired. To choose Modulated mode, the MDLTE bit in the IRFMAT register must be set. Two different modulation sources are available. The 32,768 Hz clock can be used, as well as the baud rate 16x clock. The MDSEL bit is set to a 1 to select the baud rate 16x clock. Otherwise, the 32768 Hz

clock is used. If the baud rate 16x clock is chosen, the baud rate must be set in the UART to obtain the desired modulation frequency. If a baud rate of 2400 is chosen the modulation frequency is 38.4 kHz. After these bits have been initialized, the MDLD bit in the IRCNT register is used to output modulated signals. Whenever the MDLD bit is set to a one, the LED is modulated by the modulation source. The output of the modulation source will be a 50% duty cycle if the 32kHz clock is used. If the baud rate divisor 16x clock is used, the on time for each pulse will be 3.25 ms. Care must be taken not to exceed a 29% communication duty cycle.

Modulated IR UART Output Mode

The IR UART function, described previously as it is used for the IR I/O port, can also use modulated output mode. In this mode, 6 pulses of the modulation source will be sent out instead of a single pulse. To select this output mode, the PMOD bit in the IRFMAT register must be set along with the IRURT bit. The modulation source is again chosen using the MDSEL bit. The 8250 compatible UART is again used to output data as in the IR I/O port. An output waveform is shown below.



Custom Artwork

Keyboard Overlay

Custom overlays can be made for the HP 95LX. They can be made to either lay over the standard overlay, or they can be made to stick on permanently. Northern Engraving does the standard keyboard overlays. They use a tool that was supplied by Hewlett Packard. They are willing to manufacture custom overlays using this tool for Hewlett Packard approved Independent Software Vendors. They are able to generate custom artwork to vendor specifications, or they can use artwork supplied by the vendor.

If you are interested in custom overlays for your HP 95LX application, then you should contact:

Richard Kirby
Hewlett Packard
1000 NE Circle Blvd.
Corvallis, Oregon 97330
Telephone: (503) 750-2360

IC Cards

If required, Epson can provide custom artwork printed on the IC cards. Information on submitting artwork designs to Epson is included in the materials that should be obtained directly from Epson America. To get materials from Epson, contact:

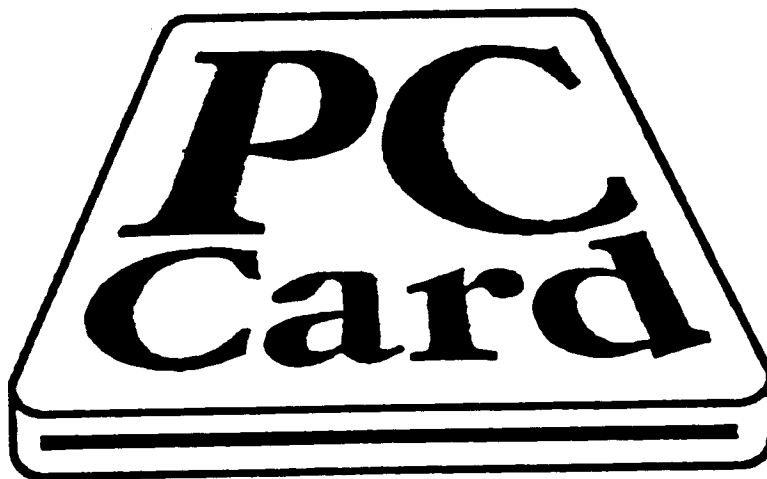
David Rifkin
Epson America
20770 Madrona Avenue
Torrence, California 90509-2842

Alternatively, labels or silkscreening can be applied to the cards after they have been delivered from Epson.

PC CARD STANDARD

Release 1.0

August 1990



**Personal Computer Memory Card International Association
PCMCIA**

Copyright 1990 Personal Computer Memory Card International Association. All Rights reserved.

THIS DOCUMENT MAY NOT BE COPIED, MODIFIED OR DISTRIBUTED, EXCEPT THAT MEMBERS OF PCMCIA MAY COPY THIS DOCUMENT FOR DISTRIBUTION AND USE WITHIN THEIR ORGANISATIONS PROVIDED THE DOCUMENT IS COPIED IN ITS ENTIRETY AND ALL COPYRIGHT AND OTHER NOTICES CONTAINED IN THE ORIGINAL ARE REPRODUCED IN EACH COPY.

THIS DOCUMENT CONTAINS MATERIAL COPYRIGHTED BY JAPAN ELECTRONIC INDUSTRY DEVELOPMENT ASSOCIATION (JEIDA).

1989 - 90 Officers

Chairman:

John Reimer

Fujitsu Microelectronics Inc.

President:

Jim Prelack

Lotus Development

Marketing Committee:

Neal Chandra

Poqet Computer

Standards Committee:

Daniel Chen

Mitsubishi Electronics America

REVISIONS

Initial Release	27 March 1990
2nd Revision	4 May 1990
3rd Revision	14 June 1990
4th Revision	12 July 1990
Release 1.0	21 August 1990

**Standards Committee
1989 - 1990**

Committee Chairman:

Daniel Chen

Mitsubishi Electronics America

Task Force Chair:

Terry Moore

Databook

Stan Sharp

ITT Cannon

Mike Dryfoos

Microsoft

Reneé Bader

Poqet Computer

Committee Members:

Art Lesh

AMP

Ken Jacobsen

Atari Corporation

Hidemaru Sato

Citizen Systems

Don Vendetti

Data I/O

Chuck Brewer

Digital Research

Chris Walke

Dupont Electronics

Richard Vincent

Epson

*** Joel Urban**

Fujitsu Components

Phil Ackerly

Fujitsu Microelectronics

Howard Honig

Hewlett Packard, Corvallis Division

Tony Wutka

IBM

Kurt Robinson

Intel

Bill Claff

Lotus

Roger Fearing

Microlytics

Tom Cruise

Molex International

Jim Clayton

Motorola, Inc.

*** Ray Salas**

NEC Technologies

Jeffrey Glacchetti

Shigma/Fujisoku

Ching Jeng

Silicon Storage Technology, Inc.

Mark Cummings

SRI

Steve Gross

SunDisk Corp.

Brady Le Blanc

Techworks

Daniel Baudouin

Texas Instruments

Avram Grossman

Toshiba

*** Committee Co-chair**

PREFACE

An extraordinary ammount of progress has taken place over the past twelve months in the development and growth of memory card technology. With the formation of the PCMCIA in mid-1989, a focus for the computer industry's interest in this memory card technology was created and work was begun on this standard.

The handful of companies who joined last year are now surrounded by dozens of additional member companies who are adding their collective interest and energy to the effort to standardize this memory card technology. The current PCMCIA membership consists of a wide variety of large and small companies, based in the United States, Europe, Japan and other countries. Participating in the creation of the standard were companies representing computer manufacturers, software suppliers, system integrators, hardware manufacturers and semiconductor memory card manufacturers.

This initial Release 1.0 of the PCMCIA standard, entitled PC Card Standard, reflects the effort and perseverance of many individuals who have spent countless late nights and air-miles in the effort to create this document. Special thanks to Daniel Chen, PCMCIA Technical Committee chair, and to all of the sub-committee chairpeople who contributed a disproportionately large amount of time and effort.

This standard has been created by the PCMCIA with the cooperation of the Japanese Electronic Industry Development Association (JEIDA).

Future releases of this standard will provide for additional capabilities beyond those supported by this initial release

Board of Directors
PCMCIA
August 1990

INTRODUCTION

Among the many applications of I.C. memory cards, one of the most appealing is to use them to replace slow, fragile, bulky and power-wasting magnetic disk drives. This standard describes a family of memory standards used as mass-storage devices in computer systems.

The intent of this standard is not to impose a single philosophy on all card applications. Rather, we want to develop a standard that will:

- Be appropriate (in its minimal form) for use with very small memory cards for example, appliance memory modules.
- Allow the use of different formats for recording data. The marketplace will decide which format becomes predominant.
- Recognize the legitimate requirements of specialized applications, and establish means by which OEMs can develop new data formats for their specific applications.

CONTENTS

	Page
1.0 GENERAL	10
2.0 SCOPE	10
3.0 CARD PHYSICAL DIMENSIONS	11
4.0 CARD INTERFACE	27
4.0 Memory Card Features	29
4.1 Signal Description	29
4.2 Operating Conditions	32
4.3 Memory Function	32
4.4 Timing Function	36
4.5 Electrical Interface	39
4.6 Card Detect	40
4.7 Battery Voltage Detect	40
4.8 Power-up and Power-down	41
4.9 Future Tasks	42
5.0 CARD METAFORMAT	43
5.1 The Standard	44
5.2 Basic Compatibility (Layer 1)	49
5.3 Data Recording Formats (Layer 2)	65
5.4 Data Organization (Layer 3)	80
5.5 System-Specific Standards (Layer 4)	82
5.6 Compatibility Issues	86
6.0 FAT FILE SYSTEM	89
7.0 EXECUTE IN PLACE	93
APPENDIX 1 – Metafont Glossary	96
APPENDIX 2 – Hot/Cold Insertion Removal	100
APPENDIX 3 – Recommended Testing Method for Hot-Insertion and Removal	102

LIST OF TABLES

		Page
1	IC Memory Card Dimensions	26
2	Host Connector Pin Configuration	26
3	Features of PCMCIA Memory Card	29
4	Memory Types and Speed Version	29
5	Operating Conditions	32
6	Main Memory Read Function for all types of Memory	33
7	Main Memory Write Function for SRAM, EEPROM and Single Supply FLASH Card	33
8	Main Memory Write Function for OTPROM, EPROM and FLASH EPROM	34
9	Attribute Memory Read Function	34
10	Attribute Memory Write Function for SRAM and Single Supply EEPROM	34
11	Attribute Memory Write Function for OTPROM, EPROM, Dual Supply EEPROM, FLASH-EEPROM and EEPROM	35
12	Write Protect Functions	35
13	Main Memory Read Timing Specification for all types of Memory	36
14	Main Memory Write Timing Specification SRAM	36
15	Attribute Memory Read Timing Specification for all types of Memory	37
16	Electrical Interface	39
17	Battery Voltage Detect	40
18	Power-up/Power-down Timing	41
19	Tuple Format	49
20	Tuple Codes	50
21	Null Control Tuple	53
22	Long Link Tuple	53
23	Link Target Tuple	54
24	No-Link tuple	54
25	End-of-List Control Tuple	55
26	Checksum Tuple	56
27	Alternate Language String Tuple	57
28	Device Information Tuples	59
29	Device ID	60
30	Device Speed Codes	60
31	Extended Device Speed Codes	61
32	Device Type Codes	61
33	Level 1 Version / Product Information Tuple	62
34	The JEDEC Identifier Tuples	63
35	Level-2 Information Tuple	66
36	Card Initialization Date Tuple	68
37	Battery Replacement Date Tuple	69
38	Format Tuple	70
39	Format Type Codes	71
40	Error Detection Type Codes	71
41	Format Tuple for Disk-like Regions	72
42	Error Detection Format Summary	73
43	Format Tuple for Memory-like Regions	74
44	Geometry Tuple	76
45	Byte Order Tuple	77
46	Byte Order Codes	77
47	Byte Mapping Codes	77
48	Data Organization Tuple	80
49	Data Organization Codes	81
50	DOS Boot-Block Structure	84
51	Extended BPB	84
52	Boot Record Format for Small Partitions	89
53	Boot Record Format for Large Partitions	90

LIST OF FIGURES

	Page
1 TYPE I PC Card Package Dimensions	18
2 TYPE II PC Card Package Dimensions	19
3 TYPE I PC	20
4 TYPE II PC	20
5 Thickness of Label	21
6 Card Connector Socket	22
7 Connector	22
8 PC Memory Card Contact Pins	23
9 Recommended Right Angle Connector PCB Footprint	24
10 Recommended Straight Connector PCB Footprint	24
11 Memory Card Guide	25
12 Read Timing Chart	37
13 Write Timing Chart (WE control)	38
14 Write Timing Chart (CE control)	38
15 Card Detect	40
16 Power-up/Power-down Timing	41

General/Scope

1.0 General

The PC Card system design guideline is endorsed by PCMCIA member companies as well as Japan Electronic Industry Development Association (JEIDA).

2.0 Scope

This design guideline details mechanical, electrical interface, host interface protocol, and data format of a parallel type IC card assembly.

SECTION 3
CARD PHYSICAL DIMENSIONS

Card Physical Dimensions

3.0 Card Physical Dimensions

This section of the specification defines the card physical outline dimensions, connector system, connector reliability, connector durability and PC card guidance system.

3.1 Card Dimensions

Two types of PC Cards are specified within this specification. The two types are Type I (see Figure 1) and Type II (see Figure 2). The Type I and Type II PC Cards differ in thickness. The Type I PC Card is preferred and the Type II PC Card is optional. The Type II PC Card thickness is greater in the substrate area (see Figures 2 & 4)

3.1.1 The PC Card dimensions for the Type I and Type II are shown in Table 1.

3.1.2 The connector location and pin numbers for the Type I and Type II PC Cards are shown in Figures 1 and 2.

3.1.3 The PC Card polarization technique and dimensions are shown in Figures 1 and 2. A mismatched PC Card and connector shall withstand a minimum 22 pounds (10Kg) static load without damage to the PC Card or connector.

3.1.4 The PC Card must be opaque (non see-through).

3.1.5 Write Protect Switch (WPS)

3.1.5.1 The WPS, if installed, shall be located to the right of the PC Card centerline when viewed from the end opposite the connector (see Figures 1, 2, 3 & 4).

3.1.5.2 The write protected position of the WPS shall be the far right position. The write protected switch position shall be indicated by an arrow and either "Write Protect" or "Protect" indicated. The arrow and indication may be indicated in the PC Card end as shown in Figures 1, 2, 3 & 4, on the bottom cover as indicated in Figure 5 or on both the end and bottom cover.

3.1.6 Battery Location

3.1.6.1 The battery, if installed, shall be located to the left of the PC Card centerline when viewed from the end opposite the connector (see Figures 1, 2, 3 & 4).

3.1.6.2 The battery holder, if installed, should be designed so that the positive (+) side of the battery faces the top surface.

3.1.7 Label

3.1.7.1 The thickness of the label, if used, (see Figure 5) shall not cause the PC Card to exceed the thickness specified in Table 1.

3.1.7.2 The label, if used, must withstand any environmental test specified by PC Card specifications.

3.1.7.3 The JEIDA and PCMCIA logo location is shown in Figure 5. The JEIDA and PCMCIA logos may be displayed on the label if authorized by the respective organizations.

3.2.0 Connector

The PC Card interconnect system specified shall be a 68 position 2 piece pin and socket. The socket contacts shall be on the PC Card memory card connector.

3.2.1 The socket contacts are located on the PC Card as shown in Figures 1,2,3 & 4.

3.2.2 The PC Card connector socket shall be configured as shown in Figure 6.

3.2.3 The PC Card connector socket layout shall match the host pin connector socket layout as shown in Figure 7.

3.2.4 Host Connector

The host pin connector shall be a 68 pin connector. The host pin connector opening polarization and pin location shall be as shown in Figure 7.

3.2.5 The host connector pin configuration is shown in Figure 8.

3.2.6 The host pin lengths are shown in Table 2.

3.2.7 The socket and pins contact area outermost plating shall be Gold or other plated materials which are compatible with Gold and meet the requirements specified in Paragraph 3.4.1.

3.2.8 The recommended host connector PCB footprints for the right angle connector (Figure 9) and the straight connector (Figure 10) are shown without mounting or hardware hole locations.

3.2.9 The interconnect system shall pass all requirements of Paragraph 3.4.0 (Connector reliability) and Paragraph 3.5.0 (Connector durability).

3.2.10 It is recommended, if a connector ejector mechanism is used, the connector mechanism pass all requirements, as applicable in Paragraphs 3.4.0 and 3.5.0 for reliability and durability.

3.3.0 PC Card Guidance (see Figure 11).

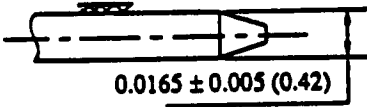
3.3.1 The PC Card shall be guided by the host connector for a minimum distance of .197" (5.0) before engagement.

3.3.2 To ensure alignment of the PC Card to connectors, the PC Card should be guided for a minimum distance of 1.570" (40.0) before engagement.

Card Physical Dimensions

3.4.0 Connector Reliability

The interconnect system is specified in Paragraph 3.2.0 shall meet or exceed all reliability test requirements of this paragraph.

No.	Item	Standard	Testing
3.4.1 Mechanical Performance	1. Office Environment	Guaranteed number of Insertions/Ejections 10,000 MIN.	See Paragraph 3.5.1
	2. Harsh Environment	Guaranteed number of Insertions/Ejections 5,000 MIN.	See Paragraph 3.5.2
	3. Total insertion force	8.8 lbs (4Kg) MAX.	Insert and extract at speed of 1" (25mm)/min
	4. Total pulling force	1.5 lbs (.68Kg) MIN.	Insert and extract at speed of 1" (25mm)/min
	5. Single pin pulling force	.022 lbs (10gr) MIN.  Gauge: Material - Tool making steel Hardness - HRC = 50 to 55	Pull the gauge pin shown at left at speed of 1" (25mm)/min. Gauge pin's surface must be wiped clean of dirt and lubrication oil.
	6. Pin holding force	2.2 lbs (1Kgs) MAX.	Push pins on the axis at speed of 1" (25mm)/min
	7. Vibration and high frequency	a. No mechanical defects should occur on the parts. b. Must not cause current interruption of 100ns or more.	MIL-STD-202F METHOD 204B, Test condition B (15G peak), 10Hz 2000Hz; See detail 1
	8. Shock	a. No mechanical defects should occur on the parts. b. Must not cause current interruption of 100ns or more.	MIL-STD-202F METHOD 213B, Acceleration 50G, Standard holding time 6 ms, semisine wave; See detail 1

No.	Item	Standard	Testing
3.4.2 <u>Electrical</u> <u>Performance</u>	1. Contact resistance (low level)	a. Initially 40 mΩ MAX. b. After test 20 mΩ MAX.	MIL-STD-1344A METHOD 3002.1 Open voltage ≤ 20 mV Test current 1 mA
	2. Withstandable voltage	a. No shorting or other damages when 500 Vrms AC is applied for 1 minute b. Current leakage 1 mA MAX.	MIL-STD-202F METHOD 301
	3. Insulation resistance	a. Initially 1,000MΩ MIN. b. After test 100MΩ MIN.	MIL-STD-202F METHOD 302 measure within 1 minute after applying 500 V DC
	4. Current capacity	0.5 A per pin	
	5. Insulation material	UL94V 0 equivalent	
3.4.3 <u>Environ-</u> <u>mental</u> <u>Performance</u>	1. Operating environment	Operating temperature: -20°C to +60°C Relative humidity: 95% MAX. (Non-condensing)	
	2. Storage environment	Storage temperature: -40°C to +70°C Relative humidity: 95% MAX. (Non-condensing)	

Card Physical Dimensions

No.	Item	Standard	Testing
3.4.4 Environ- mental Resistance	1. Moisture resistance	Contact resistance 3.4.2.1.b. Insulation resistance 3.4.2.3.b.	MIL-STD-202F METHOD 106E (excluding vibration); 10 cycles (1 cycle = 24 hours) with connectors engaged.
	2. Terminal shock	No physical damage should occur during testing. Contact resistance 3.4.2.1.b. Insulation resistance 3.4.2.3.b.	MIL-STD-202F METHOD 107G Test cond. A, -55°C to +85°C 5 cycles (1 cycle = 1 hour) with connectors engaged.
	3. Durability (High temperature)	Contact resistance 3.4.2.1.b.	MIL-STD-202F METHOD 108A Test cond. B, 85°C, 250 hours with connectors engaged
	4. Cold resistance	Contact resistance 3.4.2.1.b.	JIS C 5021, -55°C, 96 hours with connectors engaged
	5. Humidity (normal condition)	Contact resistance 3.4.2.1.b. Insulation resistance 3.4.2.3.b.	MIL-STD-202F METHOD 103B Test cond. B, 40°C, 90 to 95% RH with connectors engaged.
	6. Hydrogen sulfide	Contact resistance 3.4.2.1.b.	JEIDA 38 3ppm 40°C, approx 80% RH 96 hours, with connectors engaged
	7. Salt water spray	No harmful corrosion (to contact performance) should occur on the pin and socket contacts.	MIL-STD-202F METHOD 101D Test cond. B, Concentration 5% 35°C, 48 hours, with connectors disengaged.

3.5.0 Connector Durability

The interconnect system as specified in Paragraph 3.2.0 shall meet or exceed all durability requirements of this paragraph.

Test conditions for the mate/unmate cycles are:

Cycle rate	400-600 cycles per hour
Temperature	15 to 35°C (59 to 95°F)
Relative Humidity	30-80%
Barometric pressure	24-31 inches of Mercury

3.5.1 Office Environment

The office environment is defined in EIA-364A as class 1.1 - year round air conditioning (non-filtered) with humidity control.

