TANDY®

Reference Guide

26-3901

TANDY 600
PROGRAMMERS REFERENCE GUIDE
BIOS AND BIOS SPECIFICATION

CUSTOM MANUFACTURED FOR RADIO SHACK, A DIVISION OF TANDY CORPORATION

16 Bit Hand-Held Operating System
Programmers Reference Guide

Tandy 600
Programmers Reference Guide
BIOS and BIOS Specification

10 9 8 7 6 5 4 3 2 1

TANDY 600
PROGRAMMERS REFERENCE
BIOS AND BIOS SPECIFICATION
GUIDE

Table of Contents

Section 1
PROGRAMMERS REFERENCE

Interrupt Vectors Used by HH O/S

The following interrupts are used by HH O/S:

INT No.	Function
40h	Reserved
41h	Reserved
42h	HH O/S Function Call
43h	DBMS Function Call
44h	Free AMI File Space
45h	File System Moving
46h	Get File Address
47h	Translate Address
48h	Reserved
49h	Reserved
4Ah	Reserved
4Bh	Reserved
4Ch	RS-232C Receiver Hook
4Dh	Interval Timer Hook
4Eh	Power Low/Off Hook
4Fh	Keyboard Queue Hook

Descriptions of Interrupts

INT 42h - HH O/S Function Call

AX:	XXXXXXXX	
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH - Function Code
Other registers are function specific.

Exit: Function specific

See the section on Function Calls for descriptions of the available functions and their entry and exit parameters.

INT 43h - DB Function Call

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH - Function Code
DS:DX - Parameter Block

Exit: AX - Return value or error code

See the section on DBMS Function Calls for descriptions of the available functions and their entry and exit parameters.

INT 44h - Free AMI File Space

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:		

SP:	
BP:	
SI:	
DI:	

IP:	
FLG	

CS:	
DS:	
SS:	
ES:	

Entry: CX - Number paragraphs of memory needed

Exit: AX - Number of paragraphs of memory released

This hook is provided as a mechanism for the HH O/S to request that an application program release memory space from its AMI. Whenever a memory request has occurred during the execution of an O/S function call, and there is not sufficient free memory to satisfy the request, HH O/S will invoke this interrupt with parameters specifying the amount of additional memory required to satisfy the memory request. The application program may then free up memory in its AMI file if possible, and return to HH O/S informing it of how much memory may be recovered from the AMI file.

Some programs may run more efficiently if they can initially allocate a large block of memory to their AMI, and then incrementally release it back to the system as required. This hook is provided to support such applications. It is never necessary for an application program to use this hook vector.

On entry, CX will specify the number of paragraphs of memory that the O/S requires to complete the current memory request. The segment registers DS and ES will contain the same value as on the original entry to the O/S function being executed. The stack registers SS and SP will point to the stack that was in effect on entry to the O/S function.

If the application program is able to release any memory from its AMI file, it should do so. The space freed will always be taken from the end of the file, so that any internal data structures necessary should be moved down to place the free memory at the end of the file. The application program should then return to the O/S by doing an IRET instruction and return in AX the number of paragraphs of memory actually made available to the O/S to recover from the end of the AMI file. If no memory was made available, AX must contain 0.

Any changes to DS, ES, SS, or SP within this hook routine will be remembered by the O/S and those changed registers will be restored before returning to the application when the O/S function call completes. If, for example, the application program's stack were at the end of the AMI, the hook routine could copy the data on the stack down by the requested amount, adjust SP to point to the new stack location, and then perform the IRET. HH O/S would then remember the new location of the application program's stack, and that stack would be in effect when the interrupted function call completes.

HH O/S will have pushed some critical information onto the application program's stack before performing the INT 44h instruction. For this reason, the hook routine must preserve the contents of the stack, although it is allowed to move its location.

HH O/S is not reentrant. For this reason, the INT 44h hook routine may not make any HH O/S function calls. The INT 44h hook routine is allowed to perform any of the HH O/S interrupts that are explicitly listed as being reentrant.

INT 45h - File System Moving Hook

AX:		
BX:		
CX:		
DX:		

Entry: none

SP:	
BP:	
SI:	
DI:	

Exit: none

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

The internal file system under HH O/S is an in-memory system. Each file is contained in a single block of memory up to 64k bytes in size, with the files stored contiguously in memory. For this reason, when one file grows or shrinks, it causes other files in the system to be moved to either make more room for a file that is expanding or to occupy the space left by a file that is contracting.

Because of this dynamic nature of the file system, programs which refer to files through absolute address pointers need to be kept aware of the movement of files in the system. Whenever HH O/S moves a portion of the file system, it will invoke the File System Moving hook. The user provided hook routine can then adjust any internal pointers necessary. Any program which is concerned about the absolute address of an item in the file system can use this hook to keep its file pointers current.

The sequence of operations will be as follows:

1. The affected portions of the file system will be moved.

2. The values which were in the program's segment registers at the time of the current function call will be adjusted as appropriate for the file system movement taking place.

3. The segment portion of the vector address for the File System Moving interrupt will be adjusted as appropriate for the movement taking place.

4. The File System Moving interrupt will be invoked. The application should adjust any pointers necessary. The Translate Address interrupt can be used to aid in making these adjustments. When the hook routine has completed any adjustments it needs to make, it should perform an IRET to return to HH O/S.

5. HH O/S will complete whatever function call caused the file system to move, and then return to the program as usual.

Because HH O/S adjusts the program's segment registers automatically, it will not be necessary for a program to use this hook unless it refers to files using absolute file addresses rather than the read/write function calls. Use of this hook requires careful programming, and should generally be avoided.

NOTE: HH O/S is not reentrant. Hook routines should not perform any O/S or DBMS function calls. The only O/S operations allowed within the hook routine are INT 46h, Get File Address, and INT 47h, Translate Address.

INT 46h - Get File Address

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: BX - File handle of opened file

Exit: AX - Status code or file segment address

HH O/S is not reentrant. For this reason, device drivers and interrupt service routines can't call HH O/S functions. If a device driver, or an interrupt service routine needs to be able to find the absolute address of a file it may do so using this interrupt.

If HH O/S is not able to supply the file address at the time of the call, the Invalid Access Request error will be returned. In this case, the program should repeatedly call INT 46h until this error no longer occurs.

If BX contains 0FFFFh, the address of the AMI for the currently executing program is returned.

INT 47h - Translate Address

AX:		
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: CX - Number of addresses to translate
DS:DX - Pointer to block of addresses to translate

Exit: none

User hook routines which use the File System Moving hook (INT 45h), may use this interrupt to adjust file segment pointers. The contents of DS:DX point to a block of words containing segment addresses which may need adjustment, and CX specifies the number of values in the block. This function will adjust these segment addresses to the corresponding new segment address based upon the current motion of the file system. This function will only return meaningful values when called from within a hook routine which was invoked through INT 45h.

Error Codes

In general, HH O/S indicates the success or failure of an operation by returning a status code in AX. On return from the function call, the carry flag will be clear if no error occurred and the operation succeeded. If the carry flag is set, then the operation failed, and the status code in AX gives the reason for the failure. The list below gives the general meaning of each of the defined status codes. If any other error code is received, it should be treated as a general error indication.

The following error codes are returned by HH O/S functions calls:

1	Un-implemented function
2	File not found
4	Too many open files
5	File access denied
6	Invalid file handle
8	Out of memory
9	Invalid memory block
11	Bad file format
12	Invalid access request
18	No more files
20	File too big
21	Internal file system error
22	Bad file name
23	Exec failure
24	General I/O error
25	Bad application header
26	File checksum error
27	Device not found
28	Invalid time
29	Invalid date

In addition to those listed above, the following error codes can be returned by the data base functions:

40h	Record already opened
41h	No opened record
42h	Record not found
43h	Field not found
44h	Too many fields
45h	No fields defined
46h	Uninitialized data
47h	Bad field data size
48h	Field already exists
49h	Bad sort key specified
4Ah	Query buffer overflow error
4Bh	Bad field type specified
4Ch	Too many records
4Dh	Record too big
4Eh	Invalid Query buffer

HH O/S Function Calls

The following is a list, in numeric order, of all function calls supported by HH O/S. The function code numbers are given in hexadecimal. Function calls 00 - 57 are similar to the same numbered functions in MS-DOS. Although similar in nature, these functions are not necessarily identical to the MS-DOS functions. Function calls numbered higher than 57 are not equivalent to any MS-DOS functions and perform operations unique to the HH O/S environment.

FUNCTION 01h - Read Keyboard and Echo

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 01h (function code)

Exit: AX = Character Typed

Function 01h waits for a character to be typed at the keyboard, then echos the character to the display and returns it in AX.

FUNCTION 02h - Display Characters on Console

AX:	XXXXXXXX	
BX:		
CX:		
DX:		XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 02h (function code)
DL = Character to be displayed

Exit: None

Function 02h displays the character in DL on the LCD display.

FUNCTION 08h - Read Keyboard Without Echo

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 08h (function code)

Exit: AX = Character from keyboard

Function 08h waits for a character to be typed on the keyboard, and then returns it in AX. The typed character is not echoed to the LCD display.

FUNCTION 09h - Display Character String

AX:	XXXXXXXX	
BX:		
CX:		
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 09h (function code)
DS:DX = String to be display

Exit: None

Function 09h will display a string of characters on the LCD display. DX contains the offset (from the segment in DS) of the first character in the string. The string must be terminated with a ⌧ character. The ⌧ character is not displayed.

FUNCTION 0Bh - Check Keyboard Status

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 0Bh (function code)

Exit: AL = 00h - No characters waiting
FFh - Characters waiting

Function 0Bh checks the keyboard type-ahead buffer to see if any characters are waiting to be read. It will return AL = 00 if no characters are waiting, or AL = 0FFh if any characters are waiting.

FUNCTION 0Ch - Flush Keyboard Buffer, Read Keyboard

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 0Ch (function code)
AL = 00h, 01h, 08h
(function to perform after flushing keyboard buffer)

Exit: AX = Depends upon entry value

Function 0Ch will clear any characters waiting from the keyboard type-ahead buffer. After clearing the buffer, the function will then perform one of the following actions depending upon the contents of register AL:

AL = 01h, 08h - The corresponding HH O/S function will be executed.
AL = any other value - Return immediately

FUNCTION 25h - Set Interrupt Vector

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 25h (function code)
AL = Interrupt number
DS:DX = Address of interrupt

Exit: None

Function 25h is used to set an interrupt vector in low memory to point to an interrupt handler routine.

FUNCTION 29h - Parse File Name

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	XXXXXXXXXXXXXXXXXX
DI:	XXXXXXXXXXXXXXXXXX

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 25h (function code)
AL = Wild cards flag
DS:SI = Pointer to input string
DS:DI = Pointer to output buffer

Exit: SI = Pointer to terminating character
AX = Error code if error

Function 29h will parse an ASCIIZ string containing a file specification and return a structure containing the name broken into its component parts. The structure returned will have the following form:

Field	Length	
NAME	8	File name
EXT	3	File extension
DEVICE	8	Device name
NAME_FLAG	1	File name present flag
EXT_FLAG	1	File extension present flag
DEVICE_FLAG	1	Device name present flag

If the input string contained the name of a device rather than a file, the device name will be in the DEVICE field, and the DEVICE_FLAG will be set.

If an error is discovered in the input string, the carry flag will be set, and an error code will be in AX. In all cases, on return, SI will point to the last character used in parsing the name. Normally this will be the terminating 0. If an error occurs, however, SI will point to the character at which the error was detected.

FUNCTION 2Ah - Get Date

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 2Ah (function code)

Exit:
AL = Day of week
CX = Year
DH = Month
DL = Day

Function 2Ah returns the current date set in the operating system. The format of the values returned is:

CX	- Year (1980 - 2099)
DH	- Month (1 - 12; 1 = January, 2 = February, etc.)
DL	- Day (1 - 31)
AL	- Day of week (0 - 6; 0 = Sun., ..., 6 = Sat.)

FUNCTION 2Bh - Set Date

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 2Bh (function code)
CX = Year
DH = Month
DL = Day

Exit: AL = status code

Function 2Bh sets the current date set the operating system. The format of the parameters is:

CX - Year (1980 - 2099)
DH - Month (1 - 12; 1 = January, 2 = February, etc.)
DL - Day (1 - 31)

On return from the function, the carry flag will be reset if the operation succeeded. If the carry flag is set, the operations failed because of an error in the specified data.

FUNCTION 2Ch - Get Time

AX:	XXXXXXXX	
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 2Ch (function code)

Exit: CH = Hour
CL = Minutes
DH = Seconds
DL = Hundredths of seconds

Function 2Ch is used to get the current time from the operating system. The format of the values returned is:

CH - Hour (0-23)
CL - Minutes (0-59)
DH - Seconds (0-59)
DL - Hundredths of seconds (0-99)

FUNCTION 2Dh - Set Time

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 2Dh (function code)
CH = Hours
CL = Minutes
DH = Seconds
DL = Hundredths of seconds

Exit: AL = status code

Function 2Dh is used to set the current time in the operating system. The format of the parameters is:

CH - Hour (0-23)
CL - Minutes (0-59)
DH - Seconds (0-59)
DL - Hundredths of seconds (0-99)

On return, the carry flag will be reset if the operation succeeded. If the carry flag is set, the operation failed because of an error in the specified time.

FUNCTION 2Eh - Set Disk Verify After Write Flag

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 2Eh (function code)
AL = Verify Flag Setting

Exit: none

Function 2Eh controls the setting of the disk verify after write flag. If this flag is set, then any disk write operation will be verified after it is completed. This increases the reliability of disk write operations, but reduces performance considerably.