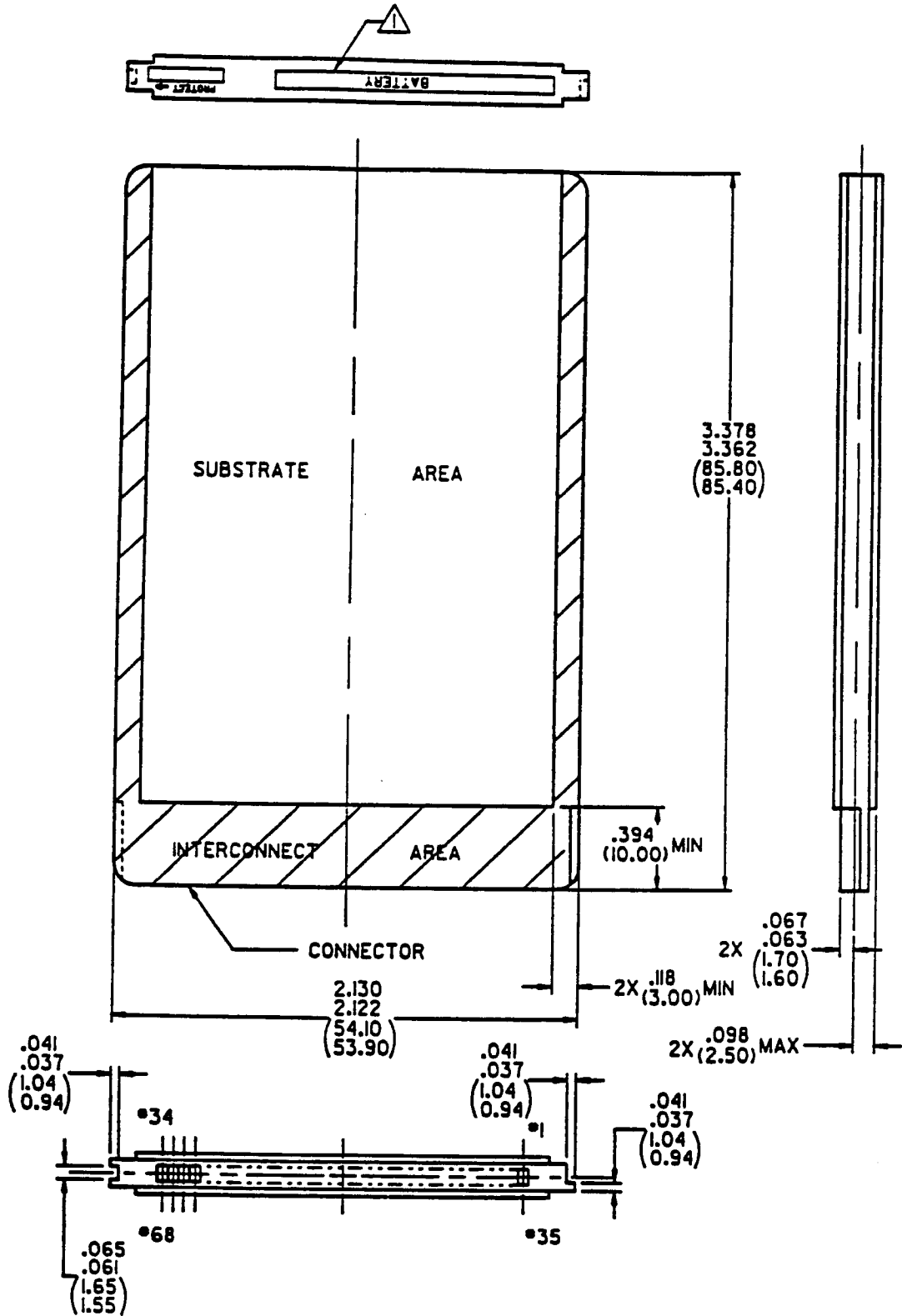
Test Sequence

- Contact Resistance per test method 3.4.2.1.A
- Mate and unmate the connector for a total of 10,000 cycles
- Contact Resistance per test method 3.4.2.1.B

3.5.2 Harsh Environment

The harsh environment is defined in EIA-364A as class 1.3 - no air conditioning, no humidity control with normal heating and ventilation.

Contact Resistance per test method 3.4.2.1.A	
Mate and unmate the connector 1,000 cycles	TOTAL CYCLES = 1,000
Humidity Resistance per test method 3.4.4.5	(1 cycle = 24hours)
Mate and unmate the connector 1,000 cycles	TOTAL CYCLES = 2,000
Humidity Resistance per test method 3.4.4.5	(1 cycle = 24hours)
Mate and unmate the connector 3,000 cycles	TOTAL CYCLES = 5,000
Humidity Resistance per test method 3.4.4.5	(1 cycle = 24hours)
Hydrogen Sulfide test per method 3.4.4.6	
Contact Resistance per test method 3.4.2.1.B	



⚠ RECOMMENDED BATTERY LOCATION. THE BATTERY HOLDER SHOULD BE DESIGNED SO THAT THE POSITIVE SIDE OF THE BATTERY IS UP.

2. THE IC MEMORY CARD SHALL BE OPAQUE (NON SEE THRU)

Fig 2. TYPE II PC Card Package Dimensions

Card Physical Dimensions

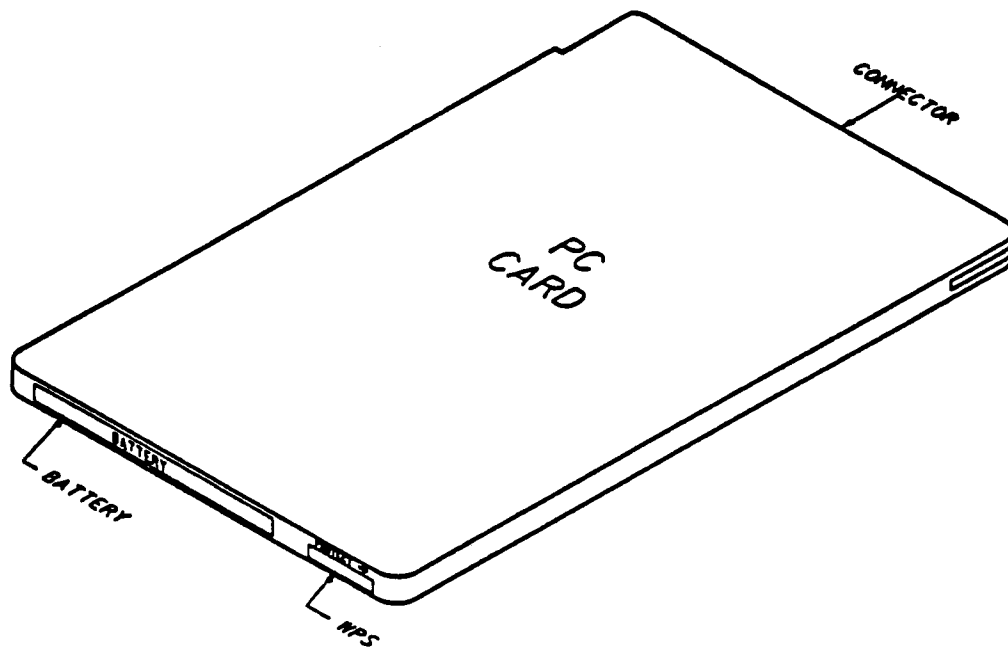


Fig 3. TYPE I PC

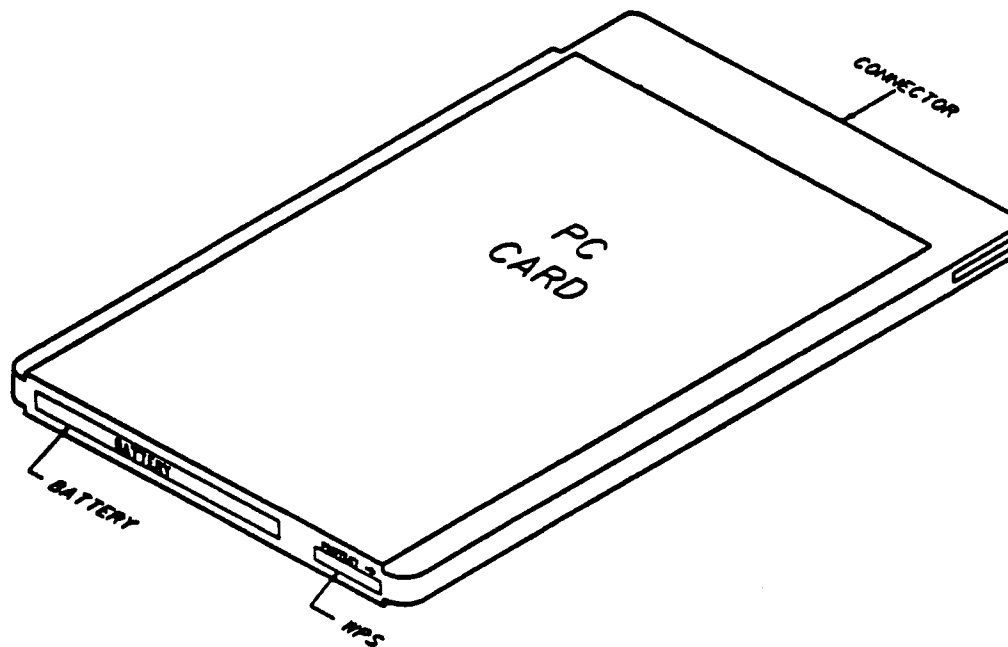
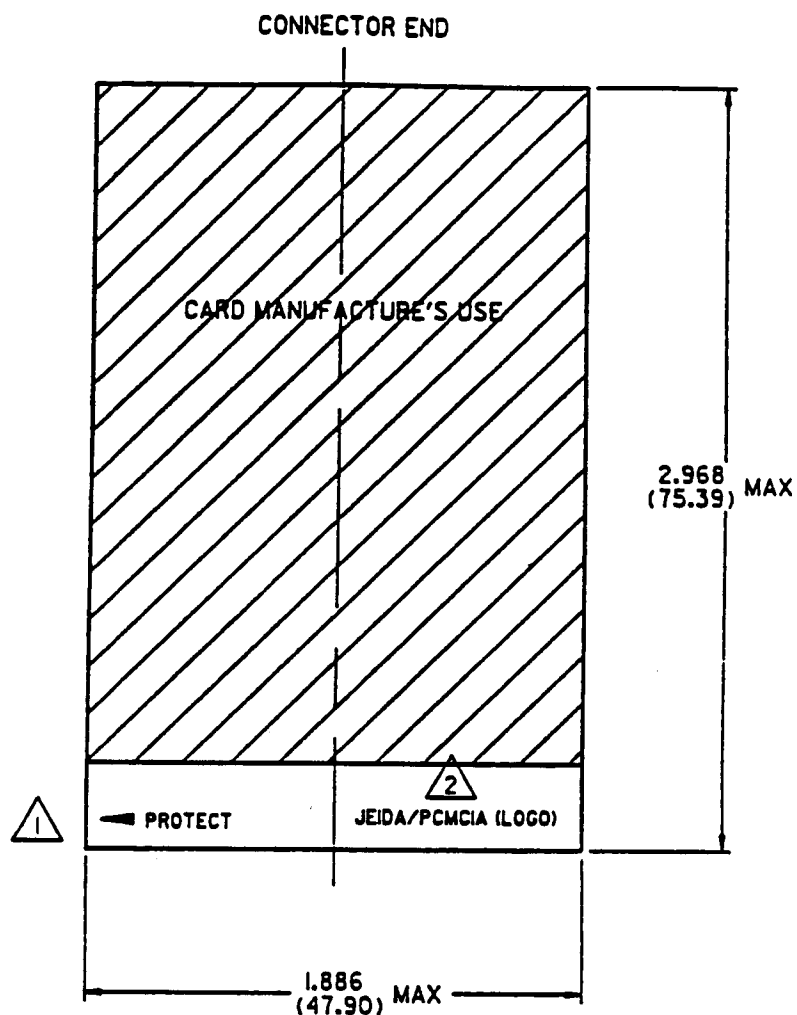


Fig 4. TYPE II PC



- △1 IF WRITE PROTECT SWITCH INSTALLED
△2 OTHER REGULATORY LOGOS.

NOTE: Labels must withstand all environmental tests as specified.

Fig 5. PC Card Label

Card Physical Dimensions

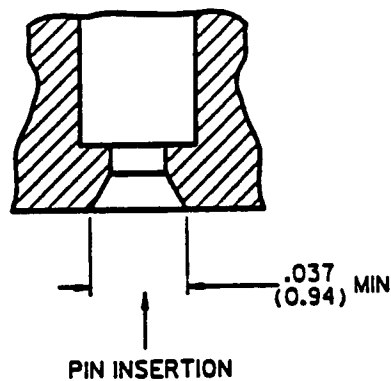


Fig 6. Card Connector Socket

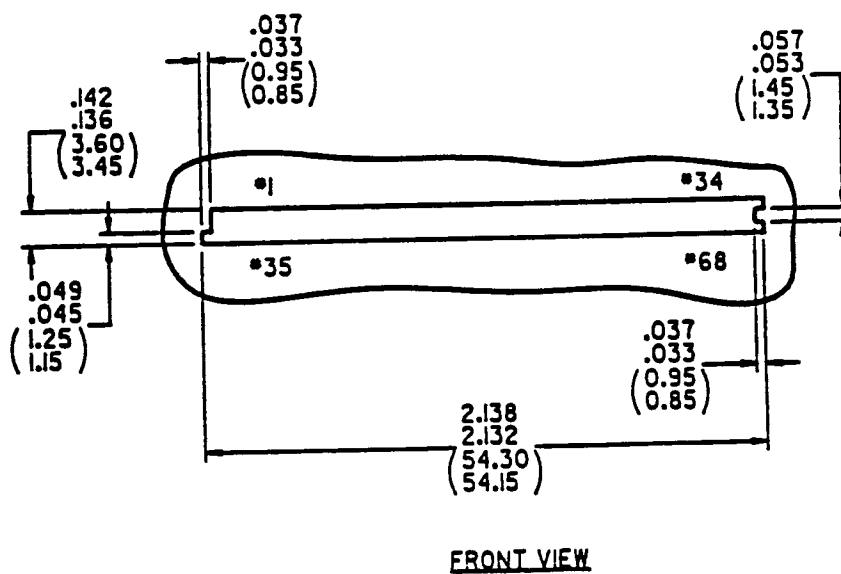
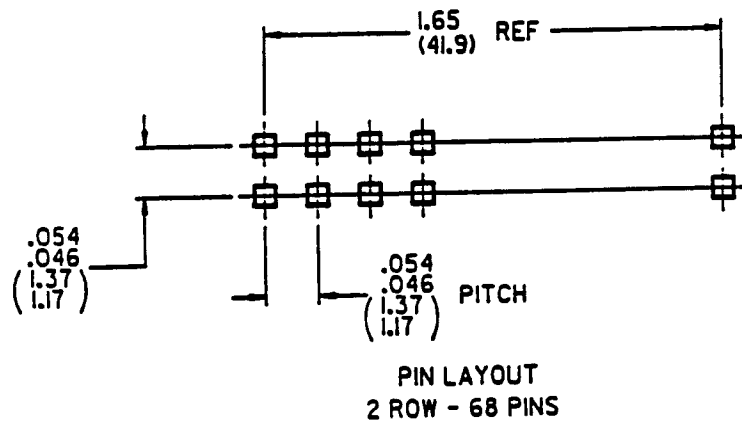
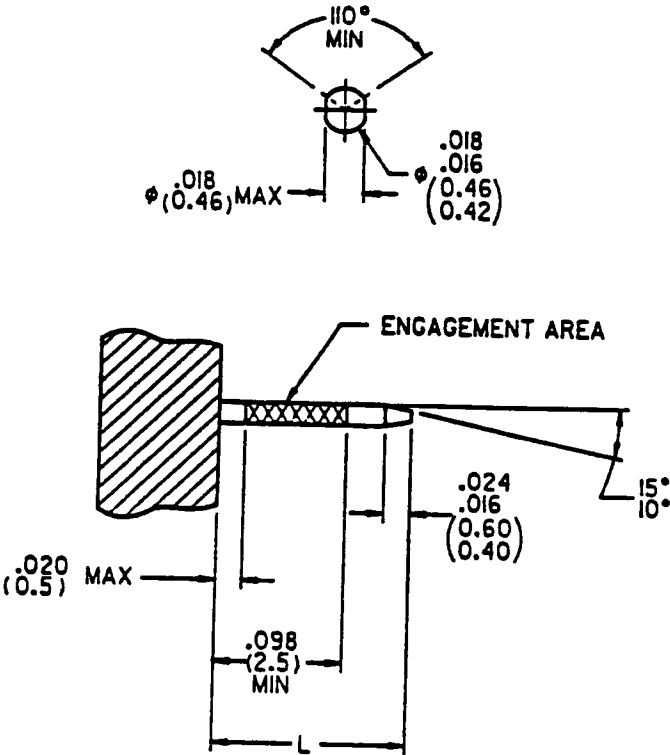


Fig 7. Connector



NOTE: Length "L" given in Table 2.

Fig 8. PC Card Contact Pins

Technical drawing of a card reader assembly. The drawing shows a side view of the assembly with various components labeled with numbers and dimensions. The assembly is symmetrical about a vertical centerline, indicated by a dashed line and the text "INSERT CARD" at the top. The main body of the assembly is labeled with dimensions: 1.654, 1.648, (42.01), and (41.81). The top of the assembly is labeled with dimensions: .052, .048, and (1.32). The bottom of the assembly is labeled with dimensions: .077, .073, and (1.955). The right side of the assembly is labeled with dimensions: .225, (5.715), and REF. The left side of the assembly is labeled with dimensions: .077, .073, and (1.955). The assembly is composed of several parts, including a card reader (1), a card (2), a card holder (3), a card guide (4), a card support (5), a card reader (6), a card (7), a card holder (8), a card guide (9), a card support (10), a card reader (11), a card (12), a card holder (13), a card guide (14), a card support (15), a card reader (16), a card (17), a card holder (18), a card guide (19), a card support (20), a card reader (21), a card (22), a card holder (23), a card guide (24), a card support (25), a card reader (26), a card (27), a card holder (28), a card guide (29), a card support (30), a card reader (31), a card (32), a card holder (33), a card guide (34), a card support (35), a card reader (36), a card (37), a card holder (38), a card guide (39), a card support (40), a card reader (41), a card (42), a card holder (43), a card guide (44), a card support (45), a card reader (46), a card (47), a card holder (48), a card guide (49), a card support (50), a card reader (51), a card (52), a card holder (53), a card guide (54), a card support (55), a card reader (56), a card (57), a card holder (58), a card guide (59), a card support (60), a card reader (61), a card (62), a card holder (63), a card guide (64), a card support (65), a card reader (66), a card (67), a card holder (68), a card guide (69), a card support (70), a card reader (71), a card (72), a card holder (73), a card guide (74), a card support (75), a card reader (76), a card (77), a card holder (78), a card guide (79), a card support (80), a card reader (81), a card (82), a card holder (83), a card guide (84), a card support (85), a card reader (86), a card (87), a card holder (88), a card guide (89), a card support (90), a card reader (91), a card (92), a card holder (93), a card guide (94), a card support (95), a card reader (96), a card (97), a card holder (98), a card guide (99), a card support (100).

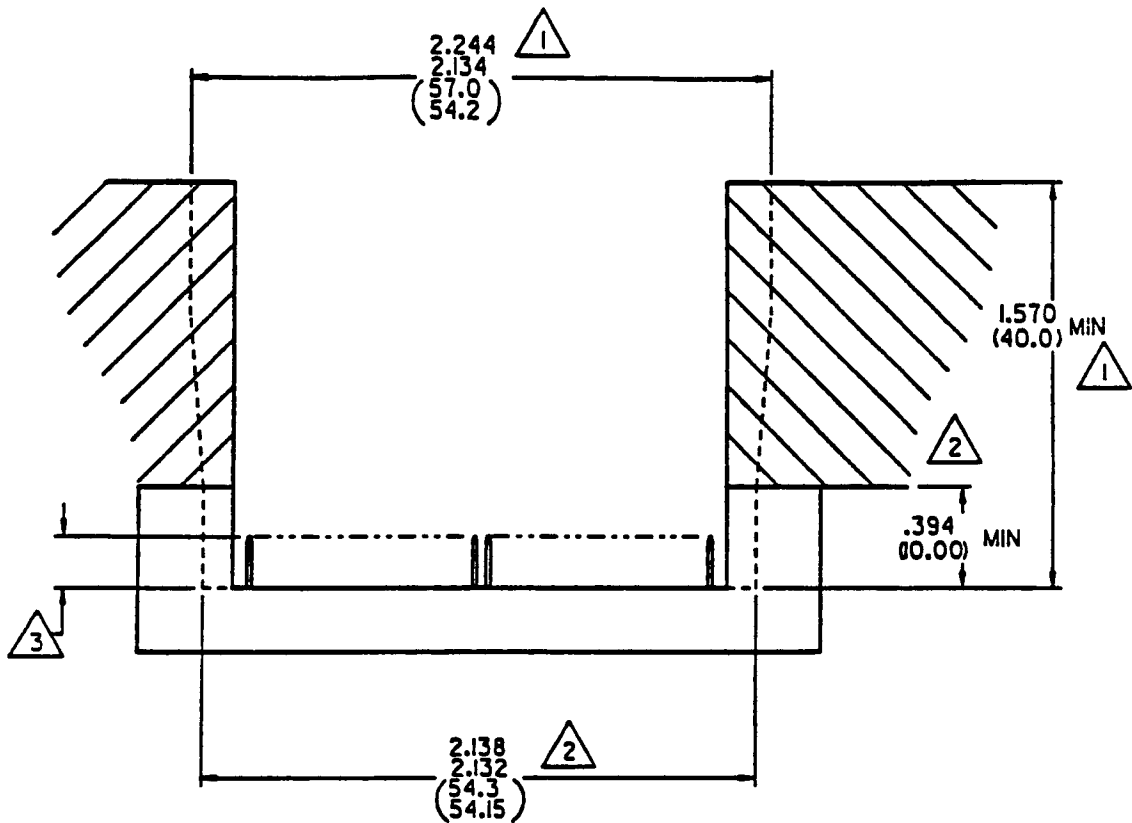
Technical drawing of a rectangular plate with dimensions and feature callouts. The drawing shows a rectangular plate with a central vertical line. Dimensions are given in inches and millimeters. Callouts include:

- *1: Top edge
- *2: Left edge
- *33: Top edge
- *34: Top edge
- *35: Left edge
- *36: Bottom edge
- *67: Bottom edge
- *68: Bottom edge

Dimensions:

- Overall width: 1.654 (42.01)
- Overall height: 1.648 (41.81)
- Distance from left edge to center line: 1.654 (42.01)
- Distance from center line to right edge: 1.648 (41.81)
- Distance from bottom edge to top edge: .225 (5.715) REF
- Distance from bottom edge to top edge: .077 (1.955)
- Distance from bottom edge to top edge: .073 (1.90)
- Distance from bottom edge to top edge: 3X (1.955)
- Distance from bottom edge to top edge: .052 (1.32)
- Distance from bottom edge to top edge: .048 (1.22)



- 24 -



- 1 IT IS RECOMENDED THAT THE I C MEMORY CARD BE GUIDED FOR A MIN DISTANCE OF 1.570 (40.0).
- 2 THE CONNECTOR MUST GUIDE THE I C MEMORY CARD FOR A MIN DISTANCE OF .197 (5.0) BEFORE ENGAGEMENT.
- 3 MAX PIN LENGTH .201 (5.1)

Fig 11. PC Card Guide Guidance

Card Physical Dimensions

	LENGTH	WIDTH	INTERCONNECT AREA 	SUBSTRATE AREA 
TYPE I	3.370 ±.008 (85.6 ±0.20)	2.126 ±.004 (54.0 ±0.10)	.065 ±.002 (1.65 ±0.06)	.065 ±.002 (1.65 ±0.06)
TYPE II	3.370 ±.008 (85.6 ±0.20)	2.126 ±.004 (54.0 ±0.10)	.065 ±.002 (1.65 ±0.06)	.098 MAX (2.5)

NOTES:

 INTERCONNECT AREA AND SUBSTRATE AREA THICKNESS ARE SPECIFIED FROM THE IC MEMORY CARD CENTER LINE TO EITHER THE TOP OR BOTTOM SURFACE

2. MILLIMETERS ARE IN PARENTHESES ().

Table 1 PC Card Dimensions

PIN TYPE	PIN LENGTH (L)	PIN #
DETECT	.142 (3.6) .134 (3.4)	36, 67
GENERAL	.171 (4.35) .163 (4.15)	ALL OTHER PINS
POWER	.201 (5.1) .193 (4.9)	1, 17, 34, 35, 51, 68

Table 2 Host Connector Pin Configuration

SECTION 4
CARD INTERFACE

Card Interface

4. CARD INTERFACE RESERVED PINS

PCMCIA MEMORY CARD PIN ASSIGNMENTS

Pin	Signal	I/O	Function	+/-
1	GND		Ground	
2	D3	I/O	Data bit 3	
3	D4	I/O	Data bit 4	
4	D5	I/O	Data bit 5	
5	D6	I/O	Data bit 6	
6	D7	I/O	Data bit 7	
7	CE1	I	Card enable	-
8	A10	I	Address bit 10	
9	OE	I	Output enable	-
10	A11	I	Address bit 11	
11	A9	I	Address bit 9	
12	A8	I	Address bit 8	
13	A13	I	Address bit 13	
14	A14	I	Address bit 14	
15	WE/PGM	I	Write enable	-
16	RDY/BSY	O	Ready/busy (EEPROM)	+/-
17	Vcc			
18	Vpp1			
19	A16	I	Address bit 16	
20	A15	I	Address bit 15	
21	A12	I	Address bit 12	
22	A7	I	Address bit 7	
23	A6	I	Address bit 6	
24	A5	I	Address bit 5	
25	A4	I	Address bit 4	
26	A3	I	Address bit 3	
27	A2	I	Address bit 2	
28	A1	I	Address bit 1	
29	A0	I	Address bit 0	
30	D0	I/O	Data bit 0	
31	D1	I/O	Data bit 1	
32	D2	I/O	Data bit 2	
33	WP	O	Write protect	+
34	GND		Ground	

Pin	Signal	I/O	Function	+/-
35	GND		Ground	
36	CD1	O	Card detect	-
37	D11	I/O	Data bit 11	
38	D12	I/O	Data bit 12	
39	D13	I/O	Data bit 13	
40	D14	I/O	Data bit 14	
41	D15	I/O	Data bit 15	
42	CE2	I	Card enable	-
43	RFSH	I	Refresh	
44	RFU		Reserved	
45	RFU		Reserved	
46	A17	I	Address bit 17	
47	A18	I	Address bit 18	
48	A19	I	Address bit 19	
49	A20	I	Address bit 20	
50	A21	I	Address bit 21	
51	Vcc			
52	Vpp2			
53	A22	I	Address bit 22	
54	A23	I	Address bit 23	
55	A24	I	Address bit 24	
56	A25	I	Address bit 25	
57	RFU		Reserved	
58	RFU		Reserved	
59	RFU		Reserved	
60	RFU		Reserved	
61	REG	I	Register select	-
62	BVD2	O	Battery voltage detect 2	
63	BVD1	O	Battery voltage detect 1	
64	D8	I/O	Data bit 8	
65	D9	I/O	Data bit 9	
66	D10	I/O	Data bit 10	
67	CD2	O	Card detect	-
68	GND		Ground	

NOTE: Active "low" signals are indicated by -(minus).
Active "high" signals are indicated by +(plus).

4.0 MEMORY CARD FEATURES

Item	Feature
Access	Random Access
Data Bus	Bus 16 bits/8 bits
Memory Types	MaskROM, OTPROM, EPROM, EEPROM, Flash-EPROM, SRAM
Memory Capacity	64MB (A0-A25) maximum
REG function	Attribute Memory for storing card identification

Table 3 Features of PCMCIA Memory Card

4.0.1 Memory Types and Speed Version

Memory Type	Speed Version			
	250ns	200ns	150ns	100ns
SRAM	defined	defined	defined	defined
MaskROM, OTPROM, EPROM EEPROM, Flash-EPROM	defined	defined	defined	not defined

Table 4 Memory Types and Speed Version

4.1 SIGNAL DESCRIPTION

Signals on the PCMCIA interface are considered asserted, (+) within the range 2.0 to 5.25 volts and negated, (-) within the range of 0.0 to 0.5 volts. All signals are considered to be active when the line is asserted; (+) unless the signal name is preceded by the minus sign, (-) when it shall be considered active when the line is negated.

All signals are grouped under 4 classifications, I (Input) O (Output) I/O (Bidirectional) and R (Reserved). Input signals are those driven by the host and Output signals are those driven by the Memory Card

All pins identified as ground shall be connected to signal ground at the host. Signal pins identified as Reserved shall have no connection at the host.

The data path to the memory card, is 16 bits wide and consists of signals D0-D15. The card supports an address bus, of 26 bits, (A0-A25) giving a maximum addressing range of 64 megabytes.

4.1.1 Address BUS (A0-A25)

Signals A0 through A25 are address bus lines driven by the host which enable direct addressing of up to 64 megabytes of memory on the card. Signal A0 is not used in word access mode. Signal A25 is the most significant bit. Bit number and significance decrease downward to A0.

4.1.2 Data BUS (D0-D15)

Signals D0 through D15 constitute the bidirectional data bus. The most significant bit is D15. Bit number and significance decrease downward to D0.

4.1.3 Card Enable (-CE1 & -CE2)

The -CE1 and -CE2 lines are active-low card enable signals driven by the host; -CE1 is used to enable even bytes, -CE2 for odd bytes. A multiplexing scheme based on A0, -CE1 and -CE2 allows 8 bit hosts to access all data on D0..D7 if desired. See table 6.

Reference Section 4.3.1 for additional information regarding (Main Memory Read Functionality).

4.1.4 Output Enable (-OE)

The -OE line is the active-low signal driven by the host which is used to gate read data from the memory card. Memory cards incorporating static RAM fall into two categories: cards for which the -OE signal must be deasserted during write operations, and cards that do not use the -OE signal during Write operations and allow the signal to be in either state.

4.1.5 Write Enable/Program (-WE/-PGM)

The -WE/-PGM signal is driven by the host and used for gating Write data to the memory card. This line is also used for memory cards employing programmable memory technologies. See Section 5 for identification of programmable memory technology cards.

4.1.6 Ready/Busy (+RDY/-BSY)

The +RDY/-BSY line, is driven low by the memory card to indicate that the memory card circuits are busy, and unable to accept a data transfer operation. The +RDY/-BSY signal is set low when the card is busy processing a previous write command. The signal +RDY/-BSY is set high (+), when the memory card is ready to accept a new data transfer command. The Host memory card socket must provide a pull-up resistor. See Table 16, Electrical Interface.

4.1.7 Card Detect (-CD1 & -CD2)

The -CD1 and -CD2 signals provide for proper memory card insertion detection, and have been positioned at opposite ends of the connector to facilitate the detection process. The signals are connected to ground internally on the memory card; thus they will be forced low whenever a card is placed in a host socket. The host socket interface circuitry shall provide 10K pull-up resistors to Vcc on each of these signal pins.

4.1.8 Write Protect (+WP)

The WP output signal is used to reflect the status of the Write Protect switch on the memory card. If the memory card Write Protect switch is present, this signal will be asserted by the card when the switch is enabled, and deasserted when the switch is disabled. If the memory card has no Write Protect switch, the card will connect this line to ground or Vcc, depending on the condition of the card memory. If the card can always be written, the pin will be connected to ground. If the card is permanently Write Protected, the pin will be connected to Vcc.

4.1.9 Attribute Memory Select (-REG)

The -REG signal is kept inactive, (+) for all normal accesses to what is known as the Main Memory of the card. When this signal is active, (-) access is limited to Attribute Memory. Attribute Memory is a separately accessed section of memory on the card and is generally used to record card capacity and other configuration and attribute information. Main Memory is used to store user data.

The timing of Attribute Memory may be different than that of Main Memory, refer to manufacturer's specifications for details. When Attribute Memory is accessed, only data signals D0-D7 are valid and signals D8-D15 shall be ignored. Signals -CE1 and -CE2 and A0 are still valid, but it is only possible to select even addresses, (a combination of -CE1 / -CE2 / A0 that requests an odd byte will result in invalid data on the bus. Ref. Table 9.

For those PC memory cards that do not have a section reserved for Attribute Memory, all Main Memory addresses shall begin with address (hex) 0H and proceed for a minimum of 16 kilobytes of contiguous space.

4.1.10 Battery Voltage Detect (BVD1 & BVD2)

The signals, BVD1 and BVD2 are generated by the memory card, as an indication of the condition of the battery on the memory card.

Both signals are kept asserted when the battery is in good condition. When BVD2 is negated while BVD1 is still asserted, the battery is in a warning condition and should be replaced, although data integrity on the card is still assured. If BVD1 is negated with BVD2 either asserted or negated, the battery is no longer serviceable and data is lost. (Ref. Table 17.)

4.1.11 Program Voltages (VPP1 & VPP2)

The VPP1 and VPP2 signals supply program voltages for programmable memory operation. These pins are to be connected to Vcc when VPP1 or VPP2 are not active and are not being used for altering programmable memory. Refer to section 5 and the Card Information Structure for more information on the characteristics of VPP1 and VPP2.

4.1.12 Card Voltage and Ground (Vcc & GND)

The VCC and GND input pins have been placed at symmetrical positions on the memory card to provide safety in the case of an inverted card insertion. Two power pins and four ground pins are employed to reduce the impedance between the memory card and the system.

Card Interface

4.1.13 Refresh (REFSH)

Intended use is for pseudostatic SRAMS (PSRAM). Will be more clearly defined for use by a future version of this standard.

4.1.14 Reserved Pins RFU

Several pins have been identified as Reserved for Future Use. Neither memory cards, nor Host systems shall make any electrical connections to these pins.

4.2 OPERATING CONDITIONS

Item	Symbol	IEEE Symbol	Conditions
Operating Voltage	VCC		5V \pm 5%
Signal Interface Level	—		TTL or CMOS Level

Table 5 Operating Conditions

4.3 MEMORY FUNCTION

4.3.0 Main Memory Function

This section describes operations of Main Memory Area.

4.3.1 Main Memory Read Function

The memory card can be configured with different types of memory devices, (such as SRAM, MaskROM, etc.). Among all types however, the Read function shares common signal state sequencing.

To access "Main Memory", the signal -REG shall be kept inactive and the signal -OE shall be active during the Read cycle. Signals -CE1 & -CE2 control the activation of the Memory Card and A0 control byte ordering on the data bus lines D0-D15. Table 6 shows the signal states and data bus validity for the Read functions described below.

When both -CE1 and -CE2 are inactive, the card is in standby mode. When either -CE1 or -CE2 become active, (low) the memory card is activated and ready for data transfers.

When -CE1 is active and -CE2 is not active, Byte Access mode is enabled (8-bit transfers). Both the even-byte data and odd- byte data outputs will be valid in data bus lines D0-D7. The selection of an even-byte or an odd-byte is controlled by signal A0.

When using word access (16-bit transfers), both -CE1 and -CE2 are active (low), and the even-byte data and odd-byte data outputs are valid in data bus lines D0-D15. During Word mode, signal A0 is ignored.

Odd-byte Only access is enabled by -CE1 being inactive and -CE2 active. During Odd-byte Only access, only data lines D8-D15 contain valid data and address signal A0 is ignored.

Function Mode	-REG	-CE2	-CE1	A0	-OE	-WE	VPP2	VPP1	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	VCC	VCC	High-Z	High-Z
Byte Access (8bits)	H	H	L	L	L	H	VCC	VCC	High-Z	Even-Byte
	H	H	L	H	L	H	VCC	VCC	High-Z	Odd-Byte
Word Access (16bits)	H	L	L	X	L	H	VCC	VCC	Odd-Byte	Even-Byte
Odd-Byte Only Access	H	L	H	X	L	H	VCC	VCC	Odd-Byte	High-Z

Table 6 Main Memory Read Function for all types of Memory

4.3.2 Main Memory Write Function for SRAM, EEPROM and Single Supply FLASH Card.

During Write mode, the function of signals -REG, -CE1, -CE2 and A0 are the same as in the Read mode.

During Write mode, Signal -OE must be kept inactive, and signal -WE/-PGM is active. The Memory Card can perform Write operations in 3 modes: Byte access, Word access, and Odd-byte Only access. Refer to Table 7 for signal states and data bus validity for Main Memory Write modes.

Function Mode	-REG	-CE2	-CE1	A0	-OE	-WE	VPP2	VPP1	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	VCC	VCC	XXX	XXX
Byte Access (8bits)	H	H	L	L	H	L	VCC	VCC	XXX	Even-Byte
	H	H	L	H	H	L	VCC	VCC	XXX	Odd-Byte
Word Access (16bits)	H	L	L	X	H	L	VCC	VCC	Odd-Byte	Even-Byte
Odd-Byte Only Access	H	L	H	X	H	L	VCC	VCC	Odd-Byte	XXX

Table 7 Main Memory Write Function for SRAM, EEPROM and Single Supply FLASH Card

4.3.3 Main Memory Write Function for OTPROM, EPROM and Flash-EPROM

During the Program function, signals -REG, -CE1, -CE2, A0, -OE, and -WE/-PGM are the same as in the Main Memory Write function for SRAM Cards. In addition, 3 access modes are supported as in the Read and Write functions. Refer to Table 8 for signal states and bus validity regarding the explanations below.

In Byte access Program mode, when A0 is asserted, VPP2 shall be at the programming voltage level and VPP1 inactive. Conversely, when A0 is negated, VPP1 shall be at the programming voltage level and VPP2 shall be inactive. For Odd-byte Only access mode, VPP2 shall be at the programming voltage level and VPP1 inactive, (A0 is not used).

When Word access mode is used, both VPP1 and VPP2 are kept at the programming voltage level.

Card Interface

Function Mode	REG	CE2	CE1	A0	OE	WE	VPP2	VPP1	D15-D8	D7-D0
Write Inhibit	X	H	H	X	X	X	VCCorVPP	VCCorVPP	XXX	XXX
Byte Access (8bits)	H	H	L	L	H	L	VCC	VPP	XXX	Even-Byte
	H	H	L	H	H	L	VPP	VCC	XXX	Odd-Byte
Word Access(16bits)	H	L	L	X	H	L	VPP	VPP	Odd-Byte	Even-Byte
Odd-Byte Only Access	H	L	H	X	H	L	VPP	VCC	Odd-Byte	XXX

Table 8 Main Memory Write Function for OTPROM, EPROM and FLASH EPROM

4.3.4 Attribute Memory Function

Attribute Memory is an optional space intended for storing memory card identification and configuration information, and does not require a large address space. Attribute Memory is limited to 8-bit wide access for economic reasons.

4.3.5 Attribute Memory Read Function

For the Attribute Memory Read function, signals -REG and -OE must be active during the cycle. As in the Main Memory Read function, the signals -CE1 and -CE2 control the even-byte and odd-byte address, but only even-byte data is valid during the Register function. Refer to Table 9 for signal states and bus validity for the Attribute Memory Read function.

Function Mode	REG	CE2	CE1	A0	OE	WE	VPP2	VPP1	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	VCC	VCC	High-Z	High-Z
Byte Access (8bits)	L	H	L	L	L	H	VCC	VCC	High-Z	Even-Byte
	L	H	L	H	L	H	VCC	VCC	High-Z	Not Valid
Byte Access (16bits)	L	L	L	X	L	H	VCC	VCC	Not Valid	Even-Byte
Odd-Byte Only Access	L	L	H	X	L	H	VCC	VCC	Not Valid	High-Z

Table 9 Attribute Memory Read Function

4.3.6 Attribute Memory Write Function for SRAM, EEPROM Card and Single Supply FLASH Cards

While writing Attribute Memory, signals -REG and -WE/-PGM must be kept active for the entire cycle while the signal -OE is kept inactive for the entire cycle. See Table 10 for signal states and bus validity for the Attribute Memory Write function.

Function Mode	REG	CE2	CE1	A0	OE	WE	VPP2	VPP1	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	VCC	VCC	XXX	XXX
Byte Access (8bits)	L	H	L	L	H	L	VCC	VCC	XXX	Even-Byte
	L	H	L	H	H	L	VCC	VCC	XXX	XXX
Byte Access (16bits)	L	L	L	X	H	L	VCC	VCC	XXX	Even-Byte
Odd-Byte Only Access	L	L	H	X	H	L	VCC	VCC	XXX	XXX

Table 10 Attribute Memory Write Function for SRAM and Single Supply EEPROM

4.3.7 Attribute Memory Write Function for OTPROM, EPROM and FLASH Cards

The Program function for OTPROM and Dual Supply EEPROM Cards is the same as the Write function for SRAM Cards except for the functionality of VPP1 and VPP2. VPP1 and VPP2 are activated as shown in Table 11 for the write function for OTPROM and Dual Supply EEPROM Cards.

Function Mode	REG	CE2	CE1	A0	OE	WE	VPP2	VPP1	D15-D8	D7-D0
Write Inhibit	X	H	H	X	X	X	VCCorVPP	VCCorVPP	XXX	XXX
Byte Access (8bits)	L	H	L	L	H	L	VCC	VPP	XXX	Even-Byte
	L	H	L	H	H	L	VCC	VCC	XXX	XXX
Byte Access (16bits)	L	L	L	X	H	L	VPP	VPP	XXX	Even-Byte
Odd-Byte Only Access	L	L	H	X	H	L	VPP	VCC	XXX	XXX

Table 11 Attribute Memory Write Function for OTPROM, EPROM, Dual Supply EEPROM, FLASH-EEPROM and EEPROM

4.3.8 Write Protect Function

Memory Writeability Combinations on Card	Symbol	WP Switch	WP Signal	Minimum Card Information Contents Related to Write Protect
Always Writeable	A	None	Low	No WP Information Needed - Memory follows WP signal which is always Low (Not Protected). Optionally the Card info may specify all devices as Always writeable.
Never Writeable	N	None	High	No WP Information Needed - Memory follows WP signal which is always High (Protected). Optionally the Card info may specify all devices as Never writeable.
Switch Controlled	S	Protect No prot	High Low	No WP Information Needed - Memory follows WP signal.
Always/Never	AN	None	Low	Card info must specify devices (addresses) which ignore the WP signal and are Never writeable. The remaining devices follow the WP signal and are therefore Always writeable.
Always/Switch	AS	Protect	High	Card info must specify both devices (addresses) which override the WP signal and are Always writeable as well as the devices which override the WP signal and are Never writeable.
		No prot	Low	
Never/Switch	NS	Protect	High	Card info must specify devices (addresses) which ignore the WP signal and are Never writeable. The remaining devices follow the WP signal.
		No prot	Low	
Always/Never Switch	ANS	Protect	High	Card info must specify both devices (addresses) which override the WP signal and are Always writeable as well as the devices which override the WP signal and are Never writeable. The remaining devices follow the WP signal
		No prot	Low	

Table 12 Write Protect Functions

Card Interface

4.4 TIMING FUNCTIONS

4.4.0 Main Memory Timing Specification

This section describes Main Memory Access Timing.