If AL = 0 on entry, the flag will be reset and no verification will take place.

If AL = 1 on entry, the flag will be set and verification will occur after each write.

FUNCTION 35h - Get Interrupt Vector

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	XXXXXXXXXXXXXXXXXX

Entry: AH = 35h (function code)
AL = Interrupt number

Exit: ES:BX - Pointer to interrupt handler

Function 35h returns the interrupt vector associated with a given interrupt.

FUNCTION 36h - Get Disk Free Space

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 36h (function code)
DL = Drive number

Exit: If carry not set
AX = Sectors per cluster
BX = Number of free clusters
CX = Bytes per sector
DX = Total clusters per drive

If carry set
AX = error code

This function is used to determine space characteristics about the specified disk drive.

FUNCTION 3Ch - Create File

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 3Ch (function code)
CX = File attributes
DS:DX = Pointer to file name

Exit: AX = status code or file handle

Function 3Ch creates a new file, or truncates an existing file to zero length in preparation for writing. If the file did not exist, then a new file will be created and given the specified attributes. If the file already existed, it will be truncated to zero length.

On return from the function, the carry flag will be reset if the operation succeeded. In this case, AX will contain a file handle which is open for read/write access on the file, with the pointer to the current position in the file set to point to the beginning of the file. If the carry flag is set, an error occurred, and the status code is in AX.

Error returns:

FUNCTION 3Dh - Open File

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 3Dh (function code)
AL = access type
DS:DX = File name

Exit: AX = status code or file handle

Function 3Dh will open a file for access. On entry to the function, DX specifies the offset, from the segment in DS, of an ASCIIZ string specifying the name of the file to open. AL specifies the type of access requested. The following access types are defined:

Access	Description
0	Read only
1	Write only
2	Read/Write

On return from the function, the carry flag is reset if the operation succeeded. In this case, AX contains a file handle opened for the requested access and the pointer to the current position is set to the beginning of the file. If the carry flag is set, an error occurred and the status code is in AX.

Error Returns:

FUNCTION 3Eh - Close File

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:	
FLG	

CS:	
DS:	
SS:	
ES:	

Entry: AH = 3Eh (function code)
BX = File handle

Exit: Status code

Function 3Eh closes a file. On entry to the function, BX specifies the handle associated with the file to be closed. On return from the function, the carry flag will be reset if no error occurred. If the carry flag is set, an error occurred and the status code is in AX.

Error Returns:

FUNCTION 3Fh - Read From File

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 3Fh
BX = File handle
CX = Number of bytes to read
DS:DX = Buffer address

Exit: AX = status code or number of bytes read

Function 3Fh reads a block of data from an opened file. On entry to the function, BX contains a file handle for a currently open file, CX contains the maximum number of bytes to read, and DX contains the offset, from the segment in DS, of the buffer to receive the data.

On return from the function, the carry flag is set if no error occurred. In this case, AX contains the actual number of bytes transferred. If this value is less than the number requested, then the end of file has been reached. If the carry flag is set on return, an error has occurred and the status code is in AX.

Error Returns:

FUNCTION 40h - Write to File

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:	
FLG	

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 40h (function code)
BX = File handle
CX = Bytes to write
DS:DX = Buffer address

Exit: AX = status code

Function 40h writes a block of data to a file. On entry, BX contains the file handle of a file opened with write access, CX contains the number of bytes to transfer, and DX contains the offset, from the segment in DS, of the buffer containing the data to write.

On return the carry flag will be reset if no errors occurred. If the Carry flag is set, an error occurred and the status code is in AX.

Error Returns:

FUNCTION 41h - Delete File

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 41h (function code)
DS:DX = File name

Exit: AX = Status code

Function 41h removes a file from the file system. On entry, the function expects DX to contain the offset (from the segment in DS) of an ASCIIZ string specifying the name of the file to delete. On return from the function, the carry flag will be reset if the operation succeeded. If the carry flag is set, an error occurred and the status code identifying the error is in AX.

Error Returns:

FUNCTION 42h - Position File Pointer

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 42h (function code)
AL = Method
BX = File handle
CX:DX = Offset

Exit: AX = status code

Function 42h moves the pointer to the current position in the specified file. On entry to the function, BX specifies the handle of a currently open file, CX:DX specifies the offset to use in moving the file pointer, and AL specifies the method to employ. The following methods are defined:

Method	Description
0	Move the pointer to CX bytes from the beginning of the file
1	Move the pointer to CX bytes from the current position
2	Move the pointer to CX bytes after the current end of the file

The value in CX:DX should be regarded as a 32 bit integer with the most significant 16 bits in CX. Internal files are restricted to being no larger than 64k bytes in size. For internal files if CX is not 0, an error will be returned.

With any of the above methods, it is possible to specify a location past the current end of the file. If this occurs, the file will be extended as required to allow the operation to succeed. An error will result if an attempt is made to position the file pointer beyond the 64k byte limit on the size of an internal file.

Error Returns:

FUNCTION 43h - Get/Set File Attributes

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 43h (function code)
AL = Operation to perform
CX = File attribute to set (if AL = 1)
DS:DX = File name

Exit: AX = Status code
CX = Current file attribute (if AL = 0)

Function 43h allows an application to read or change the attributes of a file. On entry, AL specifies the operation to perform. AL = 0 causes the current attributes to be returned in CX. AL = 1 causes the attributes in CX to be assigned to the file. DX specifies the offset (from the segment in DS) of an ASCIIZ string which gives the name of the file. On return, if the carry flag is reset the operation succeeded, if the carry flag is set, the operation failed and the status code is in AX.

Error Returns:

FUNCTION 44h - I/O Control For Devices

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 44h (function code)
AL = Request Type
BX = File handle
CX = Number of byte to read/write
DS:DX = Data buffer

Exit: AX = status code if carry set?

Request 0,1
DX = Device information

Requests 2,3,4,5
AX = number of bytes transferred

Requests 6,7
AL = device ready status
0 = not ready
FF = ready

Function 44h gets or sets device information associated with an opened handle, or sends/ receives a control string to a device handle.

The following Request Type values are allowed:

0 - Get device information
1 - Set device information
2 - Read from device control channel
3 - Write to device control channel
4 - Read from disk device control channel
5 - Write to disk device control channel
6 - Get device input status
7 - Get device output status

This function can be used to get information about device channels. Calls can be made upon regular files as well, but only request types 0, 6, 7 are defined. All other requests return an invalid function call error.

See the section on IOCTL strings for a description of the control strings which are defined for each device in the system. When writing the control string, the count in CX gives the number of bytes in the string. When reading a control string, the number in CX is the maximum number of bytes to read, and the actual number read will be returned in AX.

The device information which is read/written with functions 0/1 is a 16 bit quantity in which the information is bit encoded. The following bits are defined: (bit 0 is the low bit)

Bit number	Meaning
7	ISDEV - 0 = this channel is a file 1 = this channel is a device

For Devices, these bits are defined:

0	ISCIN - This device is console input
1	ISCOT - This device is console output
5	MODE - 0 = raw data mode 1 = interpreted data mode
14	CTRL - Specifies if the device can accept control strings

For Files, no other bits are defined.

The only bit which can be set is the MODE bit. This bit specifies if some interpretation of the data is to be made by the device driver, or if the data is to be passed directly. This bit is only meaningful for certain devices.

FUNCTION 48h - Allocate Memory Block

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

Entry: BX - Number of paragraphs requested

Exit: AX - Paragraphs address of allocated memory or Error code

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

This function will allocate a block of memory of the requested size in the O/S absolute memory region. A block allocated in this manner is guaranteed not to move from the time it is allocated until it is released.

If the carry flag is cleared on return, AX will contain the paragraph address of the requested memory block. If the carry flag is set, an error occurred, and the error code is in AX.

The allocated memory block is reserved for use by the requestor until it is released through function 49h, or until the allocating program terminates. All memory blocks allocated by a program are freed when it terminates.

FUNCTION 49h - Release Memory Block

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	XXXXXXXXXXXXXXXXXX

Entry: ES - Address of block to free

Exit: AX - Error code if carry set

This function will release back to the system a block of memory allocated through function 48h. The user loads the paragraph address of the block to be freed into ES, and then performs the system call. On return, if the carry flag is not set, the memory block will have been released back to the system. If the carry flag is set, an error occurred, and the error code is in AX.

FUNCTION 4Bh - Execute Program

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH - 4Bh (function code)
DS:DX - Pointer to execution parameter block

Exit: AX - Error code

This function is used to execute an applications program under the HH O/S. The value in DS:DX is the address of an execution parameter block. This block consists of 4 words, with the following meanings:

HIDFLG - 0 = create non-hidden AMI
1 = create hidden AMI
APLPTR - Pointer to applications file name string
AMIPTR - Pointer to AMI file name string
PRMPTR - Pointer to parameter string

1. HIDFLG specifies whether a newly created AMI should be created as a hidden file or a non-hidden file. This only affects AMI's which don't exist at the time of the exec and must be created.

2. APLPTR points to a zero terminated string that specifies the name of an internal file which contains the program to be executed.

3. AMIPTR points to a zero terminated string that specifies the name of the AMI file to use while the application is running.

4. PRMPTR points to a zero terminated string that is passed to the application being executed.

These three pointers give offsets relative to the segment address in DS. The parameter strings must be in the same segment as the parameter block which points to them.

When control is passed to the application, CX:DX will point to a copy of the execution parameter block. The application program may examine any of the strings pointed to by the execution parameter block, but it should not modify them.

If on return, the carry flag is set, an error occurred while trying to execute the specified program, and the error code will be in AX.

FUNCTION 4Ch - Terminate Process

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH - 4Ch (function code)
AL - Termination status code

Exit: Special

This function is used by an applications program to suspend execution. The AMI file for the currently executing application will be closed, and control returned to the program which invoked the one being terminated. The value in AL is a termination status code which the parent process receiving control may interrogate via function 4Dh.

With a positive value specified in AL, this function call should be viewed as a process suspension. Sufficient machine state is preserved in the AMI by HH O/S that the process can be resumed at a later time. Thus, a subsequent call to function 4Bh (execute program) specifying the same program and AMI, will cause control to return to the instruction following the one invoking function 4Ch. This means that with a positive termination code in AL, it is possible for function 4Ch to return to the caller.

When a suspended process is resumed, the parameters in the registers after the return from function 4Ch will be similar to those when a program is initially invoked. CS:DX will point to a copy of the exec parameter block that caused the re-invocation of the process. The resuming process may examine the new parameter block to determine what action to take.

With a negative termination code in AL, this function should be viewed as a process termination. The calling program's AMI file is closed and then deleted. When the AMI file is deleted all process state is lost, and it is not possible to resume. In this case, function 4Ch will not return to the caller.

FUNCTION 4Dh - Retrieve Process Termination Code

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH - 4Dh (function code)

Exit: AX - Process termination code

Function 4Dh will return the termination code set by the last program which terminated via function 4Ch.

FUNCTION 4Eh - Find First Matching File

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	

Entry: AH = 4Eh (function code)
CX = Search attributes
DS:DX = File reference

Exit: AX = Status code
CX:DX = Information buffer

Function 4Eh takes a file reference containing wild-card characters and returns a data block containing information about the first directory entry which matches the file reference. On entry, DX contains the offset (from the segment in DS) of an ASCIIZ string specifying the file reference. CX contains file attributes which restrict the search. On return, if the carry flag is reset the operation succeeded. CX contains the segment address and DX the offset of a buffer containing information about the matched file. The information in the buffer is in the following format:

Location	Description
0	File attributes
1-2	File time
3-4	File date
5-8	File size
9-21	File name (ASCIIZ string)

1. File Attributes - This byte contains the attribute bits for the file. The following bits are defined:
   - 01h - File is read only
   - 02h - File is hidden
   - 04h - File is a system file
   - 20h - Archive bit

2. File Time - This word contains the time of the last file modification. Times are stored in packed binary in the following format:
   HHHHHMMMMMMSSSSS
   H is hours, 0-23
   M is minutes, 0-59
   S is seconds, 0-29 (two second increment)

3. File Date - This word contains the date of the last time the file was modified. Dates are stored in packed binary in the following format:
   YYYYYYYMMMMDDDDD
   Y is year, 0-119 (1980-2099)
   M is month, 1-12
   D is day, 1-31

4. File Size - This field contains 2 words which give the size of the file in bytes. The least significant word is stored first.

5. File Name - This is an ASCIIZ string which gives the name of the file. Disk files do not have the drive specifier given.

If the carry flag is set on return, the operation failed and the status code is in AX.

Error Returns:

FUNCTION 4Fh - Find Next Matching File

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 4Fh (function code)

Exit: AX = Status code
CX:DX = File information buffer

Function 4Fh returns a file information block identifying the next matching file following a call to function 4Eh. It is necessary to call function 4Eh before calling function 4Fh. On return, if the carry flag is reset the operation succeeded and CX:DX contain the address of a file information block in the same format as that returned by function 4Eh. If the carry flag is set on return, the operation failed, and the status code is in AX.

Error Returns:

FUNCTION 54h - Get Setting of Disk Verify Flag

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 54h (function code)

Exit: AL = Verify flag setting

This function returns the current setting of the disk verify after write flag. On exit, AL will contain the current flag setting. If the flag is set (AL = 1), then disk write operations are being verified. If the flag is clear (AL = 0), disk write operations are not being verified.

FUNCTION 56h - Rename File

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	XXXXXXXXXXXXXXXXXX
SS:	
ES:	XXXXXXXXXXXXXXXXXX

Entry: AH = 56h (function code)
DS:DX = Original file name
ES:CX = New file name

Exit: AX = status code

Function 56h will change the name of an existing file. DS:DX contains the address of an ASCIIZ string specifying the file to be renamed. ES:CX contains the address of an ASCIIZ string specifying the new name that the file is to be given.