4.4.1 Main Memory Read Timing for all types of Memory

There are several types of Memory Cards: SRAM, OTPROM, etc., and within a memory card, several types of memory devices may be mounted. To maintain compatibility among several types of memory, read timing specifications are common. The read timing specifications are shown in Table 13.

Speed Version					250ns		200ns		150ns		100ns	
Item	Symbol	IEEE Symbol	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Read Cycle Time	t _{cR}	t _{AVAV}			250		200		150		100	
Address Access Time	t _a (A)	t _{AVQV}				250		200		150		100
Card Enable Access Time	t _a (CE)	t _{ELQV}				250		200		150		100
Output Enable Access Time	t _a (OE)	t _{GLQV}				125		100		75		50
Output Disable Time from CE	t _{dis} (CE)	t _{EHQX}				100		90		75		50
Output Disable Time from OE	t _{dis} (OE)	t _{GHQZ}				100		90		75		50
Output Enable Time from CE	t _{en} (CE)	t _{ELQNZ}			5		5		5		5	
Output Enable Time from OE	t _{en} (OE)	t _{GLQNZ}			5		5		5		5	
Data Valid from Add Change	t _v (A)	t _{AXQX}			0		0		0		0	

Table 13 Main Memory Read Timing Specification for all types of Memory

4.4.2 Write Timing for SRAM Card

Write Timing Specs are shown in Table 14.

Speed Version					250ns		200ns		150ns		100ns	
Item	Symbol	IEEE Symbol	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Write Cycle Time	t _{cW}	t _{AVAV}				250		200		150		100
Write Pulse Width	t _w (WE)	t _{WLWH}			150		120		80		60	
Address Setup Time	t _{su} (A)	t _{AVWL}			30		20		20		10	
Address Setup Time for WE	t _{su} (A-WEH)	t _{AVWH}			180		140		100		70	
Card Enable Setup Time for WE	t _{su} (CE-WEH)	t _{ELWH}			180		140		100		70	
Data Setup Time for WE	t _{su} (D-WEH)	t _{DVWH}			80		60		50		40	
Data Hold Time	t _h (D)	t _{WMDX}			30		30		20		15	
Write Recover Time	t _{rec} (WE)	t _{WMAX}			30		30		20		15	
Output Disable Time from WE	t _{dis} (WE)	t _{WLQZ}				100		90		75		50
Output Disable Time from OE	t _{dis} (OE)	t _{GHQZ}				100		90		75		50
Output Enable Time from WE	t _{en} (WE)	t _{WHQNZ}			5		5		5		5	
Output Enable Time from OE	t _{en} (OE)	t _{GLQNZ}			5		5		5		5	
Output Enable Setup from OE	t _{su} (OE-WE)	t _{GHWL}			10		10		10		10	
Output Enable Hold from OE	t _H (OE-WE)	t _{WHGL}			10		10		10		10	

Table 14 Main Memory Write Timing Specification SRAM

4.4.3 Main Memory Write Timing for OTPROM, EPROM, and FLASH EPROM

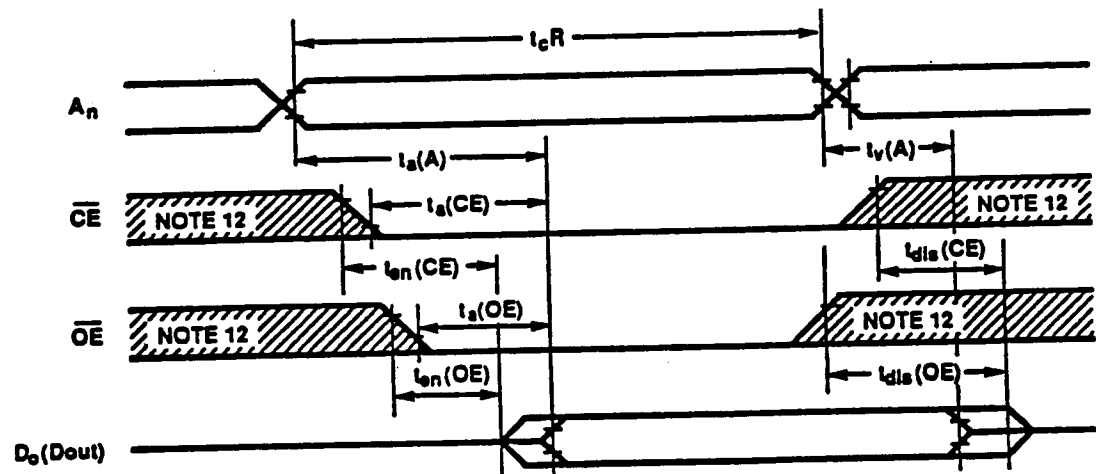
The programming specification of various memory devices are not standardized. Moreover, programming specifications may vary among different generations of the same device. Because of this situation, it is not practical to set standardized programming specifications for these memory cards.

4.4.4 Attribute Memory Read Timing Specification

The Attribute Memory's access time is defined as 300ns. Detailed timing specifications are shown in Table 15.

Speed Version			300ns	
Item	Symbol	IEEE Symbol	Min	Max
Read Cycle Time	t_{cR}	t_{AVAV}	300	
Address Access Time	$t_a(A)$	t_{AVQV}		300
Card Enable Access Time	$t_a(CE)$	t_{ELQV}		300
Output Enable Access Time	$t_a(OE)$	t_{GLQV}		150
Output Disable Time from CE	$t_{dis}(CE)$	t_{EHQZ}		100
Output Disable Time from OE	$t_{dis}(OE)$	t_{GHQZ}		100
Output Enable Time from CE	$t_{en}(CE)$	t_{ELQNZ}	5	
Output Enable Time from OE	$t_{en}(OE)$	t_{GLQNZ}	5	
Data Valid from Add Change	$t_v(A)$	t_{AXQX}	0	

Table 15 Attribute Memory Read Timing Specification for all types of Memory



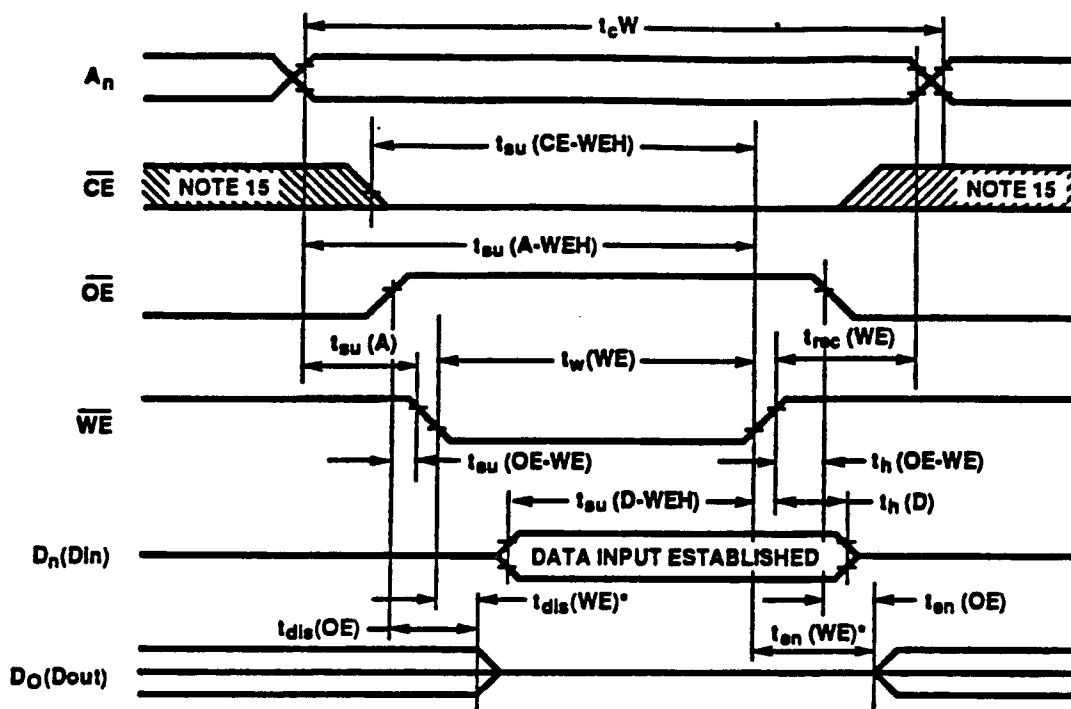
Note 12: The hatched portion may be either high or low.

Note 13: WE is high.

Note 14: Output Load = 1 TTL + 100pf

Fig 12. Read Timing Diagram

Card Interface

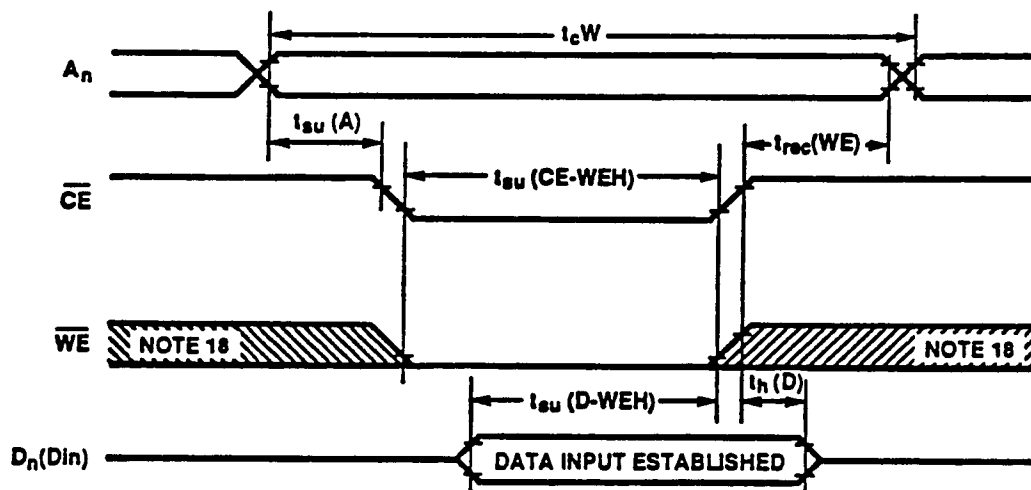


* Apply to cards for which -OE may be either [high or low] active or inactive during write operations.

Note 15: The hatched portion may be either high or low.

Note 16: When the data I/O pin is in the output state, no signals shall be applied to the data pins (D0- D15) by the system.

Fig 13. Write Timing Diagram (WE control)



Note 17: OE must be high ("H").

Note 18: The hatched portion may be either high ("H") or low ("L").

Note 19: When the data I/O pin is in the output state, a reverse phase signal should not be applied to it.

Fig 14. Write Timing Diagram (CE control)

4.5 ELECTRICAL INTERFACE

4.5.1 Signal Interface

Electrical specifications must be maintained to insure data reliability.

The Memory card operating voltage for Vcc is 4.75 to 5.25 DC. Interface signal levels are compatible with standard TTL or CMOS.

Item	Signal	Card	Host	Card Output Format
Control Signal	CE1 CE2 REG	pull-up to VCC R > 10K ohms		
	OE WE/PGM	pull-up to VCC R > 10K ohms		
	RDY/BSY RFSH	NC	pull-up NC	not defined
Address	A0-A25	pull-down R > 100K ohms*		
Data Bus	D0-D15	pull-down R > 100K ohms*		
Card Detect	CD1 CD2	connected to GND in the card	pull-up	
Reserved Pin	RFU	NC	NC	
Battery/Detect	BVD1 BVD2		pull-up NOTE 1	asserted or deasserted

Table 16 Electrical Interface

*Resistor is optional

NOTE 1: For implementation of BVD1 only type system.

4.5.2 Address Decoding

The Memory Card's maximum address space is 64M Bytes. The address bus is defined in A0-A25 with A0 being the LSB and A25 the MSB. Address bit A0 is a "don't care" when card is in word access mode.

In case of SRAM without Attribute Memory, address decoding is recommended as follows:

1. Minimum memory unit 16KB
2. Memory address starts from 00h
3. Memory units exist continuously.

Card Interface

4.6 CARD DETECT

The Memory Card provides the means to allow the system to detect when the card is inserted or removed. Signal lines CD1 and CD2 are connected to GND in the card. A pull-up resistor must be connected to CD1 and CD2 on the system side.

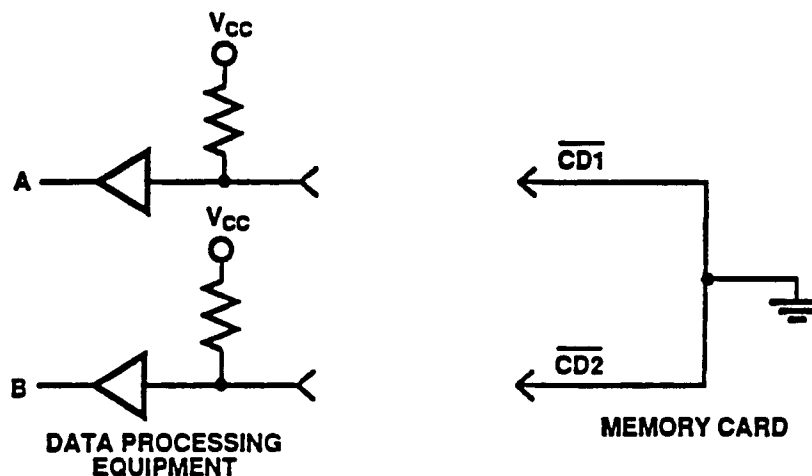


Fig. 15. Card Detect

4.7 BATTERY VOLTAGE DETECT

When using SRAM Cards, it is critical for data integrity of the system to be able to determine the status of the on-card battery. The SRAM card provides two status signals for this purpose: BVD1 and BVD2. The Memory Card contains one or two OP-AMPs and one or two reference voltages. The Memory Card compares the battery voltage with the reference voltages. Battery status is expressed on 2 digital signal lines, BVD1 and BVD2. If signal BVD2 isn't supported, BVD2 is held to V_{CC} through a pull-up resistor by the card.

BVD1 (#63)	BVD2 (#62)	COMMENT
H	H	'GREEN' Battery Operational
H	L	'YELLOW' Battery should be replaced. Data is OK.
L	H	'RED' Battery & Data integrity is not guaranteed.
L	L	'RED' Battery & Data integrity is not guaranteed.*

Table 17 Battery Voltage Detect

* If BVD2 is not supported, BVD2 is held to V_{CC} and only one reference voltage is required..

4.8 POWER-UP AND POWER-DOWN

4.8.1 Power-up/Power-down Timing

To retain data in the SRAM Card during power-up or power-down cycles, a timing specification is defined as follows.

Item	Symbol	Condition	Value		
			Min	Max	Unit
CE signal level ¹²	Vi (CE)	$0V \leq VCC < 2.0V$	0	ViMAX	V
		$2.0V \leq VCC < V_{IH}$	VCC-0.1	ViMAX	
		$V_{IH} < VCC$	V _{IH}	ViMAX	
CE Setup Time	t _{su} (VCC)		20		ms
CE Recover Time	t _{rec} (VCC)		0.001		ms
VCC Rising Time ¹³	t _{pr}	10%→90% of (VCC +5%)	0.1	300	ms
VCC Falling Time ¹³	t _{pf}	90% of (VCC -5%)→10%	3.0	300	ms

ViMAX means Absolute Maximum Voltage for Input
in the period of $0V \leq VCC < 2.0V$, Vi (CE) is only $0V-V_{iMAX}$

- ¹³ The t_{pr} and t_{pf} are defined as "linear waveform" in the period of 10% to 90% or vice-versa,
Even if the waveform is not "linear waveform", its rising and falling time must be met this specification.

Table 18 Power-up/Power-down Timing

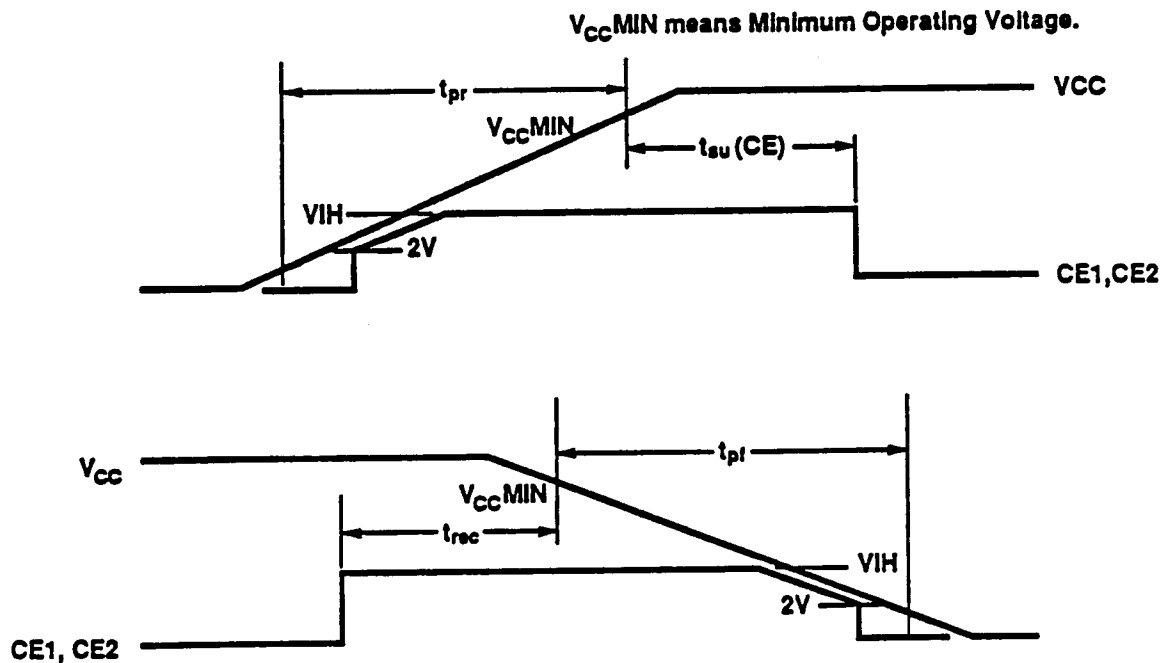


Fig. 16 Power-up/Power-down Timing

Card Interface

4.8.2 Data Retention

This specification does not intend to guarantee retention of data stored into memory cards conforming to this specification. The conditions in the preceding tables show the minimum requirements to ensure data retention. Card vendors and system vendors may have to negotiate with each other to determine the detailed method of guaranteeing data retention for specific memory card models.

4.8.3 Supplement

The data retention capability of the memory card during insertion/removal with power active depends on the individual Memory Card model, manufacturer's environmental specifications, and other conditions. Therefore, there is no guarantee of data retention during insertion/removal with power active. Appendix 2 provides technical reference information on a suggested circuit for insertion/removal with power active.

4.9 FUTURE TASKS and REMARKS

4.9.1 Insertion/Removal with Power Active

PCMCIA has recognized the market need for insertion/removal of the card with power active. PCMCIA has been discussing this issue but has not yet achieved consensus on a specification which would guarantee data retention during insertion/removal with power active. PCMCIA will continue to discuss this issue at future meetings.

4.9.2 Standardization of EPROM and EEPROM Programming

Programming specifications for EPROM and EEPROM are not standardized at the device level yet. The programming voltage, timing, and other conditions vary with individual vendors. Due to this situation, it is impossible to standardize at the memory card level. PCMCIA desires device vendors to standardize their programming voltage, timing, and other conditions. The memory card committee will continue to work towards a standard memory card which is easy to use and reliable.

4.9.3 Wide Operating Voltage

There is a large potential market of memory cards for battery powered equipment. Low voltage operation is urgently needed by this market. To respond to these needs, PCMCIA is investigating means by which low voltage memories can be supported by the standard.

4.9.4 I/O Functionality

PCMCIA is working on defining a standard I/O function. This I/O functionality will allow a variety of I/O cards, such as communications cards or disk emulation cards, to be implemented. The PCMCIA committee has set fall '90 as the target date by which it will have a draft of the I/O standard.

SECTION 5
CARD METAFORMAT

Card Metaformat

5.1 The Meta Format

5.1.1 Goals of This Standard

The following goals guided the development of this standard.

1. We want to be able to support several different file system formats on the card, both DOS compatible and other file systems (e.g., XENIX). We also want to support applications such as data storage for VCRs or musical instruments, which might not use any traditional file system to record their data. At the same time, we'd like for any computer system to be able to look someplace on a card and determine such things as the card's overall size, type and other low-level information.

Given the wide potential scope of applications for memory cards, the ability to read non-DOS cards on DOS-based systems will be of significant value to users.

The ability to detect that a given card is formatted (though perhaps not readable by the computer which the card is plugged into) is particularly valuable in that it allows system designers to protect users against common mistakes. We could prompt the user during the format routine if we detect that the card is (for example) already formatted as a data storage card for a VCR.

2. Because application requirements differ, we want to be able to support various low-level data recording strategies (akin to physical formatting for floppies). These strategies would include sequential recording of blocks of bytes with no error checking; sequential recording of blocks of bytes with embedded error checking (CRC codes); sequential recording of bytes with separate error checking (e.g., non-sequential checksum bytes); or sequential non-blocked recording of bytes.
3. For compatibility with existing operating systems and application programs, we'd like to be able to cater to those environments that believe that all media are organized in a disk-like way: with sectors, tracks and cylinders. On the other hand, we want to support those environments that simply address media as sequences of blocks.
4. We want to be able to support cards that include directly-executable ROM images, and cards that include a mixture of directly-executable images and DOS file systems.
5. We want to be able to support cards for the DOS environment that include programs that can be directly executed from ROM, or executed from RAM in the usual fashion, depending on the capabilities of the computer system.
7. The standard should be reasonably general, and should allow for future expansion without major rewrites of existing software. At the same time, for common (MS-DOS) environments, we don't want to impose excessive generality.

5.1.2 Overview

Our goals include the ability to handle numerous somewhat incompatible data-recording formats and data organizations. Taking our cue from networking standards, we have structured the overall standard as a hierarchy of layers. Each layer has a number, which increases as the level of abstraction gets higher.

The layers are:

0. The Physical Layer is the lowest layer of possible standardization. This layer specifies the form factor and electrical characteristics of memory cards.
1. The Basic Compatibility Layer specifies a minimal level of card data organization. To be compatible at this level, we merely require that each card contain a small *card information structure* ("CIS"). This structure contains certain level-1 information, primarily some fundamental information about the devices used to construct the card: size, speed, and so forth. In addition, this structure contains information on how the card is organized at levels 2, 3, and 4.

The information contained in the card information structure is commonly called the *metaformat*.

A card can comply at level 1 without being required to comply at any higher level; thus, this is an open standard. Cards that comply only at level 1 need not reserve space for the higher-level information.

The CIS can be thought of as being separate from the data recorded on the media. Under DOS, only the BIOS (or device driver) would be aware of its existence.

The information block must be recorded someplace that can be easily found by low-level software. This standard requires that the primary CIS be recorded in attribute memory, starting at address zero.¹

For flexibility, the CIS can be extended into common memory. This allows application parameters to be changed by the user, yet attribute memory can be read-only (therefore cheaper).

At this level, the standard defines two kinds of information:

- 1.1 Data structures and concepts used by all layers of this standard.
- 1.2 Physical device information.
2. The Data Recording Format Layer specifies how the data on the card is organized at the lowest level. This layer is analogous to the physical format of a floppy disk.

The use of a traditional DOS file system or boot block is NOT specified (or required) for compatibility at level 2.

¹ The metaformat standard is also applicable to non-standard card technologies that do not provide for a separate attribute memory. See Section 5.6.4, page 87.

Card Metaformat

Specific formats supported are:

- **Blocked, Unchecked** – the bytes are recorded in blocks with no error checking.
- **Blocked, Checksummed** – the bytes are recorded in blocks with checksums for error checking.
- **Blocked, with CRC** – the bytes are recorded in blocks with CRC codes for error checking.
- **Unblocked** – individual bytes of the card may be accessed or modified by software directly at random. The bytes are recorded in a way that does not correspond to a disk organization. The FlaSh file system uses cards in an unblocked way.

3. **The Data Organization Layer** specifies how the data is logically organized on the card. Possibilities are:

- **DOS** (or other operating system) file system.
- **Flash file system.**
- **Execute-in-place ROM image.**
- **Application-specific organization.**

A DOS file system can be used with any of the appropriate (blocked) level 2 organizations.

4. **The System-Specific Layer** defines standards that by their nature are specific to a particular operating environment.

4.1 **The DOS Direct-Execution Standard** defines a standardized way of preparing DOS-executable images on ROM cards. Programs that conform to this standard will execute correctly on any system that can read the ROM card; in addition, the programs will directly execute from the ROM card on those systems that support direct execution.

5.1.3 Vendor-Specific Information

Vendor-specific information allows card and software vendors to implement proprietary functions while remaining within the general framework of this standard.

Vendor-specific information comes in two kinds:

- Vendor-specific fields are areas reserved in the data structures for free use by vendors. These fields have no meaning to the standard software.
- Vendor-specific codes are encoding values reserved to represent non-standard values in standard fields. In the absence of other information, standard software must interpret vendor-specific codes as meaning "the information in this field is not specified."

The card-manufacturer field in the CIS gives knowledgeable system software enough information to interpret vendor-specific fields and code values in the card physical-description tuples.

Similarly, the OEM and INFO fields in the CISTPL_VERS_2 tuple give knowledgeable system software enough information to interpret vendor-specific fields and code values in the card logical format tuples.

A system will not, in general, be able to interpret all possible vendor-specific fields or code values. This standard requires the following behavior when a system encounters an unrecognized vendor-specific field.

- If the unrecognized field itself is vendor-specific, the system shall ignore that field.
- If a standard field contains an unrecognized vendor-specific code, the system must refuse to perform any operation that requires the information encoded in that field.

This page intentionally left blank.

5.2 Basic Compatibility (Layer 1)

This layer is the cornerstone of the standard. Any card that complies with this standard shall have at least a rudimentary card information structure (referred to as the "CIS") recorded starting at address zero of the card's attribute memory space.

The card information structure is a variable-length linked list of data blocks known as tuples. All tuples have the format shown in table 19.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Tuple code: CISTPL_XXX; see table 2.							
1	TPL_LINK Offset to next tuple in list. This can be viewed as the number of additional bytes in tuple, excluding this byte. (n)							
2..n	Bytes specific to this tuple.							

Table 19 Tuple Format

Byte 0 of each tuple contains a tuple code. A tuple code of FFh is a special mark, indicating that there are no more tuples in the list. Byte 1 of each tuple contains a link to the next tuple in the list. If the link field is zero, then the tuple body is empty. If the link field contains FFh, then this tuple is the last tuple in the list.

There are thus two ways of marking the end of the tuple list: a tuple code of FFh, or a tuple link of FFh.

- If a tuple code of FFh marks end-of-list, and the list is stored in writeable memory, it will be easy to add additional items to the list.
- If the list is stored in read-only memory, a tuple link of FFh conserves memory space.

System software must use the link field to validate tuples. No tuple can be longer than (2 + value of link field) bytes long. Some tuples provide a termination or stop byte that marks the end of the tuple. In this case, the tuple can effectively be shorter than the value implied by its link field. However, software must not scan beyond the implied length of the tuple, even if a termination byte has not been seen.

Card Metaformat

The following tuple codes are defined:

Code	Name	Description
0	CISTPL_NULL	Null tuple - ignore.
1	CISTPL_DEVICE	The device information tuple (common memory).
2-7		(Reserved for future, upwards compatible versions of the device information tuple)
8-0Fh		(Reserved for future, incompatible versions of the device information tuple.)
10h	CISTPL_CHECKSUM	The checksum control tuple.
11h	CISTPL_LONGLINK_A	The long-link control tuple (to attribute memory).
12h	CISTPL_LONGLINK_C	The long-link control tuple (to common memory).
13h	CISTPL_LINKTARGET	The link-target control tuple
14h	CISTPL_NO_LINK	The no-link control tuple
15h	CISTPL_VERS_1	Level 1 version / product-information tuple.
16h	CISTPL_ALTSTR	The alternate-language string tuple.
17h	CISTPL_DEVICE_A	Attribute memory device information.
18h	CISTPL_JEDEC_C	JEDEC programming information for common memory
19h	CISTPL_JEDEC_A	JEDEC programming information for attribute memory
1Ah-3Fh		(Reserved for future standardization.)
40h	CISTPL_VERS_2	The level-2 version tuple.
41h	CISTPL_FORMAT	The format tuple
42h	CISTPL_GEOMETRY	The geometry tuple. Only allowed for disk-like formats.
43h	CISTPL_BYTEORDER	The byte-order tuple. Only allowed for memory-like formats.
44h	CISTPL_DATE	The card initialization date and time tuple.
45h	CISTPL_BATTERY	The card battery-replacement date and time.
46h	CISTPL_ORG	The data organization tuple.
47h-7Fh		(reserved for future standardization)
80h-FEh		Vendor-specific.
FFh	CISTPL_END	The end-of-list tuple

Table 20 Tuple Codes

Note to Implementors: It is anticipated that the CIS will be written once, when the card is formatted, and then rarely (if ever) updated. The standard is not designed to allow incremental updating of the CIS on Flash media. On EEPROM devices that require the CIS to be erased occasionally (for example when a Flash-type file system is reorganized), we suggest that a buffer-page strategy be used, with an appropriate utility that can recover from a power-failure.

Note to Implementors: Most implementations will be limited to reading cards of a specific format, or at most of a few different formats. Thus, many combinations of values available in the tuples will be non-portable. We suggest that implementors restrict themselves to the suggested formats presented in section 5.3.3.

5.2.1 Byte Order Within Tuples

Within tuples, all multi-byte numeric data shall be recorded in little-endian order; that is, the least-significant byte of a data item shall be stored in the first byte of a given field.

Within tuples, all character data shall be stored in the natural order; that is, the first character of the field shall be stored in the first byte of the field. Fixed length character fields shall be padded with null characters, if necessary.

5.2.2 Byte Order on Wide Cards

If a card has a data-path wider than 8 bits, we must assign a byte order to the data path, at least for fields within the CIS that are recorded in common memory space.² This standard requires that the low-order byte of word 0 be used to record byte 0 of the CIS. Ascending bytes of each word shall be used to record bytes sequentially from the CIS; when the first word is filled, the same process shall be repeated on subsequent words until the entire CIS is recorded. On Intel-family machines, this byte order is equivalent to the native order; other machines may need to reorder the bytes when reading or writing the CIS.

The basic compatibility layer does not impose any particular byte order on non-header portions of the card. However, some data-format layers will impose further requirements.

5.2.3 Tuple Format in Attribute Memory Space

PC Cards have two address spaces: attribute memory space and common memory space. The electrical specification for PC Cards requires that information be placed only in *even* byte addresses of attribute memory space; the contents of *odd* byte addresses of attribute memory space are not defined.

For simplicity, this specification describes the tuples of the metaformat as if the bytes of each tuple were recorded consecutively. When a tuple is recorded in common memory space, the bytes will indeed be recorded consecutively; but when a tuple is recorded in attribute memory space, the data will be recorded in even bytes only.

Link fields of tuples stored in attribute memory space are handled as follows. If only the even bytes are read and the tuples are copied into system memory, packed into consecutive bytes, the link fields shall be set appropriately for byte addressing. This means that the link field values are conceptually the same, whether a tuple resides in common memory or in attribute memory.

² At present, attribute memory is byte-wide only; only the even bytes are present.

Card Metaformat

5.2.4 Use of Common Memory Space for Attribute Memory Storage

For cost reasons, many ROM cards will not implement a separate attribute memory space. Regardless of the state of the /REG line, memory cycles will always access common memory. These cards will provide an attribute-memory-style CIS starting at byte zero of the card, and recorded in even bytes only. If, for space reasons, the manufacturer wants to switch to a common-memory-style CIS (packed into ascending bytes), a long-link to common memory shall be embedded in the CIS. The target address of this long-link must be non-zero, and the common-memory CIS will be stored immediately following the attribute-memory CIS.

It's important to distinguish between *attribute memory space* and *attribute memory*. All PC Cards will have attribute memory space, accessed by asserting the /REG pin. Some PC Cards will, in addition, have attribute memory; in this case, the contents of location 0 in attribute memory space will be different and distinct from the contents of location 0 in common memory space. However, many PC Cards (e.g., ROM cards) will not have attribute memory distinct from common memory; in this case, reads from a given location in attribute memory space will return the same data as reads from the same location in common memory space. Data being accessed from attribute memory space must be stored in the even bytes only, even if attribute memory is not distinct from common memory. Regardless of the presence or absence of attribute memory, the CIS for PC Cards always begins at location 0 of attribute memory space.

This standard allows attribute information to be stored both in attribute memory *space* and common memory space. Tuples stored in common memory space are recorded byte sequentially; both the even and the odd bytes of the card are used to record data.

Note that the use of odd bytes to represent tuple data is controlled by the logical address space the tuple resides in, not by the type of memory actually used to record the tuple. If the tuple is intended to be accessed via attribute memory space, it must be stored only in the even bytes; if it's intended to be accessed via common memory space, it must be stored in even and odd bytes.

5.2.5 Control Tuples

5.2.5.1 The Null Control Tuple

The null control tuple is simply a placeholder. It has a non-standard form: it consists solely of the code byte.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE CISTPL_NULL (00h): ignore this tuple.							

Table 21 The Null Control Tuple

Software shall ignore these tuples. The next tuple begins at the next byte in sequence.

5.2.5.2 The Long-Link Control Tuples

The long-link tuples are used to jump from one tuple chain to another, beyond the limits of the 1-byte link field. The target tuple chain may be in attribute memory space or common memory space, as indicated by the tuple code.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Long-link tuple code (CISTPL_LONGLINK_A, 11h; or CISTPL_LONGLINK_C, 12h)							
1	TPL_LINK Link to next tuple (at least 4).							
2..5	TPL_ADDR target address; stored as an unsigned long, low-order byte first.							

Table 22 Long Link Tuple

The tuple code byte selects the new address space: CISTPL_LONGLINK_A indicates that the target is in attribute memory space; CISTPL_LONGLINK_C indicates common memory space.

A given tuple chain shall contain at most one long-link tuple. The long-link tuple need not appear as the last tuple in a given chain; the entire chain containing the long-link tuple will be processed before the link is honored.

Software shall verify that the Long-Link tuple points to a Link-Target tuple before processing the target chain. Because a Long-Link tuple may point to uninitialized RAM, it's important that software simply reject target tuple chains that don't begin with a Link-Target tuple.

Card Metaformat

5.2.5.3 The Link-Target Control Tuple

The link-target tuple is used for robustness. Every long-link tuple must point to a valid link-target tuple. The link target tuple has one principal field: the string "CIS". The link field of the link-target should always point to the next byte after the link-target tuple. Processing software is required to check that the link-target tuple is correct before deciding to process the linked list of tuples at the new target address.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE CISTPL_LINKTARGET (13h)							
1	TPL_LINK Link to next tuple (at least 3).							
2	TPLTG_TAG "C" (43h)							
3	"I" (48h)							
4	"S" (53h)							

Table 23 Link Target Tuple

5.2.5.4 The No-Link Control Tuple

The attribute-memory CIS of a RAM Card must be kept small for economic reasons. To save attribute memory space, processing software shall assume the presence of a (CISTPL_LONGLINK_C,0L) tuple as part of the primary tuple chain - the tuple chain which starts at address 0 of attribute memory space. This assumption can be overridden by placing an explicit long-link tuple in the attribute-memory CIS. To prevent software from trying to execute any long-link operations, the card manufacturer can place a (NO-LINK) tuple in the attribute-memory CIS.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE CISTPL_NO_LINK (14h)							
1	TPL_LINK Link to next tuple (may be zero).							

Table 24 No-Link Tuple

Note that the body of this tuple is always empty.

A given tuple chain shall contain at most one NO-LINK control tuple. No-link tuples and long-link tuples are mutually exclusive: a given chain may contain either a no-link tuple or a long-link tuple but not both.

5.2.5.5 The End-Of-List Tuple

The End-of-list control tuple marks the end of a tuple chain. It has a non-standard form, consisting solely of the code byte.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE CISTPL_END (FFh): end of this tuple chain.							

Table 25 The End-of-List Control Tuple

Upon encountering this tuple, system software shall take one of the following actions:

- If a long-link tuple was encountered previously in this chain, continue tuple processing at the location specified in the long-link tuple.
- If a no-link tuple was encountered previously in this chain, no tuples remain to be processed.
- If processing the primary CIS tuple chain (the list starting at address 0 in attribute memory space), and *neither* a long-link nor a no-link tuple were seen in this chain, then continue tuple processing as if a long-link to address 0 of common memory space was encountered.
- If processing and tuple chain other than the primary CIS tuple chain, and no long-link tuple was seen in this chain, then no tuples remain to be processed.

5.2.5.6 The Checksum Control Tuple

For additional reliability, the CIS can contain one or more checksum tuples. This tuple has three fields: the relative address of the block of CIS memory to be checked; the length of the block of CIS memory to be checked; and the expected checksum. The checksum algorithm is a straight modulo-256 sum. Relative addressing is used to make the CIS as a whole position-independent. The checksum tuple can only validate memory in its own address space.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE CISTPL_CHECKSUM (10h)							
1	TPL_LINK Link to next tuple (at least 5).							
2..3	TPLCKS_ADDR offset to region to be checksummed, stored LSB first.							
4..5	TPLCKS_LEN length of region to be checksummed, given LSB first.							
6	TPLCKS_CS the checksum of the region.							

Table 26 Checksum Tuple

The checksum is calculated by summing the bytes of the selected region, modulo 256. The result must match the value stored in byte 6 of the Checksum Tuple.

TPLCKS_ADDR contains the offset of the region to be checksummed, relative to the start address of this tuple. The address is a signed, 2 byte integer. Negative values indicate locations prior to the checksum tuple; positive values indicate locations after the checksum tuple. The exact interpretation depends on the address space containing the tuple.

TPLCKS_LEN contains the number of bytes to be checksummed, expressed as an unsigned, 2 byte integer.

If the tuple appears in common memory space, the checksum is calculated in the obvious way: simply add the contents of TPLCKS_ADDR (as a signed integer) to the base address of the tuple, yielding the target address. Starting at the target address, form the algebraic sum of all the bytes included in the range. Then compare the low-order 8 bits of this sum to the value stored in TPLCKS_CS. If identical, then the region of tuple memory covered by the checksum passes the checksum test.

If the tuple appears in attribute memory space, then things are a bit more complicated. Again, we choose to record the data structures in such a way as to minimize the differences between attribute space representation and common memory representation. To form the target address, add $(2 * \text{offset})$ to the base byte target address of the tuple. Then form the algebraic sum of the even bytes in the address range $[\text{target}, \text{target} + 2 * \text{length} - 1]$, ignoring the odd bytes. Compare the low-order 8 bits of this sum to the value stored in TPLCKS_CS.

5.2.5.7 The National-Language String Tuple

Several tuples contain character strings, which are intended to be displayed to the user under some circumstances. Some international applications need the ability to store strings for a number of different languages. Rather than having various languages used in the tuples, this standard provides Alternate-String tuples. Strings in the primary tuples are always recorded in ISO 646 IRV code, using characters in the range [20h..7Eh]. Alternate-String tuples contain two kinds of information: a code representing the language (an ISO-standard escape sequence), and a series of strings. These strings are to be positionally substituted for the primary strings when operating in that language environment.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE CISTPL_ALTSTR (16h)							
1	TPL_LINK Link to next tuple (at least p-1).							
2..m-1	TPLALTSTR_ESC ISO-standard escape sequence to select the character set for these strings. Indicates which character set is associated with these strings. The leading ESCAPE is not recorded. Terminated by a NUL (00h). A special escape sequence denotes the PC Extended-ASCII character set; see appendix							
m..n-1	Alternate string 1; translation for first string in most recent non-ALTSTR tuple. Terminated by 00h.							
n..o-1	Alternate string 2; translation for second string in most recent non-ALTSTR tuple. Terminated by 00h.							
...	Etc.							
p	FFh - marks end of strings.							

Table 27 Alternate Language String Tuple

5.2.5.8 Tuple Processing Recommendations

This standard requires system software to be carefully coded in order to prevent incompatibilities from system to system. This section presents some specific recommendations.

- The routine that reads a given tuple should be coded to start by examining the tuple code. If the tuple code is not recognized by the routine (e.g., if the code is vendor specific or represents an extension under a future standard), then the tuple should be ignored. If the code is not recognized, it is safe to read the code byte and the link byte: other bytes within the tuple may represent active registers.

Card Metaformat

- If the tuple code is known, and if the tuple does not contain active registers (which is the case for all standard tuples), then the routine should copy bytes into a buffer in main storage. Bytes should be copied from the code byte up to the last byte before the next tuple. If the link field is FFh (meaning end-of-list) then a maximum of 257 bytes should be copied from the card to the main store: the code byte, the link byte and 255 byte of potential tuple data.
- When processing a long-link tuple, software should merely record the target address and address space; it should not validate the target address, nor should it immediately begin processing of tuples from the target address. Similarly, when a no-link tuple is found, that fact should be recorded for later.

Long-link and no-link tuples should be processed when the end of the tuple chain is encountered. At that time, if a long-link is to be processed, software should validate the target address (by checking for a link-target tuple) and begin processing the target chain if it appears to be valid.

- A long-link that points to an invalid tuple chain should not usually cause any diagnostic messages to be displayed to the user. This situation may result from an uninitialized card, from a card which was initialized for some unanticipated use, or from data being corrupted. Since only the last mentioned case merits a diagnostic message, it is better to assume either that the card is uninitialized or that it is initialized in some unconfirming way.

5.2.6 Basic Compatibility Tuples

5.2.6.1 The Device Information Tuple

The device information tuples contain information about the devices on the card. The tuples contain device speed, device size, device type, and address space layout information for either attribute memory space or common memory space, as determined by the tuple code. A device information tuple for common memory space (CISTPL_DEVICE, 01h) must be the first tuple in attribute memory. The device-information tuple for attribute memory is optional.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE CISTPL_DEVICE (01h) or CISTPL_DEVICE_A (17h)							
1	TPL_LINK Link to next tuple (at least m-1).							
	Device Info 1 (2 or more bytes)							
	Device Info 2 (2 or more bytes)							
...	(etc.)							
	Device Info n (2 or more bytes)							
m	FFh (marks end of device info field).							