The file directory is searched for a file matching the first specification. This file specification may contain wild-card characters. The first matching directory entry found will be changed to match the file name given in the second specification. The second specification may also contain wild-card characters. Anywhere a '?' occurs in the second specification, the corresponding characters in the existing directory entry will not be changed.

Following the return from this function, the carry flag will be set and the error code will be in AX if an error occurred.

FUNCTION 57h - Get/Set File Date/Time

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = 57h (function code)
AL = Operation to perform
BX = File handle
CX = Time to set (AL = 1)
DX = Date to set (AL = 1)

Exit: AX = Status code
CX = Current file time (AL = 0)
DX = Current file date (AL = 0)

Function 57h returns or sets the last-write date and time in the directory entry for a file. On entry, AL specifies the operation to be performed. If AL = 0, the current date/time of the file will be returned in CX/DX. If AL = 1, the date/time in CX/DX will be assigned to the file. The time in CX and the date in DX are stored as fields of bits with the following format:

CX - HHHHHMMMMMMSSSSS
H - number of hours (0-23)
M - number of minutes (0-59)
S - number of 2 second increments (0-29)

DX - YYYYYYYMMMMDDDDD
Y - number of years since 1980 (0-119)
M - number of the month (1-12)
D - number of the day (1-31)

On return, if the carry flag is reset the operation succeeded. If the carry flag is set, an error occurred and the status code is in AX.

Error Returns:

FUNCTION D0h - Expand File

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D0h (function code)
AL = Method
BX = File handle
CX = Size of block to insert

Exit: AX = Error Code
CX = Number of bytes added or size of largest possible block that could be added

Function D0h will cause a file to be enlarged by the specified number of bytes. On entry to the function, the user specifies the file handle for a previously opened file in BX, the amount by which the file is to be expanded in CX, and the desired method in AL. The following methods are defined:

Method	Description
0	Expand the file at the beginning
1	Expand the file at the current position
2	Expand the file at the end of file

On return from the function call, the carry flag will be reset if the operation was successful, and DX will contain the number of bytes added to the file. If the carry flag is set, an error occurred, and an error code is in AX. If the call failed because of insufficient memory to carry out the operation, then the file will not have been modified, and the size of the largest block that could be added to the file will be in CX.

Internal files under HH O/S have a file size limit. Files are not allowed to grow larger than the current size limit on that file. The default size limit for an internal file is FFFFh bytes (64k-1). By use of function E1h, it is possible to set smaller size limits. An expand file request which would cause the file to exceed the current size limit on that file will fail. The value returned in CX will be the size of the largest possible block that could be added before the current size limit is exceeded.

If methods 1, or 2 are requested, the current file data will be moved to form a hole of the requested size. For method 2, the byte at the current file position will be moved to become the next byte following the insertion. Following the operation, the current file position will be set to point to the beginning of the inserted block.

Error Returns:

FUNCTION D1h - Reduce File

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D1h (function code)
AL = Method
BX = File handle
CX = Number of bytes to delete

Exit: AX = Error code
CX = Number of bytes deleted

Function D1h is used to remove a block of data from a file and cause the file to shrink by the amount deleted. On entry to the function, the handle of a currently opened file is passed in BX, the number of bytes to delete are specified in CX, and the deletion method is specified in AL. The allowed deletion methods are:

Method	Description
0	Delete the specified number of bytes at the beginning of the file.
1	Delete the specified number of bytes beginning at the current position.
2	Delete the specified number of bytes at the end of the file.

On return from the function, the carry flag will be set if an error occurred, and the error status code will be in AX. If no error occurred, the carry flag will be reset and the actual number of bytes deleted from the file will be in CX. If an attempt was made to delete past the end of the file then the number returned in AX will be smaller than the requested amount. (This is not considered an error by HH O/S.)

Error Returns:

FUNCTION D2h - Return Absolute File Pointer

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:		
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D2h (function code)
BX = file handle

Exit: AX = Absolute file address status code
DX = Size of file in bytes

Function D2h is used to find the absolute memory address where a file is stored. The user specifies the handle for a previously opened file in BX. On return from HH O/S, the carry flag will be set if an error occurred, otherwise the memory segment address of the beginning of the file will be contained in AX. The file is stored in contiguous, ascending memory locations beginning at offset 0 within this segment.

An application program may directly access the data contained in a file by placing the value returned in AX into a segment register and then performing normal memory fetch/store operations relative to that segment. Extreme caution must be taken when accessing a file in this manner, as it is possible to affect memory locations beyond the end of the given file, and thus destroy the integrity of the file system. The current size of the file will be returned in DX when function D2h is performed. At other times, use function D3h to determine the current size of the file and be careful not to go beyond that limit in making accesses to the file. Furthermore, the segment address returned indicates the location of the file in memory at the time of the function call. The file is not guaranteed to remain at this location. As a part of normal file operations, HH O/S will move files within memory. Before moving any portion of the file system, HH O/S will invoke interrupt 45h to indicate what portion of the file system is moving and by how much. File system movement will only occur as a result of a system call that allocates an absolute memory block, or modifies the size of an internal file. Programs which refer to absolute file addresses must either supply a File System Moving hook routine (entered through INT 45h) or user function D2h to get the current address of the file after any system call which may have caused it to move.

It is absolutely essential that the program know the correct address before any absolute access to a file or catastrophic damage can be caused to the internal file system.

Error Returns:

5	- Access denied; The handle specified is not opened as a file, or the file opened may not be accessed in this manner.
6	- Invalid handle; The file handle specified is not currently assigned.

FUNCTION D3h - Return File Size

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D3h (function code)
BX = File handle

Exit: AX = status code
CX:DX = File size in bytes

Function D3h returns the exact number of bytes occupied by the specified file. The file handle of a previously opened file is passed to HH O/S in register BX. If the operation succeeds, the number of bytes in the file is returned in CX:DX, with the most significant word in CX.

If the operation is not successful, the carry flag will be set on return, and AX will contain the status code indicating the cause of the failure.

Error Returns:

5 - Access denied; The specifed handle did not refer to a file.
6 - Invalid file handle; The specified handle is not currently assigned.

FUNCTION D4h - Return Size of Free Memory

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D4h (function code)

Exit: AX = Size of free memory

Function D4h returns the total number of paragraphs of free memory remaining in the system.

FUNCTION D5h - Get File Mark

AX:	XXXXXXXX	
BX:	XXXXXXXX	XXXXXXXX
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D5h (function code)
BX = File handle

Exit: AX = File mark

Function D5h returns a file status mark that can be used to test if a file has been modified. The file mark is a 16 bit unsigned integer that is changed each time that a file is modified.

To use the file mark the following procedure can be used: Read the mark immediately before closing the file and save the value. The next time the file is accessed, read the mark immediately after the file is opened and compare with the previous value. If the values are different, the file has been modified since the last time you closed it.

This function is only allowed on internal files. In order to maintain compatability with MS-DOS, disk files do not have an access control mark.

FUNCTION D6h - Sound Error Tone

AX:	XXXXXXXX	
BX:	XXXXXXXX	XXXXXXXX
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D6h (function code)
BX = Tone channel number

Exit: none

Function D6h is used to sound an error tone. The frequency and duration of the tone are set via function D7h.

FUNCTION D7h - Get or Set Error Tone Values

AX:	XXXXXXXX	
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D7h (function code)
AL = 0 = Get, 1 = Set
BX = Tone channel number
CX = Error tone frequency
DX = Error tone duration (2.5 msec increments)

Exit: none

This function is used to set the frequency and duration of the error tone sounded when function D6h is executed. The frequency is specified in Hertz and is passed in CX. The duration is specified in 2.5 millisecond increments and is passed in DX. Setting either the frequency or the duration to 0 disables the error tone.

If AL is 0 on entry, the current settings are returned. If AL is 1 on entry, the specified values are set.

There are 5 tone channels provided, numbered 0-4. Channels 0 and 1 are reserved for use by Microsoft applications, and should not be used.

FUNCTION D8h - Get Application File Handle

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D8h (function code)

Exit: AX - File handle

This function returns the file handle of the currently executing applications program.

FUNCTION D9h - Get AMI File Handle

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = D9h (function code)

Exit: AX - File handle

This function returns the file handle of the AMI file which is in use by the currently executing application.

FUNCTION DBh - Get Free Memory Address

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = DBh (function code)

Exit: AX - Free memory address

This function returns the paragraph address of the beginning of the system free memory pool.

Function D5h may be used to find the number of free paragraphs of memory in the system. It is important to note that the free memory reported by function D5h is not guaranteed to be located within the free memory pool. HH O/S uses optimization methods for speeding growth of internal files which cause excess space to be allocated to recently accessed files. This reduces the number of times that portions of the file system must be moved when files grow in size.

Applications programs should not attempt to use the free memory pool as scratch work space and so, should not use this function. It is provided to allow other operating systems and system utilities to find the end of memory used by the HH O/S, so that they can coexist with the HH O/S internal file system.

FUNCTION DCh - Initialize Math Pack Area

AX:	XXXXXXXX	
BX:	XXXXXXXX	XXXXXXXX
CX:		
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = DCh (function code)
BX = File Handle
DX = Error trap vector

Exit: AX - Error status code

Function DCh will initialize the math pack area in the specified file. The math pack area contains working storage used by the math pack. See the math pack section for a description of the user accessible data items within the math pack area.

On entry, BX contains the file handle of the opened internal file to be initialized, DX contains the offset of an error handler routine which will receive control when the math pack detects an error.

On exit, if the carry flag is reset, the math pack area was successfully initialized. If the carry flag is set, an error occurred and the error code is in AX.

FUNCTION DDh - Set Timer Channel

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:		

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = DDh (function code)
AL = Channel mode
0 = One shot timer
1 = Repeating timer
2 = Start elapsed time
3 = Query elapsed time
4 = Set auto timeout delay
5 = Get auto timeout status
6 = Enable auto timeout
7 = Disable auto timeout
BX = Timer channel number
(ignored for modes 4-7)
CX = Time delay
(ignored for modes 2-3)

Exit: For mode 3
CX = Elapsed time
For mode 5
CX = Current auto timeout status

Function DDh accesses the event timer portion of the HH O/S event flag facility. HH O/S supports 8 programmable timers which can be set to signal an event flag at the end of the specified time interval.

On entry, BX specifies the timer channel number to be set, (0-7), CX specifies the number of seconds for the timeout interval, (0-65535), and AL specifies the timer mode. Timer mode 0 is a one-shot timer. It will wait the specified number of seconds and set the flag. Timer mode 1 is a repeating timer, it will set the flag every CX seconds until the channel is reset.

Timer modes 2-3 are used for counting elapsed time. Mode 2 will place the timer channel in elapsed time mode, and set the elapsed time to 0. Timer mode 3 will return the current value of the elapsed time counter in register CX.

Specifying a time interval in CX of 0 seconds and mode 0 or 1 will disable the channel.

Timer channels 0-1 are reserved for use by Microsoft, and should not be used.

FUNCTION DEh - Set Alarm Date/Time

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = DEh (function code)
AL = Alarm type code
0 = Restart alarm
1 = Annunciator alarm
BX = Year (1980-2099)
CH = Month (1-12)
CL = Day (1-31)
DH = Hours (0-23)
DL = Minutes (0-59)

Exit: CY not set - none
CY set - Error code in AX

This function is used to set the next alarm Date/Time for the internal alarm system.

When a restart alarm occurs, if the machine is powered off, it will be turned on and the application program set via function XXh will be run. If the machine is powered on when the restart alarm occurs, it will be ignored.

When an annunciator alarm occurs, the system alarm event flag will be set. This will cause the alarm annunciator to flash on the bottom line of the system display. See function E0h for a description of the system event flags.

FUNCTION DFh - Get Alarm Date/Time

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = DFh (function code)
AL = Alarm type code
0 = Restart alarm
1 = Annunciator alarm

Exit: CY not set
AL = Day of week (0-6)
BX = Year (1980-2099)
CH = Month (1-12)
CL = Day (1-31)
DH = Hours (0-23)
DL = Minutes (0-59)

CY set
AX = Error code

This function is used to get the current setting of the next alarm in the internal alarm system.

See function DEh for a description of restart and annunciator alarms.

FUNCTION E0h - Get/Set Event Flag State

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:		

SP:	
BP:	
SI:	
DI:	

IP:	
FLG	

CS:	
DS:	
SS:	
ES:	

Entry: AH = E0h (function code)
AL = Operation to perform
0 = Get current state
1 = Set new state
BX = Event number
CX = New value (for AL=1)

Exit: AX = Current state
(for AL = 0)

This function is used to check on the state of an event flag, or set a new state for an event flag in the alarm system.

If AL = 0, the current state of the specified event flag will be returned in AX. If AL = 1, the specified event flag will be set to the state specified in CX (0 or 1).

The following event flag numbers are defined:

0	= keyboard activity flag
1	= touch panel activity flag
2	= telephone ring detect flag
3	= break key detect flag
4	= software alarm flag
5 - 7	= reserved
8 - 15	= timer channels 0 - 7
16 - 31	= comm activity flags for channels 0-15

The keyboard activity flag (event 0) is set whenever a character is sent from the keyboard.

The touch panel activity flag (event 1) is set whenever a touch panel state change occurs. (Only occurs on machines with touch panel hardware.)

The telephone ring detect flag (event 2) is set whenever a ring detect interrupt occurs, and will be reset 30 seconds later. Whenever the ring detect flag is set, the ring detect annunciator on the bottom line of the system display will flash. An application program may turn the ring detect annunciator on or off by setting or resetting this event flag.