Table 28 Device Information Tuples

The tuple code CISTPL_DEVICE indicates that this tuple describes common memory space; the code CISTPL_DEVICE_A indicates that this tuple describes register memory space.

5.2.6.1.1 The Device Info Field

The device-information tuples are composed of a sequence of device info fields. Each info field is further composed of two variable-length sequences of bytes: the device ID and the device size. Each info field defines the characteristics of a group of addresses in the appropriate memory space.

Card Metaformat

5.2.6.1.2 Device ID

The device ID indicates the device type and the access time for a block of memory.

Byte	7	6	5	4	3	2	1	0
0	Device Type Code				WPS	Device Speed		
1	Extended Device Speed (if Device Speed Code equals Eh, otherwise omitted)							
2	Additional Extended Device Speed (if bit 7 of Extended Device Speed is 1, otherwise omitted)							
3	Extended Device Type (if Device Type Code equals Eh, otherwise omitted).							

Table 29 Device ID

The WPS bit, if clear, indicates that the write protect switch is in control of this device.

5.2.6.1.3 Device ID Speed Field

If the device speed/type byte is 00h, the effect is unspecified. If the device size information is valid, the address range shall be treated as a NULL device.

If the device speed/type byte is FFh, it shall be treated as an end marker.

Bits 0 through 3, and one or two optional additional bytes, represent the speed of the devices associated with this part of the address space. The device speed field contains one of the following values:

Code	Name	Meaning
0h		(Reserved - do not use)
1h	DSPEED_250NS	250 nsec
2h	DSPEED_200NS	200 nsec
3h	DSPEED_150NS	150 nsec
4h	DSPEED_100NS	100 nsec
5h-6h		(Reserved)
7h	DSPEED_EXT	use extended speed byte.

Table 30 Device Speed Codes

The extended speed byte has the following layout:

7	6	5	4	3	2	1	0
EXT	Speed Mantissa				Speed Exponent		

If the extended speed byte is zero, then the byte should be ignored.

The EXT bit, if set, indicates that an additional extended speed byte follows. The meaning of that byte is not presently defined. However, the string of extended speed bytes may be arbitrarily long: it extends through (and including) the first byte that has bit 7 reset.

The extended device Speed Mantissa and Exponent specify the speed of the device, as follows:

Mantissa:		Exponent Part:	
0h	Reserved	0h	1 nsec
1h	1.0	1h	10 nsec
2h	1.2	2h	100 nsec
3h	1.3	3h	1 microsec
4h	1.5	4h	10 μ sec
5h	2.0	5h	100 μ sec
6h	2.5	6h	1 msec
7h	3.0	7h	10 msec
8h	3.5		
9h	4.0		
Ah	4.5		
Bh	5.0		
Ch	5.5		
Dh	6.0		
Eh	7.0		
Fh	8.0		

Table 31 Extended Device Speed Codes

5.2.6.1.4 Device ID Type Field

Bits 4 through 7 of byte 0 of the device speed/id sequence indicate the device type. The following device types are defined:

Code	Name	Meaning
0	DTYPE_NULL	No device. Generally used to designate a hole in the address space. If used, speed field should be set to Fh.
1	DTYPE_ROM	Masked ROM
2	DTYPE_OTPROM	One-time programmable PROM
3	DTYPE_EPROM	UV EPROM.
4	DTYPE_EEPROM	EEPROM
5	DTYPE_FLASH	Flash EPROM
6	DTYPE_SRAM	Static RAM
7	DTYPE_DRAM	Dynamic RAM.
8-Ch		(reserved for future use)
Dh	DTYPE_IO	I/O Device
Eh	DTYPE_EXTEND	Extended type follows.
Fh		(reserved for future use)

Table 32 Device Type Codes

The extended device type, if specified, is reserved for future use. Bit 7, if set, indicates that the next byte is also an extended type byte. The chain of extended type bytes can continue indefinitely. The end is marked by an extended device type byte with bit 7 reset.

Card Metaformat

5.2.6.1.5 The Device Size Byte

Following the device speed/type information within a Device ID structure is a device size byte.

7	6	5	4	3	2	1	0
# of address units - 1					Size Code		

If the device-size byte is zero, then it is not valid; in this case, this device ID block should be ignored. Otherwise, bits 0 through 2 indicate the address units used to describe this part of the address space:

Code	Units	Max Size
0	512 bytes	16K
1	2K	64K
2	8K	256K
3	32K	1M
4	128K	4M
5	512K	16M
6	2M	64M
7	8M	256M

Bits 3 through 7 represent the number of address units: a code of zero indicates 1 unit; a code of 1 indicates 2 units; and so forth.

If the device-size byte is FFh, then this entry should be treated as an end marker for the device information tuple. The device type and speed information encoded for this entry should be ignored.

5.2.6.2 The Level 1 Version / Product Information Tuple

This tuple contains level-1 version compliance and card-manufacturer information.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE CISTPL_VERS_1 (15h).							
1	TPL_LINK Link to next tuple (at least m-1).							
2	TPLLV1_MAJOR major version number (04h)							
3	TPLLV1_MINOR minor version number (00h)							
4	TPLLV1_INFO Product information string: name of the manufacturer, terminated by 00h. Name of product, terminated by 00h. Additional product information, in text; terminated by 00h. Suggested use: lot number. Additional product information, in text; terminated by 00h. Suggested use: programming conditions.							
m	FFh: marks end of list.							

Table 33 Level 1 Version / Product Information Tuple

5.2.6.3 The JEDEC Identifier Tuples

This optional tuple is provided for cards containing programmable devices. It provides an array of k entries, where k is the number of distinct entries in the device information tuple (codes 01h or 17h). There's a one to one correspondence between JEDEC identifier entries in this tuple and device information entries in the device-information tuple.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Format tuple code (CISTPL_JEDEC_C,18h, or CISTBL_JEDEC_A,19h).							
1	TPL_LINK Link to next tuple (at least m-1).							
2..3	JEDEC identifier for first device-info entry.							
4..5	JEDEC identifier for second device-info entry (if needed).							
6..m	JEDEC identifiers for remaining device-info entries (if needed).							

Table 34 The JEDEC Identifier Tuples

The TPL_CODE field indicates which device information tuple the JEDEC identifier tuple corresponds to. If the value is 18h (CISTPL_JEDEC_C), this tuple corresponds to the common-memory device information tuple (CISTPL_DEVICE). If the value is 19h (CISTPL_JEDEC_A), this tuple corresponds to the attribute-memory device information tuple (CISTPL_DEVICE_A).

JEDEC identifiers consist of two bytes. The first byte is the device manufacturer ID, as assigned by JEDEC committee JC-42.3. This byte if valid must always have an odd number of bits set true. The MBS (bit 7) of the byte is used as a parity bit to ensure that this constraint is met. The values of 0 and FFh are illegal JEDEC identifiers; such values are used to indicate "no JEDEC identifier for this device" and "end of JEDEC identifier list" respectively.

The second byte contains manufacturer-specific information, representing device type, programming algorithm, and so forth. If the manufacturer ID is 00h, then this byte is reserved and should be set to zero.

It's intended that JEDEC identifiers will only be provided for device-info entries that indicate some kind of programmable device. For all other entries, the corresponding JEDEC identifier field shall be absent (i.e., after the FFh byte that marks the end of the list) or set to 00h.

Examples:

If a card consists of four FooBar 27C512 devices, whose JEDEC identifier is (hypothetically) (manufacturer: 40h, device ID:15h), then the header block might be laid out as follows:

```

/reg[0]:      (CISTPL_DEVICE,
/reg[2]:      /*link*/
/reg[4]:      /*type/speed*/
/reg[6]:      /*size: units/exp*/
              );
              2,
              DTYPE_OTPROM | DSPEED_250NS,
              ((1<<3) | 4)

/reg[8]:      (CISTPL_JEDEC_C,
/reg[10]:     /*link*/
/reg[12]:     /*manufacturer ID*/
              /*18h*/
              0FFh, /*(end of list)*/
              40h,
```

Card Metaformat

```
/reg[14]:      /*mfr's info*/      15h,  
/reg[16]:      /*end of tuple*/    FFh  
);
```

(In the above example, the size byte indicates that 2 x 128K bytes are available, for a total of 256K. Programming software would use the JEDEC information to determine that in fact 4 64K x 8 devices are present; since JEIDA V4 / PCMCIA cards are always 16 bits wide, programming software could therefore deduce the the organization of the card).

The following tuples might be used to describe a card with 256K of OTPROM and 16 Kbytes RAM. In this example, RAM occupies locations [0..03FFFh], and ROM occupies locations [20000h..5FFFFh] (for ease of decoding).

```
/reg[0]:      (CISTPL_DEVICE,  
/reg[2]:      /*link*/              6,  
/reg[4]:      /*type/speed*/        DTYPE_SRAM | DSPEED_100NS,  
/reg[6]:      /*size: units/exp*/    ((1<<3) | 2)  
/reg[8]:      /*type/speed*/        DTYPE_NULL | DSPEED_NONE,  
/reg[10]:     /*size: units/exp*/    ((1<<3) | 2)  
/reg[12]:     /*type/speed*/        DTYPE_OTPROM | DSPEED_250NS,  
/reg[14]:     /*size: units/exp*/    ((1<<3) | 4)  
);  
  
/reg[16]:     (CISTPL_JEDEC_C,  
/reg[18]:     /*link*/              0FFh, /*(end of list)*/  
/reg[20]:     /*RAM: no code*/       0,  
/reg[22]:     /*no info*/            0,  
/reg[24]:     /*hole: no code*/       0,  
/reg[26]:     /*no info*/            0,  
/reg[28]:     /*manufacturer ID*/    40h,  
/reg[30]:     /*mfr's info*/         15h,  
/reg[32]:     /*end of tuple*/       FFh  
);
```

Note that place holders were left in the CISTPL_JEDEC tuple corresponding to the RAM and hole entries in the device tuple.

5.2.7 Use of Additional Tuples

Cards that only comply with the basic compatibility layer of this standard need only contain the basic device information tuple for the common memory space. Cards that comply with one of the data format standards listed in section 5.3.3 will need to supply additional information as specified in that section. In many cases, an implementation will need only to verify that certain subfields in specific tuples are compatible with its requirements.

5.3 Data Recording Formats (Layer 2)

This level defines the data-recording format for the card. If none of the layer-2 headers are present, software should assume that the card is organized as an unchecked sequence of bytes.

Card data-recording formats fall into two categories:

- *Disk-like*: the card consists of a number of blocks of data, where each block consists of a fixed number of bytes. These blocks correspond to the sectors of rotating disk drives. Conceptually, an entire block must be updated if any byte in the block is to be changed.
- *Memory-like*: the card is treated as a sequence of directly-addressable bytes of data.

Formats are further categorized according to how error-checking is performed. This standard recognizes 3 basic possibilities:

- *unchecked*: no data checking is performed at the data format layer.
- *checked with in-line codes*: data checking is performed by the data format layer using check codes. The check code for a given block is recorded immediately after the block.
- *checked with out-of-line codes*: data checking is performed by the format layer using check codes. The check code for a given block is recorded in a special table that resides separately from the data blocks.
- *checked over entire partition*: data checking is performed only over the complete partition.

This standard recognizes two kinds of check codes: arithmetic checksum and CRC. Arithmetic checksums are typically one or two bytes long; CRCs are always two bytes long.

When cards with 16-bit or wider data paths are used to record byte data, it's necessary to specify how the bytes of the data card correspond to sequential bytes of data. In this standard, all disk-like organizations require that bytes be assigned to words with the lowest byte address mapping to the least-significant byte of the word, and subsequent byte addresses mapping to increasingly significant bytes.

Memory-like formats also require that the byte mapping be specified. For maximum flexibility, both little-endian and big-endian byte orders are supported.

Card Metaformat

5.3.1 Card Information Tuples

The tuples listed in the following subsections provide generic information about how the card is intended to be used.

5.3.1.1 The Level-2 Version and Information Tuple

The level 2 information tuple serves to introduce information pertaining to the logical organization of the data on the card. The layout of the level-2 tuple is shown in table 35.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Code value indicating that this is the level-2 tuple (CISTPL_VERS_2, 40h).							
1	TPL_LINK Link to next tuple (at least m-1).							
2	TPLL2_VERS structure version (00h).							
3	TPLL2_COMPLY level of compliance claimed.							
4..5	TPLL2_DINDEX byte address of first data byte in card (LSB first).							
6..7	TPLL2_RSV6, TPLL2_RSV7 reserved; must be zero.							
8..9	TPLL2_VSPEC8, TPLL2_VSPEC9 vendor-specific bytes.							
10	TPLL2_NHDR Number of copies of CIS present on the device.							
11..k	TPLL2_OEM Vendor of software that formatted card (ASCII, variable length, terminated with a NUL (00h)).							
k+1..m	TPLL2_INFO Informational message about the card (ASCII, variable length, terminated with a NUL (00h)).							

Table 35 Level-2 Information Tuple

TPLL2_VERS represents the standardization version of the tuple. This byte should always be zero.

TPLL2_COMPLY indicates the claimed degree of compliance with this standard. At present, this should always be zero.

TPLL2_DINDEX specifies the address of the first data byte on the card. Setting this non-zero reserves bytes at the beginning of common memory. Note that the first data byte on the card must always be somewhere in the first 64 Kbytes of the card. This field should be consistent with information provided in the format tuple (if that tuple is present).

TPLL2_NHDR specifies the number of copies of the CIS that are present on the card. For compatibility with this standard, this value should be 1. This field will allow automated recovery in the face of various error conditions.

TPLL2_OEM specifies the vendor of the machine or format program that formatted the card. This is a text string, terminated by a NUL byte (00h). The value of TPLL2_OEM, combined with the value of TPLL2_INFO, determines how vendor-specific fields in the level-2 tuples are to be interpreted.³ For alternate languages, CISTPL_ALTSTR tuples may follow this tuple, specifying the string value to be substituted when using alternate languages.

TPLL2_INFO contains a text message, terminated by a NUL byte (00h). This message is intended to be displayed to users by a computer whenever the host needs to describe the type of card that's in the drive. For alternate languages, CISTPL_ALTSTR tuples may follow this tuple, specifying the string value to be substituted when using alternate languages.

Note to Implementors: if the computer system's format routine determines that the card is already formatted, it will display a message like:

Caution! This card contains data for <info>, from <vendor>.

The contents of the information field should be chosen appropriately. For example, a VCR setup card for a VCR by Shrdlu Electronics might have <info> as "Model 9770 VCR"; the <vendor> field would be "Shrdlu".

The characters used in TPLL2_INFO and TPLL2_OEM shall be chosen from the printing 7-bit ISO 646 IRV set (codes 20h through 7Eh, inclusive).

TPLL2_RSV6 and TPLL2_RSV7 are reserved for use in future versions of this standard. They shall be set to zero.

TPLL2_VSPEC8 and TPLL2_VSPEC9 are vendor specific. If not used, they shall be set to zero.

³ The PCMCIA will maintain a registry of vendor names.

Card Metaformat

5.3.1.2 The Card Initialization Date Tuple (CISTPL_DATE)

This optional tuple indicates the date and time at which the card was formatted. Its format is given in table 36.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Initialization-date tuple code (CISTPL_DATE, 44h).							
1	TPL_LINK Link to next tuple (at least 4).							
2	TPLDATE_TIME: MMM _{ls}			SSS				
3	HHH					MMM _{hi}		
4	TPLDATE_DAY: MON _{ls}			DAY				
5	YYYY							MON _{hi}

Table 36 Card Initialization Date Tuple

Bytes 2-3 (TPLDATE_TIME) indicate the time at which the card was initialized; it should be considered to be a 16-bit number, stored LSB first.

- The field HHH contains the hour at which the card was initialized; it is a number between 0 and 23.
- The field MMM contains the minute at which the card was initialized; it is a number between 0 and 59.
- The field SSS represents the two-second interval at which the card was initialized; it is a binary number between 0 and 29. To convert SSS to seconds, it should be multiplied by two.

Bytes 4-5 (TPLDATE_DAY) indicate the date the card was initialized; it should be considered to be a 16-bit number, stored LSB first.

- The field YYYY represents the year; it is a binary number between 0 and 127, with 0 representing the year 1980.
- The field MON represents the month; it is a binary number between 1 and 12, with 1 representing January.
- The field DAY represents the day; it is a binary number between 1 and 31.

If the date and time components of the date are both zero, this should be taken as an indication that the date and time were unknown when the card was first initialized.

5.3.1.3 The Battery-Replacement Date Tuple (CISTPL_BATTERY)

This optional tuple shall be present only cards with battery-backed storage. It indicates the date at which the battery was replaced, and the date at which the battery is expected to need replacement. Its format is given in table 37.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Initialization-date tuple code (CISTPL_BATTERY, 45h).							
1	TPL_LINK Link to next tuple (at least 4)							
2	TPLBATT_RDAY: MON ₁₆			DAY				
3	YYYY							MON _N
4	TPLBATT_XDAY: MON ₁₆			DAY				
5	YYYY							MON _N

Table 37 Battery Replacement Date Tuple

Bytes 2-3 (TPLBATT_RDAY) indicate the date on which the battery was last replaced; it should be considered to be a 16-bit number, stored LSB first. This field has the same interpretation as the field TPLDATE_DAY (page 68).

Bytes 4-5 (TPLBATT_XDAY, "expiration day") indicate the date on which the battery should be replaced. This field has the same format as TPLBATT_RDAY.

If either field is zero, it indicates that the corresponding date was not known when the tuple was recorded.

5.3.2 Data Recording Format Tuples

All information about the data-recording format for a given card is given in special tuples in the Card Information Structure. Each card that conforms to layer two of this standard shall contain at least one format tuple, defining how the data is recorded on the card.

If the format is disk-like, the format tuple may be followed by a geometry tuple. This tuple indicates the cylinder, track and sector layout for operating environments that need to treat all mass-storage devices in that way.

If the format is memory-like, the format tuple may be followed by a byte-order tuple. The byte-order tuple specifies two independent (but related) parameters: how multi-byte numbers are recorded on the media, and (for cards with 16-bit or wider data-paths) the assignment of byte addresses within each word.

Card Metaformat

5.3.2.1 The Format Tuple (CISTPL_FORMAT)

The format tuple defines the data recording format for a region (usually all) of a card. Its layout is shown in table 38.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Format tuple code (CISTPL_FORMAT, 41h).							
1	TPL_LINK Link to next tuple (n-1: at least 12, typically 20)							
2	TPLFMT_TYPE Format type code (TPLFMTTYPE_xxx); see table 39.							
3	TPLFMT_EDC					EDC Length		
	RFU	Error Detection Code type.						
4-7	TPLFMT_OFFSET – Byte address of the first data byte in this partition.							
8-11	TPLFMT_NBYTES – Number of data bytes in this partition.							
12-n	Additional information, interpreted based on value of TPLFMT_TYPE.							

Table 38 Format Tuple

Each format tuple implicitly begins a partition tuple set. Subsequent geometry, byte order, and data organization tuples are implicitly associated with the immediately previous format tuple.

Byte one of the tuple specifies the link to the next tuple, and therefore (implicitly) the length of this tuple. Two ranges of values are permitted. Normally, the value will be at least 20 (014h); however, if the format tuple is specifying a memory-like format, the value may be as little as 12 (0Ch), as bytes 13 through 21 must be zero for memory-like formats. If the partition does not use error-detecting codes, then the TPLFMT_EDCLOC field may be omitted.

Byte two of the tuple (TPLFMT_TYPE) specifies the kind of format used for this partition. The permitted values for this field are given in table 39.

<u>Code</u>	<u>Name</u>	<u>Description</u>
0	TPLFMTTYPE_DISK	This partition uses a disk-like format.
1	TPLFMTTYPE_MEM	This partition uses a memory-like format.
2-7Fh		(reserved for future standardization.)
80h-FFh	TPLFMTTYPE_VS	This partition uses a vendor-specific format.

Table 39 Format Type Codes

Byte 3 (TPLFMT_EDC) specifies the error-detection method, and the length of the error-detection code. Byte 3 is generally only meaningful for disk-like formats. Bit 7 is reserved; it must be zero. Bits 3-6 specify the error-detection code. The legal values are given in table 17. Bits 0-2 (TPLFMT_EDCLEN) specify the length in bytes of the error-detection code; this is a number between 0 and 7. The legal values for the length field are determined by the error-detection method in use.

Memory-like regions may use the PCC method of error detection.

<u>Code</u>	<u>Name</u>	<u>Description</u>
0	TPLFMTEDC_NONE	No error-detection code is used. If the length field is non-zero, space will be reserved for the check code, but no checking will be performed.
1	TPLFMTEDC_CKSUM	An arithmetic checksum is used to check the data. The length field must be 1 if this code is selected. See section 5.3.2.1.1 for details in calculating the checksum.
2	TPLFMTEDC_CRC	A cyclical redundancy check is used to check the data. The length field must be 2 if this code is selected. The CRC value is always recorded low-order byte first; see section 5.3.2.1.2 for details.
3	TPLFMTEDC_PCC	An arithmetic checksum is used to check the data; however, a single checksum is provided for the entire data partition. This technique is intended for use with static data on ROM or OTPROM cards. The PCC code itself is recorded in byte 18 of the tuple (field TPLFMT_EDCLOC, byte 0). The length field must be 1 if this option is selected.
4-7h		(Reserved for future standardization.)
8h-Fh	TPLFMTEDC_VS	A vendor-specific method of error checking is used.