The break key detect flag (event 3) is set whenever the user presses the break key.

The alarm annunciator flag (event 4) is set when an annunciator type alarm set via function DEh occurs. Whenever this event flag is set, the alarm annunciator on the bottom line of the system display will flash. An application program may turn on or off this annunciator by setting or resetting this event flag.

The timer channel flags (events 8-15) whenever the corresponding interval timer reaches its set interval.

The com activity flags (events 16-31) will be set whenever a character is received on the corresponding com channel.

With the exception of the telephone ring detect event flag, HH O/S does not reset the event flags. It is the responsibility of the program monitoring the event to reset the flag.

FUNCTION E1h - Set File Size Limit

AX:	XXXXXXXX	XXXXXXXX
BX:	XXXXXXXX	XXXXXXXX
CX:	XXXXXXXX	XXXXXXXX
DX:	XXXXXXXX	XXXXXXXX

SP:	
BP:	
SI:	
DI:	

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Entry: AH = E1h (function code)
BX = File Handle
CX:DX = Size limit to set

Exit: AX = Error code

Function E1h allows an upper limit to be placed on the size of a file. The file handle of an opened file is specified in BX. The size limit to be set is specified in CX:DX with the high order word in CX. After this call, any system call which would cause the file to expand beyond the specified limit will result in a 'File Too Big' error to be returned.

If the size limit specified in CX:DX is smaller than the current size of the file, the function call will fail and a 'File Too Big' will be returned.

This function may only be applied to internal files, and the file size limit may not be set larger than FFFFh. CX must always be 0 when calling this function.

FUNCTION E2h - Set Function Key Definition

AX:		
BX:		
CX:		
DX:		

Entry:

SP:	
BP:	
SI:	
DI:	

Exit:

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

FUNCTION E3h - Reopen File

Entry:

AX:		
BX:		
CX:		
DX:		

SP:	
BP:	
SI:	
DI:	

Exit:

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Error returns:

FUNCTION E4h - Memory Management for Install

AX:		
BX:		
CX:		
DX:		

Entry:

SP:	
BP:	
SI:	
DI:	

Exit:

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Error returns:

FUNCTION E5h - Touch Panel Support

Entry:

AX:	
BX:	
CX:	
DX:	

Exit:

SP:	
BP:	
SI:	
DI:	

IP:	
FLG	

CS:	
DS:	
SS:	
ES:	

Error returns:

FUNCTION E6h - Lock File Open

AX:		
BX:		
CX:		
DX:		

Entry:

SP:	
BP:	
SI:	
DI:	

Exit:

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Error returns:

FUNCTION E7h - Get/Set Alarm Application

AX:		
BX:		
CX:		
DX:		

Entry:

SP:	
BP:	
SI:	
DI:	

Exit:

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

Error returns:

HH O/S Database Function Calls

The following function calls are used to access the database primitive functions contained in the HH O/S.

All of the Database Functions receive parameters in the following manner:

AX:	XXXXXXXX	XXXXXXXX
BX:		
CX:		
DX:	XXXXXXXX	XXXXXXXX

Entry: AH = function code
DS:DX = Pointer to parameter block

SP:	
BP:	
SI:	
DI:	

Exit: AX = Return value or Error code

IP:		
FLG		

CS:	
DS:	
SS:	
ES:	

The function code for the Database Function to perform is specified in AH. DS:DX contains a pointer to the parameter block which contains the entry parameters for the specified function. The format of the parameter block for each function is specified in the description of the function.

On return, if the carry flag is set, an error occurred and the error code will be in AX. If the carry flag is not set, the value in AX is the return value of the function. Functions which do not specify any return values will return AX = 0.

The following gives a list of all database functions in numeric order:

00 - Create Database File
01 - Open Database File
02 - Close Database File
03 - Delete Database File
04 - Create Record
05 - Open Record
06 - Close Record
07 - Delete Record
08 - Get Field ID
09 - Get Field Data Type
0A - Get Field Name
0B - Create Field
0C - Delete Field
0D - Get Field From Open Record
0E - Put Field To Open Record
0F - Get Field From Specified Record
10 - Rename Field
11 - Find Matching Record
12 - Get Number of Records
13 - Sort File
14 - Get Current Sort Order
15 - Begin Query Definition
16 - End Query Definition
17 - Open Query Record
18 - Close Query Record
19 - Put Query Field
1A - Move Record
1B - Check Query Field
1C - Change Field Data Type
1D - Get Record Size
1E - Compare Records
1F - Put Field To Specified Record

Function 00h - Create Database File

Parameter Block:

00:	FILE_NAME	Pointer to file name

Return Value:

AX:	STATUS	File handle or error code

The FILENAME parameter contains the offset from the segment in DS of a string specifying the name of the database file to be created. The database file name string is an ASCIIZ string of up to 13 characters (counting terminating 0). There may not be a disk drive specifier in the name, as database files must be in memory to be accessed.

A database file of the specified name will be created and initialized. If a database file with this name already exists, it will be deleted and recreated as an empty file.

On return, if the create was successful, AX will contain a file handle which should be used for future accesses to the file.

Error Returns:

Function 01h - Open Database File

Parameter Block:

00:	FILE_NAME	Pointer to file name
01:	ACCESS_TYPE	0 = Read 1 = Write 2 = Read/Write

Return Value:

AX:	STATUS	File handle or error code

The FILENAME parameter contains the offset from the segment in DS of a string specifying the name of the database file to be opened. The database file name string is an ASCIIZ string of up to 13 characters (including terminating 0). The file name string should not contain a disk drive specification as database files must be resident in memory to be accessed.

The ACCESSTYPE parameter specifies the kind of access desired for the file. An attempt to open a file with the Read Only attribute for Write or Read/Write will fail.

On return, if the open was successful, AX will contain a file handle which should be used for future accesses to the file.

Error Returns:

Function 02h - Close Database File

Parameter Block:

00:	FILE_HANDLE	File handle of file to close

Return Value:

AX:	STATUS	Error code

The FILEHANDLE parameter specifies the file handle for a previously opened database file. The file will be closed and the file handle released. The file is guaranteed to be closed after this function returns, even if an error occurs.

Error Returns:

Function 03h - Delete Database File

Parameter Block:

00:	FILE_NAME	Pointer to file name

Return Value:

AX:	STATUS	Error code

The FILENAME parameter contains the offset from the segment in DS of a string specifying the name of the database file to be deleted.

The specified file will be deleted from the file system. If the specified file has the Read Only attribute set, it will not be deleted and an error will be returned.

Error Returns:

Function 04h - Create Record

Parameter Block:

00:	FILE_HANDLE	File handle
01:	Record_ID	Record identifier

Return Value:

AX:	STATUS	Error code

The FILEHANDLE parameter contains the file handle of a database file which was opened for Write or Read/Write access.

The RECORDID parameter specifies the ID the record being created is to have. If the ID specified is too large to be a legal ID, the highest legal ID will be used instead. It is not possible to create a database with holes in the record numbering.

A new record will be created and all of the fields initialized to null values. The created record will be open for read/write access.

There can only be one opened record at a time. This operation will fail if there is already a record open.

Error Returns:

Function 05h - Open Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	RECORD_ID	Record Identifier
02:	ACCESS_TYPE	0 = Read 1 = Write 2 = Read/Write

Return Value:

AX:	STATUS	Error code

The FILEHANDLE parameter contains the file handle for a previously opened database file.

The RECORDID parameter specifies the record ID of the record to open.

The ACCESSTYPE parameter specifies the type of access to be performed upon the record. The database file must have been opened with an access type that allows the requested access type.

There can only be one opened record at a time. This operation will fail if there is already a record open.

Error Returns:

Function 06h - Close Record

Parameter Block:

00:	FILE_HANDLE	File Identifier

Return Value:

AX:	STATUS	Record ID of closed record or error status

The FILEHANDLE parameter specifies the file handle for a previously opened database file.

The currently open record will be closed, and its record ID returned in AX.

Error Returns:

Function 07h - Delete Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	RECORD_ID	Record Identifier

Return Value:

AX:	STATUS	Error code

The FILEHANDLE parameter contains the file handle for an open database file. The RECORDID contains the record ID of the record to be deleted.

Deleting a record can cause the record ID's of other records to change. Any record ID's which the application program is storing may become invalid when a delete operation is performed.

Deleting a record is not allowed if there is an open record in the database file. This function will return an error if there is an open record.

Error Returns:

Function 08h - Get Field ID

Parameter Block:

00:	FILE_HANDLE	File to access
01:	FIELD_NAME	Pointer to field name

Return Value:

AX:	STATUS	Field Identifier or Error code

The FILEHANDLE parameter contains the file handle for an open database file. The FIELDNAME parameter contains the offset from the segment in DS of the character string specifying the name of the field. The field name string is an ASCIIZ string up to 16 characters in length.

The value returned in AX will be the Field Identifier to use when referring to this field.

Error Returns:

Function 09h - Get Field Type

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	FIELD_ID	Field Identifier

Return Value:

AX:	STATUS	Field Data Type or Error Code

The FILEHANDLE parameter specifies the file handle for a previously opened database file. The FIELDID parameter specifies the field to be accessed.

The Field Data Type of the specified field will be returned in AX. The following are the defined field data types:

1 - Character
2 - Numeric
3 - Date

Error Returns:

Function 0Ah - Get Field Name

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	FIELD_ID	Field Identifier
02:	NAME_BUFFER	Pointer to buffer to receive field name

Return Value:

AX:	STATUS	Field Identifier or Error code

The FILEHANDLE parameter specifies the file handle of a previously opened database file. The FIELDID parameter specifies the field to be accessed. The NAMEBUFFER parameter specifies the offset from the segment in DS of a buffer to contain the returned name.

The name of the specified field will be returned as an ASCIIZ string with a maximum length of 17 bytes (16 characters plus terminator).

Error Returns:

Function 0Bh - Create Field

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	FIELD_NAME	Pointer to Field Name
02:	FIELD_TYPE	Field Data Type

Return Value:

AX:	STATUS	Field Identifier or Error code

The FILEHANDLE parameter specifies the file handle of a previously opened database file. The FIELDNAME parameter specifies the offset from the segment in DS of an ASCIIZ string which specifies the name of the field to be created. The field name may not be longer than 16 characters. If it is, it will be truncated to 16 characters. The FIELDTYPE parameter specifies the field data type of the field to be created. The following field data types are defined:

1 - Character
2 - Numeric
3 - Date

Creation of a field is not allowed if there is an open record. This operation will fail if there is a record open. A maximum of 64 fields may be defined.

Error Returns:

Function 0Ch - Delete Field

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	FIELD_NAME	Pointer to Field Name

Return Value:

AX:	STATUS	Error code

The FILEHANDLE parameter specifies the file handle of a previously opened database file. The FIELDNAME parameter specifies the offset from the segment in DS of an ASCIIZ string which gives the name of the field to be deleted.

Deletion of a field is not allowed if there is an open record. The operation will fail and return an error code in this case.

The data for the specified field will be removed from all records in the specified file.

Error Returns:

Function 0Dh - Get Field From Open Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	FIELD_ID	Field Identifier
02:	TRANSFER_LENGTH	Number of bytes to read
03:	BUFFER	Pointer to buffer to receive data

Return Value:

AX:	STATUS	Number of bytes transferred or Error Code

The FILEHANDLE parameter specifies the file handle of a previously opened database file. The FIELDID parameter specifies the field ID of the field to access. The TRANSFERLENGTH parameter specifies the number of bytes to be read from the field. The BUFFER parameter specifies the offset from the segment in DS of the buffer to receive the transferred data.

On return, AX will contain the actual number of bytes placed in the buffer.

The bytes transferred will always be taken starting at the beginning of the field data. It is not possible to read a field in a series of partial reads.

Error Returns:

Function 0Eh - Put Field into Open Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	FIELD_ID	Field Identifier
02:	TRANSFER_LENGTH	Number of bytes to write
03:	BUFFER	Pointer to buffer containing data to write

Return Value:

AX:	STATUS	Error Code

The FILEHANDLE parameter specifies the file handle of a previously opened database file. The FIELDID parameter specifies the field ID of the field to access. The TRANSFERLENGTH parameter specifies the number of bytes to be written to the field. The BUFFER parameter specifies the offset from the segment in DS of the buffer containing the data to write.

If the operation succeeds, AX will contain a zero on return. Otherwise, the carry flag will be set and AX will contain an error code.

The bytes transferred will always be written starting at the beginning of the field. It is not possible to write a field in a series of partial writes.

Error Returns:

Function 0Fh - Get Field From Specified Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	RECORD_ID	Record Identifier
02:	FIELD_ID	Field Identifier
03:	TRANSFER_LENGTH	Number of bytes to read
04:	BUFFER	Pointer to buffer to receive data

Return Value:

AX:	STATUS	Number of bytes transferred or Error Code

The FILEHANDLE parameter specifies the file handle of a previously opened database file. The RECORDID parameter specifies the record from which the data is to be read. The FIELDID parameter specifies the field ID of the field to access. The TRANSFERLENGTH parameter specifies the number of bytes to be read from the field. The BUFFER parameter specifies the offset from the segment in DS of the buffer to receive the transferred data.

On return, AX will contain the actual number of bytes placed in the buffer.

The bytes transferred will always be taken starting at the beginning of the field data. It is not possible to read a field in a series of partial reads.

Error Returns:

Function 10h - Rename Field

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	OLD_NAME	Pointer to current name
02:	NEW_NAME	Pointer to new name

Return Value:

AX:	STATUS	Field Identifier or Error code

The FILEHANDLE parameter specifies the file handle of a previously opened database file. The OLDNAME parameter specifies the offset from the segment in DS of an ASCIIZ string containing the name of the field. The NEWNAME parameter specifies the offset from the segment in DS of an ASCIIZ string containing the new name to be given to the field.

Error Returns:

Function 11h - Find Matching Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	STARTING_RECORD	Record ID to start search
02:	SEARCH_DIRECTION	Search Direction (0 = forward)(1 = backward)
03:	SEARCH_KEY	Offset of query key buffer
04:	SEARCH_KEY_SEGMENT	Segment of query key buffer

Return Value:

AX:	STATUS	Record ID of next record or Error Code

This function will perform a query upon the specified database. The SEARCHKEYSEGMENT: SEARCHKEY specify the segment and offset to a buffer containing the query key to be used to guide the search. Query keys are created by the use of functions 15h - 19h.

The search will begin with the record specified by the STARTINGRECORD parameter, and proceed in the direction specified by SEARCHDIRECTION. The record ID of the first record found that matches the query key will be returned. If the record specified by STARTINGRECORD matches, its record ID will be returned.

See the section on Database Queries for a further description of the query process.

Error Returns:

Function 12h - Get Number of Records in File

Parameter Block:

00:	FILE_HANDLE	File Identifier

Return Value:

AX:	STATUS	Number of records or Error Code

This function returns the count of records existing in the file specified by the FILEHANDLE parameter.

Error Returns:

Function 13h - Specify Sort Key

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	NUMBER_OF_KEYS	Number of sort key fields
02:	SORT_KEY	Pointer to sort key

Return Value:

AX:	STATUS	Error code

This function is used to define a sort key for the specified database. A sort key is made up of a series of sort key fields which have the following form:

DIRECTION	byte	0 = Descending, 1 = Ascending
FIELD_ID	byte	Field to sort

As many sort key fields may be specified as there are fields in the database file. The sort precedence of the keys is according to the order in which they are specified.

Error Returns:

Function 14h - Get Current Sort Key Definition

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	NUMBER_OF_KEYS	Number of sort key fields to return
02:	SORT_KEY	Buffer to receive sort key

Return Value:

AX:	STATUS	Number of sort key fields or Error code

This function will return all or a portion of the current sort key in effect for the specified database file. The NUMBEROFKEYS field specifies the number of key fields to be returned. The key fields returned will always be the first N defined fields. The return value in AX specifies the number of key fields actually transferred. This will always be less than or equal to the number requested. If the NUMBEROFKEYS parameter is 0, no data will be transferred, but the return value in AX will be the number of fields in the sort key.

Error Returns:

Function 15h - Begin Query Definition

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	MODE	Query definition mode
02:	BUFFER_LENGTH	Size of query definition buffer
03:	BUFFER	Offset of query key buffer
04:	BUFFER_SEGMENT	Segment of query key buffer

Return Value:

AX:	STATUS	Error Code

The FILEHANDLE parameter specifies the opened database file for which the query is to be defined.

The MODE parameter may be either 0 or 1. In mode 0, the query key is not built, but the size of the query key is determined. In this case, the buffer must be at least 16 bytes long. In mode 1, the query key is actually built. In this case, the buffer must be large enough to contain the query key being built. The size required may be determined by making a complete pass through the query definition in mode 0. When the query definition is closed, the total size of the query is returned. This much space can then be allocated for the buffer, and then another pass made in mode 1 to define the query key.

The BUFFERLENGTH parameter specifies the size of the buffer.

The BUFFER and BUFFERSEGMENT parameters specify the segment and offset to the buffer which contains the query key being defined.

Error Returns:

Function 16h - Close Query Definition

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	BUFFER	Offset of query buffer
02:	BUFFER_SEGMENT	Segment of query key buffer

Return Value:

AX:	STATUS	Size of Query key

The FILEHANDLE parameter contains the file handle of an opened database file for which a query if being constructed.

The BUFFER and BUFFERSEGMENT parameters specify the segment and offset to the buffer which contains the query key being defined.

The return value in AX is the total size of the query key in bytes.

Error Returns:

Function 17h - Open Query Definition Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	BUFFER	Offset of buffer
02:	BUFFER_SEGMENT	Segment of query key buffer

Return Value:

AX:	STATUS	Error code

The FILEHANDLE parameter contains the file handle of an opened database file for which a query key is being constructed.

The BUFFER and BUFFERSEGMENT parameters specify the segment and offset to the buffer which contains the query key being defined.

Error Returns:

Function 18h - Close Query Definition Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	BUFFER	Offset of buffer
02:	BUFFER_SEGMENT	Segment of query key buffer

Return Value:

AX:	STATUS	Error code

The FILEHANDLE parameter contains the file handle of an opened database file for which a query key is being constructed.

The BUFFER and BUFFERSEGMENT parameters specify the segment and offset to the buffer which contains the query key being defined.

Error Returns:

Function 19h - Put Query Definition Field

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	BUFFER	Offset of query buffer
02:	BUFFER_SEGMENT	Segment of query buffer
03:	FIELD	Pointer to field descriptor

Return Value:

AX:	STATUS	Error Code

The FILEHANDLE parameter contains the file handle of an opened database file for which a query key is being constructed.

The BUFFER and BUFFERSEGMENT parameters specify the segment and offset to the buffer which contains the query key being defined.

The FIELD parameter contains the offset from the segment in DS of the field descriptor for the query field. The field descriptor has the following format:

POINTER	word	Pointer to literal string. Not used for field to field comparisons.
TYPE	byte	Type of comparison to perform.
OPERAND1	byte	Field ID of left operand field.
OPERAND2	byte	Count of characters for a literal value, or Field ID of second operand for a field value.

TYPE - This byte specifies the type of comparison to perform. The low 4 bits contain the relational operator to apply, the high 4 bits specify modes of the comparison.

Relational Operators:

=	0
<>	1
<	2
>	3
<=	4
>=	5

Mode bits

bit 7 - 0 = compare the field to a literal value. The length of the literal is in OPERAND2 and POINTER is a pointer to the literal.
1 = compare the field to another field. The field ID of the second field is in OPERAND2.

bit 6 - 0 = Case is significant in string comparisons
1 = Case is not significant

bit 5 - 0 = No wild cards are present
1 = Wild cards may be present

Error Returns:

Function 1Ah - Move Record

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	OLD_ID	Original Record ID
02:	NEW_ID	New Record ID

Return Value:

AX:	STATUS	Error code

The FILEHANDLE parameter specifies the file handle of an opened database file.

The OLDID parameter specifies the record ID of an existing record which is to be moved.

The NEWID parameter specifies the record ID the record is to have. If this is greater than the greatest valid ID, the greatest valid ID will be used instead.

Error Returns:

Function 1Bh - Check Query Definition Field

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	FIELD	Pointer to field descriptor

Return Value:

AX:	STATUS

This function checks a query definition field descriptor for errors.

The FILEHANDLE parameter contains the file handle of an opened database file.

The FIELD parameter contains the offset from the segment address in DS of a query field descriptor. This descriptor is formatted as described in function 19h.

Error Returns:

Function 1Ch - Change Field Type

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	FIELD	Field Id of Field to change
02:	TYPE	New Field Type

Return Value:

AX:	STATUS

This function will change the data type associated with an existing field in the specified file. Any data stored in the changed field is not converted to the new type. The data conversion must be performed by the program performing the change field type function call.

The procedure to use is: Perform the Change Field Type function call. For each record in the file, read the data from the changed field, convert it to the new data type, and write it back to the field.

Error Returns:

Function 1Dh - Get Record Size

Parameter Block:

00:	FILE_HANDLE	File Identifier
01:	RECORD	Record number

Return Value:

AX:	STATUS

This function will return the total number of bytes occupied by the specified record in the specified file. The value is returned in AX.

Error Returns:

Function 1Eh - Compare Two Records

Parameter Block:

00:	FILE_HANDLE_1	File Identifier for first file
01:	RECORD_ID_1	Record Number in first file
02:	FIELD_LIST_1	Pointer to field list for first file
03:	FILE_HANDLE_2	File Identifier for second file
04:	RECORD_ID_2	Record Number in second file
05:	FIELD_LIST_2	Pointer to field list for second file
06:	COUNT	Number of fields in field lists

Return Value:

AX:	STATUS

This function will perform a comparison of two records. The records to be compared are specified by giving a file handle and record ID for each record. It is legal for the file handles to be the same, comparing two records in the same file, or different, comparing records from two different files.

The field list parameters point to two arrays of field ID's which identify the fields to be compared and the order in which to compare them. Both arrays must be the same length and contain as many field ID's as specified in COUNT.

On return from the function, if the carry flag is set, an error occured and the error code is in AX. Otherwise, AX will contain:

0	- Record_1 < Record_2
1	- Record_1 = Record_2
2	- Record_1 > Record_2

Error Returns:

Function 1Fh - Put Field to Specified Record

Parameter Block:

00:	FILE_HANDLE_1	File Identifier
01:	RECORD_ID	Record Identifier
02:	FIELD_ID	Field Identifier
03:	TRANSFER_LENGTH	Number of bytes to write
04:	BUFFER	Pointer to buffer to write

Return Value:

AX:	STATUS	Number of bytes transferred or Error Code

The FILEHANDLE parameter specifies the file handle of a previously opened database file. The RECORDID parameter specifies the record to which the data is to be written. The FIELDID parameter specifies the field ID of the field to access. The TRANSFERLENGTH parameter specifies the number of bytes to be written to the field. The BUFFER parameter specifies the offset from the segment in DS of the buffer containing the data to be written.

On return, AX will contain the actual number of bytes written to the field.

The bytes transferred will always be taken starting at the beginning of the buffer. It is not possible to write a field in a series of partial writes.

Error Returns:

Application Programs Under HH O/S

Introduction

Applications programs in the HH O/S system reside in internal or memory files in the HH O/S file system. Unlike a disk operating system which copies the program from disk into memory in order to run, a program under HH O/S is always in memory, and runs directly from the location it occupies in the file system. This achieves space reductions, as there are not two copies of the program file in the system, but complicates the rules that the program must obey in order to operate successfully.

An important point to note is that because the HH O/S internal file system is an in-memory file system, and because there is no memory protection hardware available on the 8086 family microprocessors, the internal file system is quite fragile. It is very easy for an ill behaved program to damage the internal file system.

In addition to the file containing the program code to be executed, an application also needs memory for a work space to contain variables, and working data. This work space is also contained in a file within the internal file system. Application work space files are called AMI (Application Memory Image) files.

An AMI file contains the working variables, stack space, and user data associated with an invocation of an application program. Because the entire state of the running program is contained within this one file, it is possible for the application to suspend execution, and then resume where it left off at a later time. This allows there to be multiple invocations of a given application in existence in the system at the same time, each AMI file corresponding to an application is a separate invocation of that application.

It is the operational philosophy of the Microsoft Hand-Held applications software that a user can exit an application at any time, execute a second application, and return to the first application at a later time with exactly the same context as when he exited. This is accomplished through the use of the AMI. The AMI file is a program 'state' file which contains the complete operating state of an invocation of the application program. When the application terminates, its complete state is preserved in the AMI file. Because of this, when the application is re-executed it is possible to return to the exact place where it left off. A single application will have as many 'states' as there are AMI files for that application. For this reason, no variable data should be stored in the application code file.

Because the program's work space (AMI file) is actually a file within the file system, it is necessary to understand something of how the file system works. A file is a contiguous block of memory up to 64k bytes in size that has a name and a size associated with it. Because all of the files in the system are stored contiguously, when a file grows or shrinks, all of the files above it in the system must move up or down. This means that the file system is a very dynamic, and the actual memory addresses where a file is located can be changing frequently.

Each application has its code (be it native 8086 code or Microsoft QCODE) stored in an application load file. The system manager program recognizes files having extensions of the form: .!DD (where DD is a two digit number), as an application program file. Applications will appear ordered (in the left hand column of the system manager display) by the number 'DD'.

Programs are always invoked through the HH O/S execute program function (OSEXEC). Normally, the call to OSEXEC will be made by the Microsoft System Manager program. The OSEXEC call specifies the name of the application program to run, the name of the AMI file to use, and an optional parameter string to be passed to the application. If the specified AMI file exists, and is recognized as being a valid AMI for the specified application, execution of the application will resume at the point where the application previously terminated. If the specified AMI file does not exist, a new one will be created, and control will be passed to an entry point specified in the application header at the beginning of the application code file.

The application code file and the AMI file for the currently executing program are open while the program is running, and the handles for these files may be obtained by using the Get Application File Handle, and Get AMI File Handle system calls. By using the Expand file and Reduce file system calls on the file handle for the current AMI, a program may expand or reduce the size of its work space. Care should be taken when using the Reduce file call that the reduction does not reduce the size of the file below that portion used for statically allocated data and the program stack.

Programs terminate execution through the HH O/S terminate program function (OSTERM). When a program terminates, control will be returned to the parent who invoked the program. Normally, this will be the Microsoft System Manager program. The OSEXEC function in HH O/S maintains a stack of program invocations to allow this call-return type operation to be successful. This stack is large enough to allow 4 levels of depth.

A program may terminate for one of two reasons. It may complete its task, and terminate of its own accord without user intervention, or it may be told to terminate by the user. The Microsoft Hand-Held software allows the user to switch easily between applications programs. It does this through the use of functions keys which have special meanings. In order to support this philosophy of easily switching between applications, an application should be ready to terminate anytime it is reading keyboard input. When the user requests the application to terminate, the HH O/S will return a special <QUIT> character code to the application. When the application sees this <QUIT> character, it must place itself into some orderly state and then perform an OSTERM function. When an application terminates, it should release any space that it has obtained for its data region that it is not actually using. This frees the memory so that other applications can make use of it.