Table 40 Error Detection Type Codes

Card Metaformat

The code in TPLFMT_EDC only specifies the method to be used to verify data integrity. To determine whether the code is to be interleaved with the data or stored in a separate table, the value in TPLFMT_EDCLOC must be consulted.

Bytes 4-7 (TPLFMT_OFFSET) specify the absolute byte address of the first data byte governed by this tuple. The value is stored as a 32-bit quantity, LSB first.

Bytes 8-11 (TPLFMT_NBYTES) specify the number of bytes in the partition, including (if present) the error-detection codes. The value is stored as a 32-bit quantity, LSB first.

5.3.2.1.1 The Format Tuple for Disk-like Regions

When the TPLFMT_TYPE field of the format tuple has the value TPLFMT_DISK, bytes 12 through 21 of the tuple are interpreted as shown below.

Byte	7	6	5	4	3	2	1	0
12-13	TPLFMT_BKSZ Block size. For unblocked formats, this value should be 0. This field corresponds to the number of data bytes/sector. The value in this field must be a power of 2.							
14-17	TPLFMT_NBLOCKS Number of data blocks in this partition.							
18-21	TPLFMT_EDCLOC Location of the error-detection code. If zero, the error detection code is interleaved with the data blocks. If non-zero, the error-detection code is stored in a linear table starting at the specified address on the card. If using PCC, the first byte of this field contains the check code: bytes 19-21 must be zero if present.							

Table 41 Format Tuple for Disk-like Regions

Bytes 12-13 (TPLFMT_BKSZ) specify the number of data bytes in each block in the partition. This value does not include error check bytes. The value in this field must be a power of 2 between 128 and 2048; this standard recommends that 512 be used wherever possible. The value is stored as a 16-bit quantity, LSB first.

Bytes 14-17 (TPLFMT_NBLOCKS) specify the number of data blocks in the partition. This value is stored as a 32-bit quantity, LSB first. The quantity:

$$\text{TPLFMT_NBLOCKS} \cdot (\text{TPLFMT_BKSZ} + \text{TPLFMT_EDCLEN})$$

shall be less than or equal to TPLFMT_NBYTES.

Bytes 18-21 (TPLFMT_EDCLOC) specify where the error-detection codes are stored. This value is stored as a 32-bit quantity, LSB first. If the value stored in this location is zero, or if this field is not present, then codes (if present) are interleaved with the data blocks, with the code for a given data block following immediately after that block. If the value stored in this location is non-zero, it shall be the address of the first byte of the error-detection code table. This table shall be an array of values, with TPLFMT_NBLOCKS entries, containing the error-detection codes for each data block. Each entry in the table shall be TPLFMT_EDCLEN bytes long. The value stored in TPLFMT_EDCLOC shall be at least TPLFMT_OFFSET, and shall be no greater than $TPLFMT_OFFSET + TPLFMT_NBYTES - (TPLFMT_EDCLEN \cdot TPLFMT_NBLOCKS)$.⁴

If PCC error checking is selected, then the TPLFMT_EDCLOC field is used to add the actual LRC value, rather than pointing to the cell that holds the PCC.

The bit TPLFMT_EDC_RFU is reserved for future use and shall always be zero.

Table 42 summarizes some possible error-detection strategies.

EDC Format	EDC Length	EDC Location	Description
TPLFMTEDC_NONE	0	0	No error checking is performed; no room is reserved for error-detection tables. The data blocks are recorded sequentially.
TPLFMTEDC_NONE	2	0	No error checking is performed; but room is reserved for a two-byte error-detection code after each data block.
TPLFMTEDC_NONE	1	non-zero	No error checking is performed; but room is reserved for an out-of-line table of error-detection codes, with one byte per data block. The data blocks themselves are recorded contiguously.
TPLFMTEDC_CKSUM	1	non-zero	Data is checked using a one-byte arithmetic checksum of the data. The checksum is stored in an out-of-line table. The data blocks themselves are recorded contiguously.
TPLFMTEDC_CRC	2	0	Data is checked using SDLC CRC codes. The check-code for a data block is stored immediately following the data block.
TPLFMTEDC_PCC	1	special	Entire partition is checked using a one byte arithmetic checksum. The checksum is stored in the TPLFMT_EDCLOC field of the tuple itself.

Table 42 Error Detection Format Summary

⁴ In other words, the table must be entirely contained in the range of bytes between TPLFMT_OFFSET and $TPLFMT_OFFSET + TPLFMT_NBYTES - 1$. Since the first data byte of block 0 resides at TPLFMT_OFFSET, the standard requires that the EDC table appear after all the data blocks in the partition. The standard does not require that the table occur immediately after the last block, nor does it preclude use of spare space for vendor-specific purposes.

Card Metaformat

5.3.2.1.2 The Format Tuple for Memory-like Regions

When the TPLFMT_TYPE field of the format tuple has the value TPLFMT_MEM, bytes 12 through 21 of the tuple are interpreted as shown below.

Byte	7	6	5	4	3	2	1	0
12	TPLFMT_FLAGS Various flags (reserved)						AUTO	ADDR
13	(reserved; must be zero)							
14..17	TPLFMT_ADDRESS Physical address at which this memory partition should be mapped, if so indicated by TPLFMT_FLAG. Four bytes, stored LSB first.							
18..21	TPLFMT_EDCLOC Error-detection code location, with same meaning as for disk-like regions. Used for PCC checking only; byte 18 holds the check value, and bytes 19..21 must be zero or omitted.							

Table 43 Format Tuple for Memory-like Regions

As with the format tuple for disk-like regions, zero bytes at the end of the format tuple for memory-like regions may be omitted; to do this, the next tuple must begin immediately after the last non-zero byte in the format tuple.

Byte 12 (TPLFMT_FLAGS) contains several control bits.

- Bit 0 (TPLFMTFLAGS_ADDR), if set, indicates that bytes 14-17 (TPLFMT_ADDRESS) represent a physical address to be associated with the first byte of this region. If clear, bytes 14-17 do not represent a physical address.
- Bit 1 (TPLFMTFLAGS_AUTO), tells the system whether to automatically map the region into memory when the card is inserted (or at system start-up). If set, the system should attempt to map the region into memory; if TPLFMTFLAGS_ADDR is set, the system should attempt to map the code at the specified address. A system shall ignore these field if it cannot perform the specified mapping; it may also, at the designer's option, ignore these fields even if it could perform the mapping.

Byte 13 is reserved, and must be set to zero if present.

Bytes 14 through 17 (TPLFMT_ADDRESS) represent the physical address at which the partition should be mapped into the host's address space. For 80x86-family machines, this is a linear address, not a segment/offset address. If the flag TPLFMTFLAGS_ADDR is not set, then this field is reserved and must be zero.

NOTE TO IMPLEMENTORS: A system can be fully with this standard and not honor these fields. The automatic mapping feature is not intended for general purpose use or for building interchangeable BIOS extensions for general purpose systems. The automatic mapping feature of this standard is included for use in low-cost embedded systems, and is not intended as a general execute-in-place specification. Many important issues are deliberately not addressed by this part of the standard, such as what to do when the card is removed, or how to resolve conflicts when cards in different sockets both need to be mapped to a specific physical address. It is anticipated that general purpose DOS-based systems will ignore these fields.

Bytes 18-21 (TPLFMT_EDCLOC) have the same meaning for memory-like regions that they have for disk-like regions. A memory-like region has only two options for error checking: none or PCC. Therefore, if this field is used, byte 18 contains the check code, and bytes 19-21 are reserved and must be zero.

5.3.2.1.3 Arithmetic Checksums As Error-Detection Codes

Arithmetic checksums shall be computed by summing together all the data bytes of the block using eight-bit two's-complement addition, ignoring any overflow that occurs. The resulting sum shall be stored in an external table (for block-by-block checksum) or in the format tuple itself (for PCC checking).⁵

5.3.2.1.4 CRC Error-Detection Codes

CRC codes shall be computed using the SDLC algorithm.⁶ In this algorithm, the data to be checked is considered as a serial bit stream, with the low-order bit of the first byte taken as the first bit of the stream. This bit stream is conceptually taken as the coefficients of a polynomial in x^n , where n is the number of bits in the stream, and where the first bit is the coefficient of the term in x^{n-1} . This polynomial is divided (modulo 2) by the polynomial $x^{16} + x^{12} + x^5 + 1$, leaving a remainder of order 15 or less.⁷ The one's complement of this remainder is the error check code; it is recorded with the complemented coefficient of x^{15} as its least-significant bit, and with the complemented coefficient of x^0 as its most-significant bit.

The SDLC CRC has a convenient property: when the check code is appended to the data stream, and the algorithm is run on the result, the remainder will always be $x^{12} + x^{11} + x^{10} + x^8 + x^3 + x^2 + x^1 + x^0$ (assuming that neither the data nor the CRC have been corrupted).

Despite its complicated formal definition, the SDLC CRC is quite easy to compute both in hardware and in software. Commercially available chips (such as the Fairchild 9401) can compute the CRC directly from a serial data stream. There are several well known methods for computing the CRC one byte at a time using a lookup table. Even so, computing a CRC in software is somewhat slower than computing a simple checksum.

5.3.2.1.5 Byte Mapping for Disk-Like Media

Within a disk-like partition, cards with data-paths wider than 16 bits shall be byte mapped with the lowest byte address of each word corresponding to the least-significant byte of that word, and with increasing byte addresses corresponding to increasingly significant bytes.

⁵ This method has the disadvantage that the checksum of a block of zero data is also zero; however, it is consistent with current practice. We don't anticipate that implementors will want to interleave checksums and data; if that's desired, the standard should introduce another error-detection code type, one's complement of checksum.

⁶ Also known as CRC-CCITT or HDLC CRC.

⁷ As an additional refinement, the initial remainder is set to all ones, rather than all zeroes; this causes the CRC code for a block of all zeroes to be non-zero.

Card Metaformat

5.3.2.2 The Geometry Tuple (CISTPL_GEOMETRY)

This tuple shall only appear in partition tuples for disk-like partitions. It provides instructions to those operating systems that require that all mass-storage devices be divided into cylinders, tracks and sectors.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Geometry tuple code (CISTPL_GEOMETRY, 42h).							
1	TPL_LINK Link to next tuple. (at least 4)							
2	TPLGEO_SPT Sectors per track.							
3	TPLGEO_TPC Tracks per cylinder.							
4..5	TPLGEO_NCYL Number of cylinders, total.							

Table 44 Geometry Tuple

Byte 2 (TPLGEO_SPT) specifies the number of sectors per simulated track on the memory card. This is a number between 1 and 255. A value of zero is not permitted.

Byte 3 (TPLGEO_TPC) specifies the number of tracks per simulated cylinder on the device. This is a number between 1 and 255. A value of zero is not permitted.

Bytes 4-5 (TPLGEO_NCYL) specify the number of simulated cylinders on the device. This is a number between 1 and 65535, stored as a 16-bit integer, LSB first.⁸

The product

$$\text{TPLGEO_NCYL} \cdot \text{TPLGEO_TPC} \cdot \text{TPLGEO_SPT}$$

shall be less than or equal to the number of blocks recorded in field TPLFMT_NBLOCKS of the format tuple (section 5.3.2.1.1 page 72).

⁸ This value is one greater than the same quantity as represented by the PC BIOS. This standard records the number of simulated cylinders; the PC BIOS records the maximum cylinder number. Since cylinder numbers on the PC start at zero, the maximum cylinder number on the PC is one less than the number of cylinders.

5.3.2.3 The Byte-Order Tuple (CISTPL_BYTEORDER)

This tuple shall only appear in a partition tuple set for a memory-like partition. It specifies two parameters: the order for multi-byte data, and the order in which bytes map into words (for 16-bit or wider cards).

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Tuple code (CISTPL_BYTEORDER, 43h).					
1	TPL_LINK		Link to next tuple; should be at least 2.					
2	TPLBYTE_ORDER		Byte order code: see table 42.					
3	TPLBYTE_MAP		Byte mapping code: see table 43.					

Table 45 Byte Order Tuple

Byte 2 (TPLBYTE_ORDER) specifies the byte order for multi-byte numeric data. Symbolic codes for this field begin with the text "TPLBYTEORD_", and are listed in table 46.

Code	Name	Description
0	TPLBYTEORD_LOW	Specifies that multi-byte numeric data is recorded in little-endian order.
1	TPLBYTEORD_HIGH	Specifies that multi-byte numeric data is recorded in big-endian order.
2-7Fh		Reserved for future standardization.
80h-FFh	TPLBYTEORD_VS	Vendor-specific.

Table 46 Byte Order Codes

Byte 3 (TPLBYTE_MAP) specifies the byte mapping for 16-bit or wider cards. Symbolic codes for this field begin with the text "TPLBYTEMAP_", and are listed in table 47.

Code	Name	Description
0	TPLBYTEMAP_LOW	Specifies that byte 0 of a word is the least-significant byte (multi-byte cards).
1	TPLBYTEMAP_HIGH	Specifies that byte 0 of a word is the most-significant byte (multi-byte cards).
2-7Fh		Reserved for future standardization.
80h-FFh	TPLBYTEMAP_VS	Vendor-specific.

Table 47 Byte Mapping Codes

Card Metaformat

If a byte-order tuple is not present, the data shall be recorded using little-endian byte order (TPLBYTEORD_LOW), and shall be mapped with byte 0 of each word corresponding to the least-significant byte (TPLBYTEMAP_LOW).

For applications involving DOS file systems, little-endian byte order and low-to-high byte mapping are mandatory.

5.3.3 Standard Data Recording Formats

This standard allows great flexibility in adjusting the card format to meet specific requirements. For simplicity, this standard further specifies *recommended formats* – formats which are expected to be commonly used.

- **Generic** – the bytes are recorded in 512-byte blocks with no error checking; the first data byte of the card appears at byte address 512 (200h).
- **Single-byte checksum format** – the bytes are recorded in 512-byte blocks, with a separate region reserved for error-checking codes, and with a sector buffer.
- **Two-byte Embedded CRC format** – the bytes are recorded in 512-byte blocks, with each block followed by a 16-bit CRC.
- **Raw byte format** – the bytes are recorded sequentially in an unblocked form.

In order to maintain a reasonable degree of interchangeability, this standard recommends that all layer-2 conforming implementations be able to read and write generic-format cards.

When an implementation is presented with a card whose format is not supported by that implementation, the implementation shall refuse to write on the card, except to reinitialize it. If the basic format is not supported by the implementation at all, the implementation shall return an error to applications whenever they attempt to access the card. The implementation may allow read-only access to a card whose basic format is supported but whose error-detecting code is not. For example, an implementation that only supports the creation of generic-format cards could allow single-byte-checksum cards to be read, by ignoring the checksum bytes. With a little more sophistication, the implementation could also allow embedded-CRC format cards to be read.

5.3.4 Mixed Data Formats

Layer 2 allows a card to have multiple partitions, with each partition having its own data-recording format. In this case, there shall be one Format Tuple (section 5.3.2.1, page 70) in the Card Information Structure for each partition on the card. The additional tuples that refer only to a given partition shall appear immediately following the format tuple that defines the partition.

An implementation is not required to support multiple partitions on a single card. When presented with a card with multiple partitions, an implementation may:

- Only allow access to the first partition (if supported)
- Scan the CIS for the first partition type of a supported type, and only allow access to that partition.
- Deny access to the card.

Note to Implementors: We anticipate that for most applications, only 1 or 2 regions will be required. (Two regions would be used by a ROM card that contained both executable ROM images and a DOS file system).

Card Metaformat

5.4 Data Organization (Layer 3)

This layer defines the data organization of a particular partition on a memory card. At this level, the possibilities become manifold. Some examples are:

- A partition can contain a DOS file system (or a file system for some other operating system). This can be used with any disk-like level 2 format.
- A partition can contain a FlaSh file system (used with memory-like formats).
- A partition can use a vendor-specific organization.
- A partition can use an application-specific organization.

Layer 3 of this standard provides an unambiguous means of specifying the organization of the partition.

5.4.1 Data Organization Tuples

All information about the organization of a given partition is given in special tuples in the Card Information Structure. Each card that conforms to layer three of this standard shall contain at least one data-organization tuple for each partition defined on the card.

At present, the data-organization tuple is the only tuple defined by layer three.

5.4.1.1 The Organization Tuple (CISTPL_ORG)

The Organization Tuple appears in the list of partition-specific tuples that follows each format tuple. It has the format shown in table 48.

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE Tuple code for this tuple. (CISTPL_ORG, 46h)							
1	TPL_LINK Link to next tuple (at least n-1).							
2	TPLORG_TYPE Data organization code							
3..n	TPLORG_DESC Text description of this organization, terminated by 00h.							

Table 48 Data Organization Tuple

Byte 2 (TPLORG_TYPE) specifies the type of data organization in use. The possible values of this byte are given in table 49.

<u>Code</u>	<u>Name</u>	<u>Description</u>
0	TPLORGTYPF_FS	This partition contains a file system. The description field specifies the file system type and version.
1	TPLORGTYPF_APP	This partition contains application-specific information. The description field specifies the application name and version.
2	TPLORGTYPF_ROMCODE	This partition contains executable code images. The description field specifies the name and version of the organization scheme.
3-7Fh		(Reserved for future standardization.)
80h-FFh	TPLORGTYPF_VS	This partition uses a vendor-specific organization. The contents of the description field are vendor-specific.

Table 49 Data Organization Codes

Bytes 3 through the end of the tuple (TPLORG_DESC) contain a NUL-terminated ASCII-text description of the organization. For file system organizations, this field should specify the file system type. This field shall contain only characters in the printing ASCII set, 020h through 07Eh. (For international use, one or more CISTPL_ALTSTR tuples can follow this tuple.)

For DOS file systems, TPLORG_TYPE shall contain TPLORGTYPF_FS, and TPLORG_DESC shall contain the string "DOS". For FlaSh file systems, TPLORG_TYPE shall contain TPLORGTYPF_FS, and TPLORG_DESC shall contain the string "FlaSh".

The intent of this field is two-fold.

- For operating systems with sufficient flexibility, it allows the appropriate file-system driver to be selected based on the value of this field.
- If a card cannot be read due to software incompatibilities, a utility program can display the contents of this field along with other card information to inform the user as to what kind of information is really on the card.

Card Metaformat

5.5 System-Specific Standards (Layer 4)

Layer four of this standard specifies things that are only relevant in certain operating environments. At present, all layer-four standards are specific to the DOS environment. The following DOS-specific standards are defined:

- An interchange format for cards formatted with the DOS file system (section 5.5.1).
- A standard for directly-executable programs (specified in a separate document).
- A standard for interpreting older cards formatted without the Card Information Structure (section 5.5.3).

5.5.1 Interchangeable Card Format

This standard would be of little use if it did not allow the free interchange of information between DOS systems. Rather than limiting all DOS implementations to a single format, this standard requires that all implementations support the following format *in addition* to any other formats.

The Interchangeable card format has the following characteristics:

Layer 1: the card information structure shall contain at least a device information tuple.

Layer 2: the card information structure shall contain the following tuples.

1. Level-2 information tuple, with the following fields set.
 - TPLLV2_COMPLY shall be 0.
 - TPLLV2_NHDR shall be 1, indicating that only one copy of the CIS is present.
 - TPLLV2_VSPEC8 and TPLLV2_VSPEC9 shall be zero.
2. A single format tuple. This tuple shall indicate that the partition uses a disk-like format with 512-byte blocks. It shall further indicate that the card uses no error detection code, that the EDC length is zero, and that the first data byte of the card appears at byte 512, or higher, of the card. This tuple shall indicate that the partition covers all but the first 512 bytes of the card, and that there are (partition_size / 512) blocks in the partition.

In addition, the CIS may optionally contain the following tuples:

- A single geometry tuple. If present, this tuple shall contain information that matches the information presented in the boot block BPB.
- A single card-initialization date tuple.
- A single battery-replacement date tuple.

Layer 3: the card shall contain a DOS-compatible file system. The boot sector shall be recorded in data block zero (that is to say, starting at byte 512 of the card). The BPB in the boot sector shall describe the geometry of the file system in any convenient fashion. This standard recommends that geometry parameters be set to appropriate powers of two.

5.5.2 Execute In Place

The proposed format for cards supporting direct execution of application programs from the card ("execute in place") is described in a separate document.

5.5.3 Interpreting Cards Without Card Information Structures

Some existing systems use RAM cards with a pseudo-floppy organization.

Pseudo-floppy cards have the following format:

A series of contiguous logical sectors, as viewed by MS-DOS

The BIOS sector addressing scheme (head, cylinder, sector) is mapped one-to-one to the logical sectors

The logical sectors are arranged exactly as in the case of a floppy disk, i.e. the first is the **BOOT SECTOR**, then come a variable number of **FILE ALLOCATION TABLE (FAT)** sectors, then a number of **ROOT DIRECTORY** sectors, and finally the card is filled up with **DATA** sectors

Sectors are a standard MS-DOS size: 128, 256 or 512 bytes; 512 byte sectors are the standard default, since this allows room for executable code in the first (BOOT) sector

The **BOOT** sector is defined exactly as for a floppy; it thus contains the *BIOS Parameter Block (BPB)*, and thus a definition of the number of bytes per sector, number of copies of the FAT etc.

5.5.3.1 Boot Sector Structure

The boot sector would typically be 512 bytes, if it needs to contain bootstrap code as well as the BPB. Otherwise, it could be as small as 128 bytes. However, the first 30 to 50 bytes of the bootstrap sector are always recorded in a standard format.