The <QUIT> character indicates that the user has requested the program to terminate, with the presumption being that some other program will be run. In addition to the <QUIT> character an application program may see a <SUSPEND> character. Like the <QUIT>, the <SUSPEND> character indicates that the running program must place itself into an orderly state, and then perform an OSTERM function. However, the <SUSPEND> character additionally means that the user is executing a 'pop-up' function and that when that 'pop-up' terminates, control will return to the program being <SUSPEND>ed. (An example of a 'pop-up' function is the system calculator generally accessed through CTL-F2.)

The following is a list of the system special character codes, and what they mean:

1. <QUIT> - 6700h - Program termination request. The program should place itself into an orderly state and perform an OSTERM function.

2. <SUSPEND> - C100h - Program suspend request. The program should place itself into an orderly state and perform an OSTERM function.

3. <REDRAW> - C200h - The application program should redraw its screen. Programs should redraw their screen on return from an OSTERM function, and additionally when they see this character.

Program Header

Applications programs contain a header used by the HH O/S execute program function to verify that the file is executable and to provide the necessary information for the program to be executed. The format of this header is as follows:

Length	Name
2	APLCHK
2	AMICOD
1	LDRID
1	AMIVIS
1	AMITYP
3	AMIEXT
2	APLIP
2	APLSP
2	APLSIZ
2	DATPOS
2	DATLOC
2	DATLEN
Total: 22	

The following fields are optional. Their presence is determined by bit 40h in the LDRID byte field. If this bit is set, these fields must be present in the application header.

2	APLSIZ
2	reserved
2	reserved
2	reserved
2	reserved
Total: 32	

Each field is now described in detail.

1. APLCHK - Application Check.

   This is a one work 'magic' number (5F10 hex) used to determine whether this is an application load file. Its presence ensures that a simple misnaming of a file to have an application style extension will not cause a catastrophe.

   It should be noted that attempting to execute a badly formed application load file is quite likely to cause the handheld software to crash causing in memory file system damage.

2. AMICOD - AMI (application work space) Code.

   The number in this field is used to identify an AMI (application work space) file as belonging to a particular application program. Each application in the system should have a unique number in this field. Numbers in the range 0-8FFFh are reserved for use by Microsoft. Numbers in the range 8000h-FFFFh are for use by ISV's for their own applications. When an application program terminates, the O/S places this code number into a field in the header of the AMI file. When the application program is reactivated, the AMI codes in the application header and the AMI header are checked to ensure that the application actually owns the AMI.

   If an application program were to be run using an AMI file owned by another application, the result whould be a system crash that would almost certainly destroy the machine's internal file system. This mechanism is a safeguard to help prevent that from occuring.

3. LDRID - Loader Id Code.

   This is used by the handheld operating system EXEC call to determine the type of application being run, and mode information about the exec being performed. The LDRID byte is divided into two 4 bit fields. The low order 4 bits identify what type of program code the application file contains. The following values are defined:

0,1,3	8086 native code application
2,4	Microsoft qCode application
5	Application indirection file

   The high order 4 bits contain bit encoded mode information about the exec. The following bits are defined:

10h	- reserved
20h	- reserved
40h	- Application header contains optional fields
80h	- Use system workspace file ¤¤ SYS006.SYS as the workspace for this program.

1. Bit 40h - If this bit is set, the application header contains the optional fields listed below, and their values will be checked during the exec process.

2. Bit 80h - If this bit is set, the system workspace file ¤¤SYS006.SYS will be used as the workspace file. If another workspace file name is specified, it will be ignored and ¤¤SYS006.SYS will be used. There are certain restrictions on programs which use the system workspace file. They are allowed to use the expand and reduce file system calls to perform dynamic memory management, but must never reduce the size of the file below its size on entry to the program. Programs which use the system workspace file must also not quit with a negative termination code in AL (delete workspace file on termination). Although the workspace file is not deleted on exit, each invocation of the program will enter through the initial invocation entry point. Thus, the terminate process system call will never return.

   In order to use the system workspace file, the initial AMI size used by the program must not be larger than 4096 bytes.

4. AMIVIS - AMI Visability Type.

   Used by the system manager to determine when the application is to be displayed in the left margin, possible values are.

ASCII code	Display Style
V	Always visible
S	Only visible when owned files are present
I	Always invisible (together with owned files)

5. AMITYP - Owned File Type

   Used by system manager program to tell if owned files are AMI's or data files, possible values are:

ASCII code	Owned file type
.	Owned files are AMIs
-	Owned files are data

6. AMIEXT - Owned Extension.

   Used by system manager program to associate owned file with application, the 3 bytes contain the ASCII characters that make up the extension. This field should only contain upper case ASCII characters that are legal in a file name, and should be blank padded on the right if less than 3 characters long.

7. APLIP - Application Initial Pc.

   The address to jump to within the load file on application start up.

8. APLSP - Application Initial Sp.

   Place to set SP within application's AMI before starting to execute the load file.

9. APLSIZ - Application Initial AMI size.

   This word specifies the size in bytes that the AMI file should be when an application is being run on a newly created AMI.

10. DATPOS - Data Image Position in Load File.

    The position within the application load file that an image of the AMIs initialized data is to be found.

11. DATLOC - Data Image Position in AMI.

    The position within the AMI that the image of the initialized data is to be copied.

12. DATLEN - Data Image length.

    The length of the initialized data image.

The following define the optional fields in the application header.

1. APLSIZ - Program File Size

   This word contains the size in bytes of the program code file. This word is checked against the actual length of the file at exec time. If the current file size is less than APLSIZ, the program code file is considered to be damaged and will not be run.

2. Reserved

   Reserved for future use. Must be initialized to -1 (0FFFFh).

Device Drivers Under HH O/S

The HH O/S operating system recognizes a fixed set of devices. These devices are defined in the Microsoft Hand Held BIOS Specification, and ordinarily, each device is mapped into a set of routines in the BIOS which handle the function calls associated with that device. The mapping between logical device and the routines that handle the functions for that device is accomplished by each device being accessed through a different BIOS interrupt. By modifying the interrupt vector associated with a given BIOS interrupt, it is possible to map the device to a different set of routines to handle the device functions. The HH O/S installable Device Driver mechanism defines a regular scheme for accomplishing this alternate mapping.

A device driver is an 8086 native code program that resides in a file in the HH O/S internal file system. The device driver file contains a header that identifies it as a device driver to the operating system, and provides the control information necessary to install it and remove it. The installation process involves placing the device driver code in a protected place in memory, calling an initialization entry point, and then modifying the interrupt vector for the device to point to the function call entry point of the device driver. Removing the device driver from the system involves calling a deactivation entry point in the driver, deallocating the protected memory location that the driver occupied, and then modifying the interrupt vector to point back to the original BIOS entry point.

General Rules For Device Drivers

Device drivers are 8086 machine code programs. They may occupy only a single segment of memory which may be up to 64k bytes in size. During normal operation, the device driver resides in a region of memory that allows it to stay at a fixed address. However, when a device driver is removed from the system, it may be necessary to move other drivers to different locations in order to recover the memory occupied by the driver being removed. For this reason, the device driver must be coded in such a way that it can be relocated easily. As long as the driver does not do FAR calls or jumps to locations within itself, and does not refer to absolute segment addresses within itself for data access, it should be address independent. The exception to this address independence would be interrupt service routines which are contained in the device driver. Before HH O/S relocates the driver, it will call the initialization entry point telling the driver where it is being relocated to. This gives the driver a chance to perform any operations necessary before the code is moved.

Device drivers are called by HH O/S as a normal part of its operation. Because HH O/S in not reentrant, device drivers may not call HH O/S for any reason.

Device Driver Header

The device driver header is used by the HH O/S to verify that a file is a device driver, which device the driver belongs to, and additional control information needed to install and remove it. The format of the header is as follows:

Field Name	Field Length	Data Type
DVRCHK	2 bytes	word
DEVNAM	8 bytes	byte
DVCTRL	2 bytes	word
DVFUNC	2 bytes	word
VECNUM	1 byte	byte
VECTOR	4 bytes	dword
LENGTH	2 bytes	word

Total length of header = 21 bytes

1. DVRCHK - This field is used by the HH O/S to verify that the file is a device driver. It must contain the value 5C10h.

2. DEVNAM - This field specifies the name of the device that this driver belongs to. The name must be in upper case, left justified in the field, and padded with blanks to the right. The following names are legal and represent the complete set of devices for which a driver may be defined.

   KYBD - Keyboard input. BIOS INT 50h
   LCD - LCD Text output. BIOS INT 51h
   GRPH - LCD Graphics output. BIOS INT 52h
   CASS - Cassette. BIOS INT 53h
   PRN - Printer. BIOS INT 56h
   DSK - Disk support. BIOS INT 57h
   COM - Communication/Modem support. BIOS INT 59h
   BAR - Bar Code Reader Support. BIOS INT 5Ah
   TCH - Touch Panel Support. BIOS INT 5Bh

3. DVCTRL - This field defines the control entry point of the device driver. This entry point will be called when the device driver is installed, when the HH O/S finds it necessary to move the device driver to a new location in memory, and when the device driver is being removed.

4. DVFUNC - This is the device function entry point. This offset will be placed in the interrupt vector when the device is installed.

5. VECNUM - This 1 byte field contains the interrupt vector number of the vector used by the device.

6. VECTOR - This field is not initialized by the driver. It is used by the HH O/S to store the vector to the BIOS driver for the device while the new driver is installed. When the driver is removed, the values stored here are returned to the interrupt vector to restore the device to the default BIOS driver for the device.

7. LENGTH - This field specifies the amount of memory in paragraphs that the device driver requires. This much memory will be allocated for the driver when it is installed. This value may be larger than the actual size of the file to allow space for buffers.

Descriptions of Device Driver Entry Points

The following entry points must be provided in a device driver file:

DVCTRL - Device Driver Control

This entry point is used for controlling the device driver file during installation and removal. This entry point will be called with a FAR call. When initialization is complete, it should return with a FAR return. The following functions are required:

Function 0 - Initialize device driver.

INPUT:	AH	- Function Code
OUTPUT:	none	
ERRORS:	CY set if error occured. BIOS error code in AL	

This function is called when the device driver is first installed to initialize and activate the driver. At the time that this function is called, all memory has been allocated, and the driver located at the correct address for execution.

If the carry flag is set on return from function call 0, it is assumed that the initialization failed and the driver installation will be aborted.

Function 1 - Prepare driver for movement

INPUT:	AH	- Function code
	DX	- Address driver will be moved to
OUTPUT:	none	
ERRORS:	none	(must succeed)

When the HH O/S finds it necessary to move a device driver, this function will be executed to inform the driver of the fact. The device driver must place itself in a state so that it is safe for the code to be moved to a new location.

Function 2 - Driver Movement Complete

INPUT:	AH	- Function code
OUTPUT:	none	
ERRORS:	none	(must succeed)

When the HH O/S has completed moving a device driver file, this function will be called to tell the driver to reactivate itself. The driver will be at its new location when this call is made.

Function 3 - Deactivate Device Driver

INPUT:	AH	- Function code
OUTPUT:	none	
ERRORS:	none	(must succeed)

When the HH O/S is removing a device driver, this function will be called to tell the device driver to deactivate itself. Following the return from this function, it is assumed that the system is in a safe state for the device driver to be removed and the original BIOS driver restored.

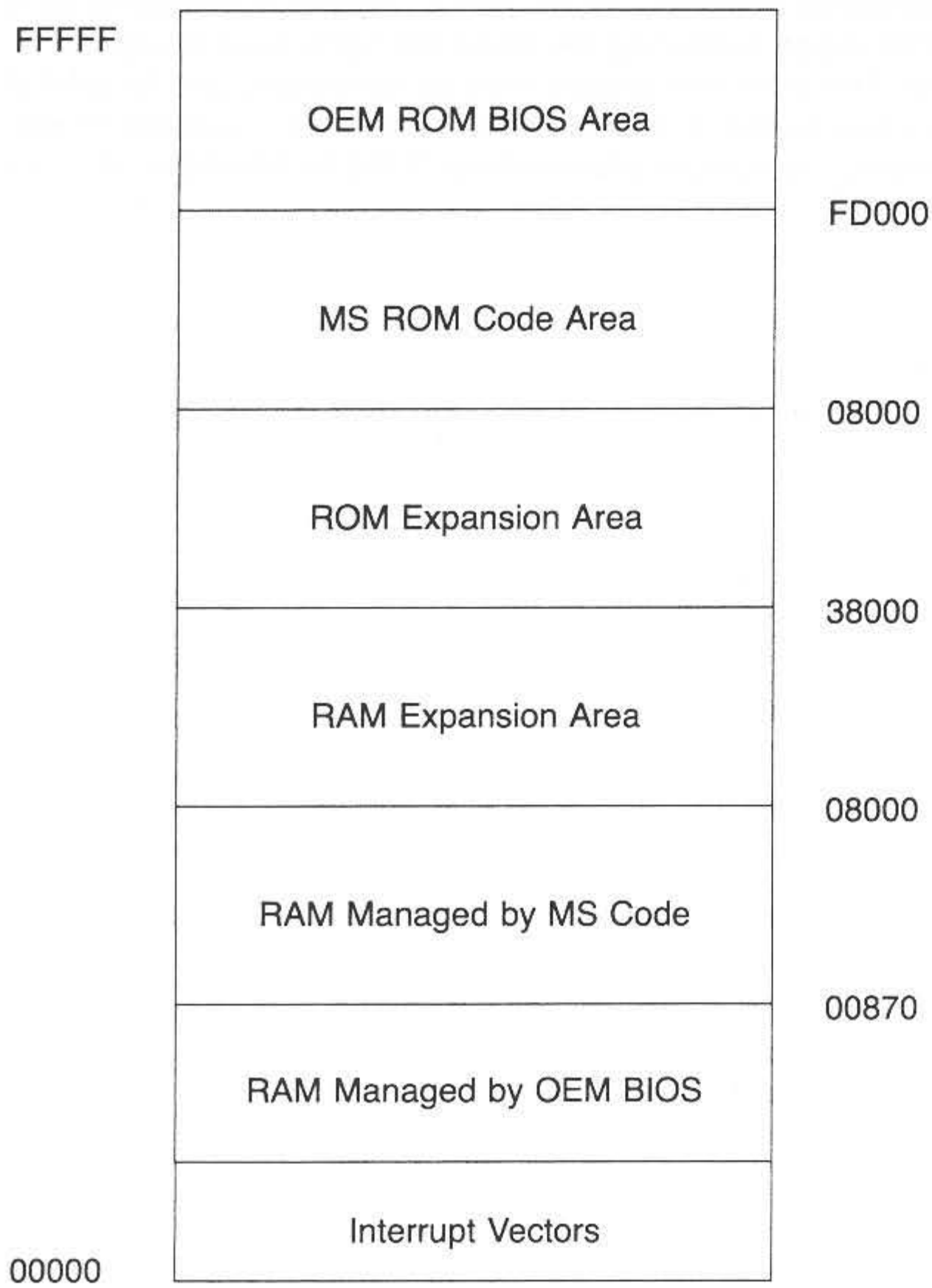
DVFUNC - Device Function Entry Point

This entry point is the BIOS function call entry point. This routine will be entered through the interrupt associated with the device. The entry and exit parameters depend upon the device and the function being requested.

Section 2
BIOS
SPECIFICATION

Table of Contents

Macroscopic Memory Map



The interrupt vectors in the range 40h - 4Fh are reserved for use by the Microsoft ROM code, any interrupts outside of this range may be used by the OEM.

The RAM Managed by the OEM BIOS is shown as being at lower addresses than the RAM Managed by MS Code in the above diagram. This is for illustrative purposes only. When control is transferred to the MS Code, the BIOS supplies parameters advising it of how much RAM it has and where this RAM is located. The MS code will not use RAM outside of this region (except for MS reserved interrupt vectors). The RAM Managed by the OEM BIOS can actually be anywhere that is convenient. The only restriction is that the RAM assigned to the MS code must be contiguous.

Similarly, the areas shown as MS ROM code area and ROM expansion area are merely illustrative. The Hand-Held O/S contained in the MS ROM code is capable of managing up to 5 separate ROM regions. The ROM region containing the Hand-Held O/S must always be resident, and at the same address. The other four regions may be removable, and located at any address. When the BIOS transfers control to the MS code at power up it supplies to the HH O/S a pointer to a table describing the number and locations of the ROMs installed in the machine at that time.

Calling the BIOS

All BIOS calls are made via software interrupts. This presents a clean, easily documented interface to the BIOS. If the user documentation describes only this interface to the BIOS and a note discouraging the use of undocumented ROM routines, it will be possible for the ROM to be easily revised without introducing incompatibilities.

Except as otherwise noted, registers that are not used to return values must be preserved by the BIOS.

For any BIOS calls where no provisions are made for the BIOS to return error codes on "out-of-range" parameters, it may be assumed that those BIOS functions will never be called with an illegal parameter.

Except for MNI interrupt entry and exit, the BIOS functions should switch to a BIOS stack on entry and restore the user's stack on exit. The BIOS may not assume that there is sufficient space on the calling program's stack for it to function.

Software Interrupt Map

The BIOS software interrupt assignments are shown below. During cold start initialization, the BIOS must initialize the interrupt vectors to point to the appropriate support routine within the BIOS.

Interrupt Number	Function
40h-4Ah	Reserved, initialized to point to a dummy IRET in the BIOS
48h	Touch Panel Hook (IRET) — Unsupported Software Interrupt
4Ah	Ring Detect Hook, initialized to point to a dummy IRET in the BIOS
4Bh	Touch Panel Hook, initialized to point to a dummy IRET in the BIOS
4Ch	RS-232 Receiver Queue Hook, initialized to point to a dummy IRET in the BIOS
4Dh	Interval Timer Hook, initialized to point to a dummy IRET in the BIOS
4Eh	Power Low/Off Hook, initialized to point to a dummy IRET in the BIOS
4Fh	Keyboard Queue Hook, initialized to point to a dummy IRET in the BIOS
50h	Keyboard Support
51h	LCD Text Support
52h	LCD Graphics Support
53h	Cassette Support — Unsupported Software Interrupt
54h	Calendar/Clock/Alarm Support
55h	Sound Support
56h	Printer Support
57h	Disk Support
58h	Power Off
59h	Communications/Modem Support
5Ah	Bar Code Reader Support — Unsupported Software Interrupt
5Bh	Touch Pannel Support — Unsupported Software Interrupt
5Ch-6Fh	Reserved, initialized to point to a dummy IRET in the BIOS

70h	Keyboard Queue Hook, for European key layout (Initialized as IRET) This hook can be invoked right before invocation of "INT 4FH — Keyboard Queue Hook." Key layout can be modified for European key layouts by using this hook.
71h	BIOS Special Extended Function Support. This support is provided for installable printer drivers.

Installable Drivers

The Hand-Held software will support a limited form of installable device drivers. These device drivers will be small machine code programs which will be contained in files within the file system of the Hand-Held computer. The user will be able to install a particular file as a driver for a particular device. When installed, the driver program will be placed in a protected memory location by the Hand-Held O/S, and the appropriate interrupt vectors will be changed to point to the installed driver. The driver program must then provide support for all specified BIOS functions for the given device. As an example, an installed driver may be used to perform special character code translation for a different printer than the one supported in the BIOS printer driver code. It would be possible to configure the Hand-Held computer to support different printers by providing an installable driver for each printer.

This mechanism does not provide for the inclusion of new devices not already described in this document. It allows the replacement of the BIOS code to handle a specific type of device with new code to handle that same type of device.

Device Request/Release Logic

In order to aid the Hand-Held operating system in performing device management to avoid contention between devices, the BIOS must support a device request/release system.

The logic of this system is intended to be quite simple. A single flag for each device is all that is required to implement it. When a device request function call is received, the 'in use' flag for that device is tested. If the device is not in use, the flag is set to indicate that it is in use, and the function returns with an error code of 0 which indicates a successful request. If the 'in use' flag is set, then the function returns a 'Device In Use' error. When a device release function call is received, the 'in use' flag for the device is reset. It is not an error to release a device which is not 'in use'. There is no record of who 'owns' the device, the flag merely indicates that the device is in use.

Standard BIOS Error Codes

In the interrupt function definitions, references are made to standard BIOS error codes as return values for a number of the interrupt functions. The following are the defined error codes and their meanings:

Error Code	Meaning
0	No Error Occurred
1	Receiver Queue Overflow
2	Parity Error
3	Device Not Available
4	Device In Use
5	Device Timeout
6	Receiver Overrun
7	Framing Error
8	Carrier Detect Absent (modem only)
9	Out of Paper (printer only)
10	General I/O Error
11	Operator Aborted the Operation by Typing BR
12	Checksum Error
13	Data Buffer Overflow
14	Write Protect
15	Device Not Ready
16	Seek Error
17	Data Not Found
18	Write Fault
19	Program Not Found
0FFh	General Error (not one of the above errors)

Character Code Definitions

There are two cases with which the BIOS must be concerned when dealing with character codes. The first case is that of which character codes are to be recognized for display on the LCD display and what graphic shape is to be generated for each of these characters. The second case concerns the set of characters which the user may type on the keyboard, and what character codes are to be returned.

The character set used for output to the LCD display will be the same as is used in the IBM PC. This character set contains 256 displayable characters using a single 8 bit code for each character defined.

The character set used for input is a superset of the IBM PC input character set. A single byte is used to return the codes 1 through 255 decimal, while an extended code is used for special functions. The extended character codes are two byte codes with the first byte containing a binary 0, and the second byte containing the extended character code. The BIOS routine that returns keyboard characters will return single byte characters in the AL register. A two byte extended code will be returned with AL = 0 and the extended code in AH.

Keys which don't appear on the keyboard may be simulated by combinations of other keys.

The following table defines the extended function codes. Those shown with a Y are required, those shown with an N are not required. The Cursor Left + and Cursor Right + characters are not shown in the IBM Keyboard Mapping List. These operations will be mapped to function keys in the IBM keyboard, and do not require special characters.

Extended Code		Function	IBM Keyboard Mapping	Non-IBM Keyboard Mapping
Y	0	Break	Break	Break
N	3	ASCII NUL	ASCII NUL	ASCII NUL
Y	15	Back Tab	Back Tab	Shift TAB
Y	16-25	ALT Q, W, E, R ALT T, Y, U, I, ALT O, P	ALT Q, W, E, R ALT T, Y, U, I, ALT O, P	ALT Q, W, E, R ALT T, Y, U, I, ALT O, P
Y	30-38	ALT A, S, D, F ALT G, H, J, K ALT L	ALT A, S, D, F ALT G, H, J, K ALT L	ALT A, S, D, F ALT G, H, J, K ALT L
Y	44-50	ALT Z, X, C, V ALT B, N, M	ALT Z, X, C, V ALT B, N, M	ALT Z, X, C, V ALT B, N, M
Y	59-68	F1-F10	F1-F10	F1-F10
Y	71	Cursor Left + +	Home	CTRL Cursor Left
Y	72	Cursor Up	Cursor Up	Cursor Up
Y	73	Cursor Up +	Page Up	Shift Cursor Up
Y	75	Cursor Left	Cursor Left	Cursor Left
Y	77	Cursor Right	Cursor Right	Cursor Right
Y	79	Cursor Right + +	End	CTRL Cursor Right
Y	80	Cursor Down	Cursor Down	Cursor Down
Y	81	Cursor Down +	Page Down	Shift Cursor Down
N	82	Insert	INS	Insert
Y	83	Delete	DEL	Delete (note 1)
Y	84-93	Shift F1-F10	Shift F1-F10	Shift F1-F10
Y	94-103	CTRL F1-F10	CTRL F1-F10	CTRL F1-F10
Y	104-113	ALT F1-F10	ALT F1-F10	ALT F1-F10

N	114		CTRL PrtSc	
N	115		CTRL Cursor Left	
N	116		CTRL Cursor Right	
Y	117	Bottom of Window	CTRL End	ALT Cursor Down
Y	118	Cursor Down + +	CTRL Page Down	CTRL Cursor Down
Y	119	Top of Window	CTRL Home	ALT Cursor Up
N	120-131	ALT 1,2,3,4,5,6	ALT 1,2,3,4,5,6	ALT 1,2,3,4,5,6
		ALT 7,8,9,0,-,=	ALT 7,8,9,0,-,=	ALT 7,8,9,0,-,=
Y	132	Cursor Up + +	CTRL Page Up	CTRL Cursor Up
Y	133	Cursor Left +	(not required)	Shift Cursor Left
Y	134	Cursor Right +	(not required)	Shift Cursor Right
Y	135	Shift Car. Ret.	Shift Car. Ret.	Shift Car. Ret.
Y	136	CTRL Car. Ret.	CTRL Car. Ret.	CTRL Car. Ret.
Y	137	ALT Car. Ret.	ALT Car. Ret.	ALT Car. Ret.
Y	138	Shift ESC	Shift ESC	Shift ESC
Y	139	CTRL ESC	CTRL ESC	CTRL ESC
Y	140	ALT ESC	ALT ESC	ALT ESC
Y	141	BREAK +	Shift BREAK	SHIFT BREAK or CTRL BREAK
Y	142	Pause	CTRL NumLock	Pause
Y	143	Print Screen	Shift PrtSc	Print Screen
N	144	Insert +	Shift INS	Shift Insert
N	145	Delete +	Shift DEL	Shift Delete
N	146	LABEL	F1	Label (note 2)
N	147	Shift LABEL	Shift F1	Shift Label
N	148	CTRL LABEL	CTRL F1	CTRL Label

1) The Delete key should not be confused with the Backspace character (08h), or the Rubout character (7Fh), which is also sometimes called delete.

2) Microsoft software will respond to F1, Shift F1 and CTRL F1 as a request to display function key labels. It will additionally respond to LABEL, Shift LABEL, and CTRL LABEL if the OEM provides this key.

Keyboard Support

The BIOS scans the keyboard at interrupt level (triggered by either a keyboard interrupt or a timer interrupt depending on the particular hardware), and is responsible for handling auto-repeat (if supported), key encoding, and queuing to support type-ahead.

Just prior to placing the key stroke data into the key stroke queue, the keyboard scan routine must execute an INT 4Fh. This permits code to be "hooked" into the keyboard scan routine to take action on specific keys. For example, BASIC will use this hook for detecting BREAK and for implementing key trapping. The keyboard driver has two BREAK key flags: one used in BIOS, and the other will be returned to OS. Both flags are set simultaneously by depressing the BREAK key; the former is cleared right before the BIOS loop including self BREAK-check, and the latter is returned and cleared when the OS invokes Function 3.

The format of this hook is:

INT 4Fh - Keyboard Queue Hook

Entry: AX = Keyboard character
FLAGS.C=0

Exit: FLAGS.C=0 - the key stroke data (returned in AX) should be put into the queue
FLAGS.C=1 - the key stroke data should be discarded (i.e., do not put the data into the queue)
AX = Key stroke data to store in queue

Notes:

1. See the section on Character Code Definitions for a discussion of character codes and their meanings.

2. To simplify queue management, it may be desirable to store all key strokes as two byte codes, even though the extended character codes are the only true two byte codes.