The first three bytes of the boot sector are reserved for a short or a near jump:

E9 XX XX
or
EB XX 90

This gives us a simple way to detect a pseudo-floppy boot block. There then follows the BIOS Parameter Block.

The format of the boot-sector header is shown in tables 47 and 48, below.

Note: The information in these tables is controlled by DOS; the data formats are included here only for reference.

Card Metaformat

Byte	7	6	5	4	3	2	1	0
0-2	Short or near jump: 0E9h XX XX or 0EBh XX 090h.							
3-0Ah	System ID (OEM name and version) (8 bytes)							
0Bh-0Ch	Bytes per sector (2 bytes)							
0Dh	Sectors per cluster							
0Eh-0Fh	# reserved sectors							
10h	Number of FATs							
11h-12h	# root directory entries							
13h-14h	# sectors in logical volume							
15h	Media Descriptor Byte							
16h-17h	# sectors per FAT							
18h-19h	# sectors per track							
1Ah-1Bh	number of heads							
1Ch-1Dh	# hidden sectors							

Table 50 DOS Boot-Block Structure

With DOS 4.0 and later, the following additional fields are defined:

1Eh-1Fh	# hidden sectors (most significant word)
20h-23h	# sectors in logical volume (4 bytes)
24h	Physical drive number
25h	Reserved
26h	Extended boot signature (29h)
27h-2Ah	Volume ID (binary) (4 bytes)
2Bh-35h	Volume label (11 bytes)
36h-3Dh	Reserved (8 bytes)

Table 51 Extended BPB

The BPB and Header Information

The information in the BPB can be accessed by the device driver, whether embedded in the ROM BIOS, or external to the BIOS, or separately loaded as an MS-DOS *Installable Device Driver*. In this way, differing configurations of ROM BIOS to Logical Sector Mappings can be used; a card can have "multiple heads", for instance.

5.5.3.2 Handling Pseudo-Floppies in a Conforming System

Pseudo-floppies can easily be handled from within a conforming system, if the following procedure is followed during card-insertion processing:

1. Read the first byte of attribute memory. If it is 01h, process the CIS from attribute memory in the ordinary way. If all metaformat information is present in attribute memory, or in common memory as specified by the attribute memory, then this is not a pseudo-floppy. If a CIS is present in attribute memory, but no layer-2 information is present, or if no CIS is present, proceed to step 2.
2. Read the first few bytes from the card's common memory (starting at physical address 0) into a local buffer.
3. Compare the first five bytes of the buffer to the CIS link-target tuple signature (13h, xx, "CIS", where xx is a link value in the range [03h, FEh]). If they match, then this card has the CIS metaformat structure in RAM (and probably has no attribute memory); use the ordinary processing rules. Otherwise, no CIS is present at all; proceed to step 4.
4. Compare the first three bytes of the buffer to the DOS boot block signature: 0E9h, XXh, XXh, or 0EBh, XXh, 90h. If it matches, assume that this is a DOS-format pseudo-floppy. Extract the relevant geometry and block-size information from the BPB, assume that there is no error detection, and assume that the card consists of a single data partition encompassing the number of blocks indicated in the BPB.

Card Metaformat

5.6. Compatibility Issues

5.6.1 Buffer Pages

Some vendors use a buffer page to improve the reliability of memory cards in the face of power failures. This standard does not directly provide a means for specifying the location of the buffer page. Space can easily be reserved for a buffer page by proper adjustment of the values in the format tuple. If needed, a vendor-specific tuple can be added to specify the location of the buffer page within the partition.

5.6.2 Formatting Cards Under DOS

Use of a CIS does not require a special (non-DOS) format utility in the common case where the entire card is to be formatted as a DOS file system. For example, the memory card BIOS could determine all the relevant information (including the pseudo-disk geometry) using information that is passed to the BIOS format function, and transparently construct the CIS during the format operation. DOS is unaware of the existence of the CIS; when it reads block 0 of the disk, the BIOS returns the first user-accessible block.

For multi-format cards, a special format utility is required, in order to get the proper partition information in the CIS.

5.6.3 Adapting Existing Software

The intent of this standard is that existing BIOS and DOS driver software be easy to migrate to the new standard. This process involves making the following modifications to system software:

1. The existing software should be examined to determine how its data layout corresponds to the options presented by this standard.
2. From this, design a standard CIS which describes how the data is presently laid out; however, move everything up (for example, by 512 bytes) to leave room for the CIS at the front of the card. Place a copy of this CIS in the code for the BIOS or driver.
3. Modify the driver or BIOS to check for a CIS signature whenever a card may have been changed.
 - If the card has a CIS signature, do a byte-by-byte comparison of the CIS on the card with the CIS in the BIOS or driver code. If the CIS matches, set a flag indicating that a 'standard-conforming card' is installed. If the CIS does not match, set a different flag indicating that an card with an unsupported format is installed.
 - For flexibility, the comparison operation could ignore device-information tuples and (possibly) the partition length field of the format tuple. It could also be structured to allow the tuples to appear in any order.
 - If the card does not have a CIS signature, assume that it is an old-style card. Set a flag indicating that a supported, non-conforming card is installed.
4. Whenever I/O is performed, check the flags.
 - If an unsupported card is installed, return an error.
 - If a supported non-conforming card is installed, convert the block address to the byte address just as you did previously.
 - If a supported conforming card is installed, convert the block address to the on-card byte address just as you did previously; then add 512 (or whatever base address you decided on in step 2).

Using the address you've just calculated, perform the I/O operation.

5.6.4 Using this Standard with Non-Standard Hardware

It is possible to use the metaformat portion of this standard to organize data, even in the absence of attribute memory support. To do so, we recommend the following adaption:

Format the card so that a card information structure begins at location zero on the card; make the first tuple a LINK_TARGET tuple. This gives software a reasonably unambiguous way to determine whether the card is formatted or not.

This page is intentionally left blank

SECTION 6

FAT FILE SYSTEM

FAT File System

6. FAT FILE SYSTEM

6.1 Introduction

This section describes the essential elements of how the DOS FAT file system is implemented on IC memory cards. This is not a full description of the FAT file system, as it is assumed the reader is familiar with this technology.

The FAT file system is the default data transfer standard recognized by DOS. It is recognized that the FAT does not make the most efficient use of memory for all types of memory technology. However, it does offer a familiar, compatible data structure requiring little system software overhead. Future versions of DOS suited for specific technologies or applications will be incorporated into the FAT file system at a later date.

6.2 Supporting FAT on IC cards

FAT file systems will be contained within partitions on IC memory cards. The partition is described by the PCMCIA metaformat; the metaformat will include information to identify partitions as FAT partitions. The card device driver will present the partition as a block device to DOS. The partition will begin at sector 0 beginning at the first address within the partition.

The logical format of a FAT partition is described by the BIOS standard DOS data structure. It is incorporated into the boot sector of the partition. Two different formats are supported, one for partitions smaller than 32Mb and one for partitions greater than or equal to 32Mb. Note that DOS 3.3 and later versions support partitions larger than 32Mb.

Offset	Size	Contents
+00	3	ID (Jump instruction to boot sector) It must be EBh,xxh, 90h or 91h
+03	8	OEM name and version
+0B	2	Bytes per sector
+0D	1	Sectors per allocation unit
+0E	2	Reserved sectors count
+10	1	Numbers of FATs BPB
+11	2	Numbers of root directory entries
+13	2	Numbers of sectors in root directory
+15	1	Media descriptor
+16	2	Numbers of sectors per FAT
+18	2	Sectors per track
+1A	2	Numbers of heads
+1C	2	Numbers of hidden sectors
+1E	22h	Reserved for future use
+40	—	Bootstrap code etc.

Table 52 Boot Record Format for Small Partitions

Contents
ID (Jump instruction to boot cord) It must be EBh,xxh 90h or E9h,xxh,xxh
OEM name and version
Bytes per sector
Sectors per allocation unit
Reserved sectors count
Numbers of FATs BPB
Numbers of root directory entries
Numbers of sectors in logical image
Media descriptor
Numbers of sectors per FAT
Sectors per track
Numbers of heads
Numbers of hidden sectors
Number of logical sectors
Reserved section and bootstrap code

Record Format for Large Partitions

used, the image size at offset 13h is set to 0, and the alternative field at

values chosen for sectors per track and number of heads be powers of
by the device driver. It is also recommended that the media
to avoid conflicts with older versions of DOS.

AT file systems must be structured as standard DOS block device
quests from DOS in terms of sectors and tracks; they must be prepared
to physical card addresses, based on the geometry parameters defined
the cards with a variety of logical formats, and should be prepared to
than being tied to a specific format.

This page is intentionally left blank

SECTION 7
EXECUTE IN PLACE

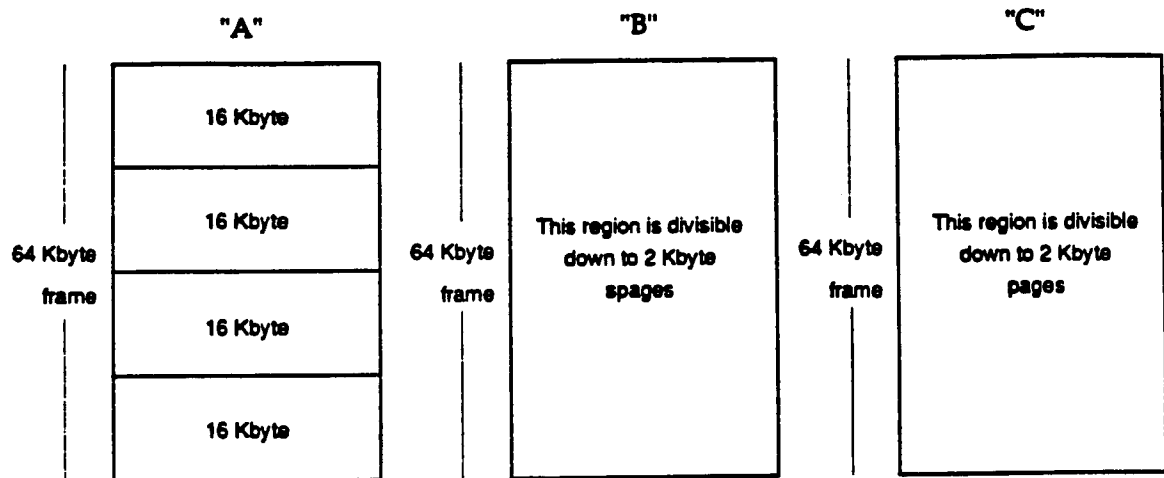
Execute In Place

7. EXECUTE IN PLACE

7.1 General

Common program execution in MS-DOS computers occurs in system RAM space. These programs are often time loaded from magnetic media based I/O devices such as floppy disks, hard disks, or tape drives. Programs stored in an IC memory card can be executed directly when appropriate mapping methods are employed. This committee recognised this need and provided methods for program eXecution-In-Place (XIP). A framed window memory paging structure is used.

7.2 Page/Frame Definition



1. Three regions ("frames") of 64 Kbytes are defined.
2. These three regions are not required to be contiguous to each other.
3. Within each region the 64 Kbytes must be contiguous.
4. One region must be subdivided into four pages of 16 Kbytes each.
5. The other regions may be "single pages" of 64 Kbytes, or they may be subdivided into pages as small as 2 Kbytes.
6. There is no specific requirement to implement any of the regions in hardware. Software emulation and "copying" schemes are acceptable.
7. Software vendors may assume that the frame "A" is implemented in hardware to optimize performance. They may further assume that regions "B" and "C" may be implemented by "copying", which would make it desirable to place the root overlay or other infrequently paged code in regions "B" and "C".
8. All API (Application Programmer Interface) issues including definition of read-only pages are left to a later release.

APPENDICES

APPENDIX 1

Metaformat Glossary

attribute memory: PCMCIA/JEIDA standard memory cards provide a separate memory address space for recording fundamental card information. This memory is intended to be used by the card manufacturer to record basic configuration information. This memory is selected by asserting the /REG line on the card interface. It is typically, but not necessarily, read-only.

Attribute memory space need not be physically distinct from common memory space; but it must be logically distinct.

basic compatibility layer: The layer of this standard (layer 1) which mandates the use of a card-information structure (CIS) at the beginning of any complying card.

big-endian byte order: a means of specifying the order in which multi-byte numeric objects are recorded, when broken into bytes. Big-endian byte order specifies that the most-significant byte shall be recorded in the lowest byte address; bytes of decreasing significance shall be recorded sequentially in subsequent bytes. Cf. little-endian byte order.

block: for disk-like data formats, a block is the fixed-length sequence of bytes. In such formats, data must usually be read or written as a series of one or more blocks.

byte: in this standard, a byte is eight bits.

byte mapping: the sequence in which byte data is recorded on cards. For 8-bit memory cards, the byte mapping is one-to-one, and not at issue for standardization. For 16-bit and wider cards, the byte mapping within words of the card is arbitrary, and so is governed by this standard.

buffer page: a region of memory on a card used to improve reliability when updating a card. A buffer page typically includes an indication of the region of the card being updated, an image of the desired value for the region of the card, and a flag that indicates that the buffer page is valid. If power fails while a card is being updated, the buffer page can be used to automatically complete the transfer when power is restored.

Card Information Structure: a data structure written at the beginning of every card that complies with this standard, containing information about the formatting and organization of the data on the card.

checksum: an arithmetic error-checking code for data recording based on summing the bytes of data to be checked. Checksums are frequently used by systems that perform error-checking in software.

CIS: card information structure.

common memory: PCMCIA/JEIDA standard cards provide two memory address spaces. The term "common memory" denotes the primary address space, containing the memory used for application data storage. See also attribute memory.

CRC: cyclical redundancy check.

cyclical redundancy check: an error-checking code for data recording based on bitwise polynomial division of the data bytes to be checked. As used in this standard, refers to the 16-bit SDLC version of this code, using the polynomial $x^{16} + x^{12} + x^5 + 1$, with the check-register initialized to all ones. CRCs are typically used by systems that perform error-checking in hardware.

cylinder: a unit of disk organization. A disk is typically viewed as a collection of cylinders. Each cylinder on a disk is divided into tracks; each track is further divided into sectors. Typically, all of the sectors within a cylinder can be accessed without moving the arm of the disk. See *sector*.

data organization: the logical organization of data on a card, independent of the data-recording format. The data organization of a memory card will almost always be some kind of file system.

data organization layer: the layer of this standard covering the data organization of the card.

data-recording format: the organization of a memory card into sequences of bytes that are updated or accessed by a single logical operation. The data-recording format of a card includes such details as whether the card's data is organized into blocks of bytes; whether the card includes error checking codes for each block; and so forth. The data-recording format does not specify whether a file system is used. The data-recording format of a card is akin to the physical format of a diskette. Cf. *data organization*.

data recording format layer: the layer of this standard (layer 2) that specifies the data-recording format of a card.

DOS: the disk operating system for 80x86 architecture systems, such as the IBM PC. DOS is available in several different versions, which are largely compatible with each other; the term generically designates all of them.

EDC: *error-detection code*

EEPROM: Electrically-Erasable Programmable Read-Only Memory. A non-volatile memory device which can be programmed electrically, and in which individual bytes can be erased electrically. Usually writes and erasures are much slower than reads.

EPROM: Erasable Programmable Read-Only Memory. A memory device which can be programmed electrically, and erased in bulk by some means, usually by exposure to ultraviolet light.

error-detection code: a numeric code derived from the contents of a data block, used to determine whether the data read from the block are probably correct.

file system: an operating-system specified method of structuring data on a mass-storage device. A file system standard consists of a set of data structures and the rules by which those structures are interpreted. We sometimes say that a card has a file system recorded on it; by this we mean that an operating system utility program has placed the appropriate information on the card, allowing the card to be interpreted and manipulated by the operating system.

Not all cards have file systems on them. Some cards are managed directly by application programs.

flash EPROM: a type of EPROM that can be electrically erased. It differs from EEPROM in that generally the entire memory must be erased at once.

FlaSh: a trademark of Microsoft, describing a file system designed for use with UV-erasable or Flash EPROM memory cards.

Kbyte: kilobyte. 1 Kbyte = 1024 bytes.

little-endian byte order: a means of specifying the order in which multi-byte numeric objects are recorded, when broken into bytes. Little-endian byte order specifies that the least-significant byte shall be recorded in the lowest byte address; bytes of increasing significance shall be recorded sequentially in subsequent bytes. Cf. *big-endian byte order*.

ISO 646 IRV: International Standards Organisation standard number 646 (Character codes), International Reference version. A character set very similar to ASCII, used internationally for representing textual information. It differs from ASCII only in that code 24h represents the international currency symbol rather than the dollar sign ("\$"). Except in the alternate/national string tuple, all character data shall be represented using the printing characters from this character set.

LSB: least-significant byte.

metaformat: in this standard, the word metaformat is used to encompass the contents, layout, and interpretation of the card information structure. The PCMCIA Metaformat Standard is outlined in Section 5 of this document.

one-time programmable: A term describing memory that can be programmed to a specific value once, and thereafter cannot be changed (or can only be revised in a limited way). One-time programmable EPROMs are ordinary EPROMs that have been packaged in such a way that ultra-violet light cannot be used to erase the contents of the EPROM. Such packaging is usually less expensive.

OTP: one-time programmable

paragraph: on Intel 80x86 family machines, a paragraph is a block of sixteen bytes, aligned on a sixteen-byte boundary.

partition: a region of a mass storage device. In this standard, partitions are used to allow a single card to contain two different kinds of data; for example, a card might contain a normal DOS file system in one partition, and directly-executable ROM images in another partition. Most RAM cards will contain only a single partition that contains all the usable storage of the device.

partition check code: A simple method of verifying the contents of an entire partition. A checksum is computed by summing together all the data bytes of the partition; this sum is compared to a value stored in the format tuple that defines the partition. This method is typically used for partitions that change relatively infrequently, such as data partitions in OTP memory.

PCC: partition check code

PSP: program-segment prefix. Under DOS, the PSP is the primary data structure for a process, containing its command line, information about exception handling, and so forth.

reserved: As used in this standard, a reserved field or code value is set aside for use in future standardization. Vendors shall not use reserved fields or code values for any purpose except compliance with future versions of this standard.

sector: As used in this standard, a sector is the fundamental data storage unit of a disk. A sector is the smallest unit of data that can be individually read or updated. Disk sectors correspond to memory card blocks.

TPL: abbreviation used in symbolic codes to represent the word "tuple".

TSR: acronym for terminate-and-stay-resident. Under DOS, a TSR is a program that is loaded semi-permanently into memory, extending the system's functionality.

tuple: in this standard, a tuple is a block that appears in the Card Information Structure. Tuples are used to record various items of information about the card layout. All tuples have a common format, shown in Table 19, page 49.

vendor specific: in this standard, this term indicates bits, fields, or code values that are specific to a particular vendor and are not defined by this standard. This standard further distinguishes two kinds of vendor: the card manufacturer, and the supplier of the card data contents.

word: as used in this standard, a word is the smallest addressable unit of a given card. Eight-bit cards have eight-bit words composed of one byte; 16-bit cards have 16-bit words, composed of two bytes; and so forth.

APPENDIX 2

Hot/Cold Insertion Removal

How to retain the data stored into memory cards depends on system conditions and environments etc.

There are many cases of insertion and removal. This table shows whether data is retained and system design requirements for data retention under major system conditions.

		System Side			Data Retention	Remark
		Power	Conditions	CE		
1	Cold Insertion	0V	-	-	YES	Refer to Table 18 for the power-up sequence
2	Cold Removal	0V	-	-	YES	Refer to Table 18 for the power-down sequence
3	Warm Insertion	Vcc	Hi-z	"H"	YES	<u>Conditions for guaranteeing the data retention</u> 1. Place buffer between system and card. 2. Control signal (/CE) from system biased to Vcc and be High-Z. 3. In the card, /CE is turned High by internal pull-up resistor on the /CE before the buffer enable is disabled. 4. System Vcc is forced from card when inserted into system. In such case, system Vcc must maintain the certain voltage. 5. Utilizing 2 CD pins, system enables buffer and then accesses to card after detecting the completion of card insertion. <u>Disturbance from card to system during insertion</u> There is no disturbance to system because there are buffers between system and card and the buffers are enabled after card is ready.
4	Warm Removal		Hi-z	"H"	YES	<u>Conditions for guaranteeing the data retention</u> 1. Place buffer between system and card. 2. Control signal (/CE) from system biased to Vcc and High-Z. 3. In the card, /CE is turned High by internal pull-up resistor on the /CE until the buffer enable signal is enabled. 4. Utilizing 2 CD pins, system stops accessing to card and then disables the buffer before card is completely removed from system. <u>Disturbance from card to system during removal</u> There is no disturbance to system because there are buffers between system and card and the buffers are disabled before card is removed.

		System Side			Data	Remark
		Power	Conditions	CE	Retention	
5	Hot Insertion	Vcc	Active	"L"	NO	<u>Impossible to guarantee data retention</u> Effective pin length differences cannot be used to tell the system that a card is being inserted. A detect means other than pin length must be used for incoming cards.
6	Hot Removal	Vcc	Active	"L"	NO	<u>Impossible to guarantee data retention</u> Utilizing card insertion detect by effective pin length difference (0.28mm) of CD pins or mechanical inter-lock system, the system should prevent this from happening by forcing CE high whenever either CD pin signals card not-present.