3. The recommended size of the key stroke queue is 64 bytes. This will accommodate 32 two byte key stroke entries if the above recommendation is followed.

4. If the key stroke queue is full, additional key strokes should be discarded. (This must be done after the INT 4Fh call, not before it, since the INT 4Fh routine must see the keys even if the queue is full.

5. It may prove useful for the code hooked into INT 4Fh to be able to modify the key stroke data. Therefore, the BIOS keyboard scan routine must queue the contents of AX as returned by the INT 4Fh routine, not the original key stroke data it passed to the INT 4Fh routine.

The following interrupts are used by applications to obtain keyboard status and data from the BIOS:

INT 50h, Function 0 - Check Key Stroke Queue

Entry: AH = 0 (function code)

Exit: AL = number of key strokes in the keyboard queue (0 through N, where N is the maximum number of keystrokes the queue can hold. Two byte keystrokes should be counted as a single keystroke.)
FLAGS.Z = 1 if AL = 0 (queue is empty)
FLAGS.Z = 0 if AL.NE.0 (queue is not empty)

INT 50h, Function 1 - Fetch Key Stroke

Entry: AH = 1 (function code)

Exit: FLAGZ.Z = 1 if no key stroke is available
(In this case AH and AL are undefined)

FLAGZ.Z = 0 if key stroke is available and: AX = Key stroke data
(See the section on Character Code Definitions for details format of the key stroke data.)

INT 50h, Function 2 - Clear Keyboard Queue

Entry: AH = 2 (function code)

Exit: No registers specified

This routine discards any data in the key stroke queue.

INT 50h, Function 3 - Check Break Key

Entry: AH = 3 (function code)

Exit: FLAGS.Z = 1 if the BREAK key is not pressed.

FLAGS.Z = 0 if the BREAK key is pressed.

This function is intended as a fast check to see if the break key is pressed.

INT 50h, Function 4 - Check Key Stroke

Entry: AH = 4 (function code)

Exit: FLAGS.Z=1 if no key stroke is available
(in this case AX is undefined)

FLAGS.Z=0 if key stroke is available

AX = keystroke data

This function returns the next character in the keystroke queue, but does not remove it from the queue. It is intended to allow an application to check what character will be returned by the next call to function 1. The difference between this function and function 1 is that the character returned is not removed from the keystroke queue.

INT 50h, Function 5 - Stuff Key Stroke

Entry: AH=5 (function code)
BX=Key stroke

Exit: AL=Queue full flag
0=character was placed in the queue
0FFh=character was not placed in queue because the queue was full

This function is intended to allow an application to place a character code into the keystroke queue. The character code in BX will be a standard BIOS character code (single byte if BL<>0, two byte if BL=0). The specified character should be placed at the end of the queue. If the keyboard queue is full, the character is discarded, and the queue full flag returned.

Screen (LCD) Text Support

Since BASIC is the only application needing to read characters from the screen, this capability will be built into BASIC rather than the BIOS. Therefore, the BIOS does not need to keep a text buffer containing the ASCII codes of the characters on the screen.

The BIOS may need to maintain a reverse video bit for each character position on the screen. A bit is set to 1 if the corresponding character position on the screen is in reverse video. A 0 indicates that the corresponding character is in normal video. The need for the reverse video bits depends upon the BIOS implementation, and is merely a suggested method. The high level applications code does not use these bits in any way.

Screen positions are given by a column and row address, with 0 being the leftmost column and the top row of the display.

The following interrupts support the display of text on the screen:

INT 51h, Function 0 - Return text size of screen

Entry: AH = 0 (function code)

Exit: AL = Screen redraw flag
DH = highest numbered column
(number of columns on screen)-1
DL = highest numbered row
(number of rows on screen)-1

This function returns various attributes about the text screen. The screen redraw flag in AL specifies whether the contents of the text screen are preserved when a power off/power on cycle occurs. If the contents of the screen are preserved across a power cycle, then AL should be 0FFh. If the screen contents are not preserved, AL should be 0. If AL = 0 is returned, then the application program being run will be told that it needs to redraw its screen.

INT 51h, Function 1 - Clear Screen

Entry: AH = 1 (function code)

Exit: No registers specified

This routine clears the screen, positions the cursor to the upper left corner, and clears all reverse video bits. This function does not affect the on/off status of the cursor or affect the current on/off status of the reverse video. This function also ignores the current video mode (reverse/normal) and clears the screen so that all pixels are off.

INT 51h, Function 2 - Set Cursor Position

Entry: AH = 2 (function code)
DH = cursor column
DL = cursor row

Exit: No registers specified

INT 51h, Function 3 - Report Cursor Position

Entry: AH = 3 (function code)

Exit: DH = cursor column
DL = cursor row

INT 51h, Function 4 - Cursor on

Entry: AH = 4 (function code)

Exit: No registers specified

INT 51h, Function 5 - Cursor off

Entry: AH = 5 (function code)

Exit: No registers specified

INT 51h, Function 6 - Enable Line Wrap

Entry: AH = 6 (function code)

Exit: No registers specified

INT 51h, Function 7 - Disable Line Wrap

Entry: AH = 7 (function code)

Exit: No registers specified

With line wrap enabled, displaying a character in the rightmost column of the display causes the cursor to advance to the leftmost column of the next line of the display. When this occurs on the bottom line, the display is scrolled up one line and the cursor is positioned to the leftmost column of the new blank line on the bottom of the screen.

With line wrap disabled, the cursor is not advanced after displaying a character in the rightmost column of the display. Consequently, a scroll is not automatically performed after displaying a character in the rightmost column of the bottom line of the display.

The BIOS intialization sequence should enable line wrap.

INT 51h, Function 8 - Display Character at Cursor Position and Advance Cursor

Entry: AH = 8 (function code)
AL = character code

Exit: DH = cursor column (after advance)
DL = cursor row (after advance)

INT 51h, Function 9 - Position Cursor, Display Character, and Advance Cursor

Entry: AH = 9 (function code)
AL = character code
DH = cursor column
DL = cursor row

Exit: DH = cursor column (after advance)
DL = cursor row (after advance)

This function has the combined effect of calling the three separate functions to position the cursor, display a character, and report the cursor position. It should not be coded as a series of software interrupts however, since the entire purpose of combining these functions is to improve the speed of the combined operations.

INT 51h, Function A - Scroll

Entry: AH = A (function code)
AL = Blanking flag
0 = don't blank the source region
FF = blank the source region
BH = destination column
BL = destination row
CH = column count
CL = row count
DH = source column
DL = source row

Exit: No registers specified

The most common actions performed by the scroll function are to roll the display and move characters within a given line. It may be desirable to optimize these cases, perhaps taking advantage of hardware scrolling capabilities. Diagonal scrolling must also be supported, and it is also possible for the source and destination regions to overlap. This function will never be called with parameters that would cause the area being scrolled to overlap the edge of the screen.

The blanking flag indicates whether blanking should be performed on the source region. If the blanking flag is set, then it is necessary to blank the source region. Only that portion of the source region which is not overlapped by the destination should be blanked. When blanking, the current video mode (reverse/normal) is ignored, and the blanking will reset all pixels in the blanked region.

Note that if they are used, the reverse video bits must also be scrolled when the display is scrolled.

INT 51h, Function B - Reverse Video of Specified Block

Entry: AH = B (function code)
AL = Video state to set
0 = Normal Video
1 = Reverse Video
CX = Number of characters

Exit: No registers specified

Starting at the current cursor position, the specified number of characters are set to the specified video state. If the number of characters specified goes past the end of a line, output should wrap to the beginning of the next line. If the specified number of characters would cause the output to go beyond the end of the last line on the screen, the extra characters are ignored. In other words, wrap to the next line on any line but the last. On the last line, ignore any extra characters.

After this function is completed, the cursor should be returned to its original location. This function does not affect the setting of the current reverse video state. This function will never cause the screen to scroll.

INT 51h, Function C - Enable Reverse Video

Entry: AH = C (function code)

Exit: No registers specified

INT 51h, Function D - Disable Reverse Video

Entry: AH = D (function code)

Exit: No registers specified

When reverse video is disabled, BIOS puts the characters on the display in normal video, clearing the corresponding reverse video bits. When reverse video is enabled, the characters are put on the display in reverse video with the corresponding reverse video bits set to 1.

The BIOS initialization sequence should disable reverse video.

INT 51h, Function E - Blank Screen Region

Entry: AH = E (function code)
CH = number of columns
CL = number of rows
DH = start column number
DL = start row number

Exit: No registers specified

This function causes a rectangular region on the screen to be blanked. The start column and row specify the upper left corner of the region to be blanked. The current screen mode (reverse/normal) is ignored, and the blanked region will be cleared so that all pixels are turned off. The reverse video attribute bits for the specified region must also be cleared. This function does not affect the cursor location or the current setting of the reverse video mode.

Screen (LCD) Graphics Support

BIOS graphics support is fashioned after the GW-BASIC 2.0 Graphics Interface Specification. A thorough reading of that document is recommended prior to implementing the BIOS graphics support described below.

The BIOS maintains a graphics cursor which uniquely addresses a pixel of the display. The internal representation of the graphics cursor should be chosen by the OEM to optimize the speed of graphics operations. The only restriction on the internal representation is that it must not exceed 6 bytes in length.

The BIOS also maintains a graphics attribute which specifies what effect a wet pixel(s) BIOS call has on the affected pixel(s). (The LCDs being used for Hand-Held computers today support only two states for a pixel, on or off. Therefore the graphics attribute simply specifies one of these two states.) As with the graphics cursor, the internal representation of the graphics attribute should be chosen to optimize the speed of graphics operations. There are no size restrictions on the internal representation of the graphics attribute since only the external form is used when passing parameters to and from the BIOS. Because graphic functions are speed critical routines, they never turn the cursor on or off even if the cursor disturbs their work. Turning the cursor on or off should be performed before or after invocation of graphic functions if it is necessary.

Graphics coordinates are specified such that Y runs vertically with 0 being the top row of pixels on the screen, and X runs horizontally with 0 being the leftmost column of pixels on the screen.

The following interrupts support graphics on the screen:

INT 52h, Function 0 - Return Screen Graphics Resolution

Entry: AH = 0 (function code)

Exit: CX = highest numbered pixel in X
(number of pixels in X direction)-1
DX = highest numbered pixel in Y
(number of pixels in Y direction)-1

INT 52h, Function 1 - Map X,Y Coordinates to Graphics Cursor

Entry: AH = 1 (function code)
CX = X coordinate
DX = Y coordinate

Exit: No registers specified

This routine sets the graphics cursor to select the pixel specified by the X and Y coordinates for subsequent graphics BIOS calls.

INT 52h, Function 2 - Move the Graphics Cursor Right One Pixel

Entry: AH = 2 (function code)

Exit: FLAGS.C = 0 - cursor updated
FLAGS.C = 1 - cursor was already at maximum X, therefore it was not changed

INT 52h, Function 3 - Move the Graphics Cursor Left One Pixel

Entry: AH = 3 (function code)

Exit: FLAGS.C = 0 - cursor updated
FLAGS.C = 1 - cursor was already at minimum X (0), therefore it was not changed

INT 52h, Function 4 - Move the Graphics Cursor Down One Pixel

Entry: AH = 4 (function code)

Exit: FLAGS.C = 0 - cursor updated
FLAGS.C = 1 - cursor was already at maximum Y, therefore it was not changed

INT 52h, Function 5 - Move the Graphics Cursor Up One Pixel

Entry: AH = 5 (function code)

Exit: FLAGS.C = 0 - cursor updated
FLAGS.C = 1 - cursor was already at minimum Y (0), therefore it was not changed

INT 52h, Function 6 - Set Graphics Attribute

Entry: AH = 6 (function code)
AL = external representation of attribute
0 for turning pixels off
1 for turning pixels on

Exit: No registers specified

This routine sets the graphics attribute as specified. It should also set up the internal representation of the graphics attribute accordingly.

INT 52h, Function 7 - Set Current Pixel to Current Attribute

Entry: AH = 7 (function code)

Exit: No registers specified

This routine sets the pixel addressed by the graphics cursor to the current graphics attribute. This has the effect of turning off (graphics attribute = 0) or turning on (graphics attribute = 1) the pixel.

INT 52h, Function 8 - Set Multiple Pixels to Current Attribute

Entry: AH = 8 (function code)
BX = number of pixels affected

Exit: No registers specified

This routine turns off or on (depending on the current graphics attribute) BX pixels at and to the right of the current graphics cursor. For example, if BX = 2, the pixel addressed by the graphics cursor and the pixel immediately to its right are set to the current graphics attribute (0 or 1).

Notes

1. The graphics cursor must be unchanged upon exit from this routine.

2. It is assumed that the caller will never specify a pixel count such that the operation extends beyond the rightmost pixel of the screen. Therefore this routine need not check for this condition.

3. It is assumed that the caller will never specify a pixel count of 0. Therefore this routine need not check for this condition.

4. This routine should be written to handle multiple pixels in an efficient manner. For example, all affected pixels in a given byte should be handled with a single fetch and store of the byte from video memory. In fact, if all the bits of a given byte are affected it is not even necessary to fetch the byte from video memory.

INT 52h, Function 9 - Read Attribute of Current Pixel

Entry: AH = 9 (function code)

Exit: AL = external attribute of pixel
0 for pixel off, 1 for pixel on

INT 52h, Function A - Fetch Graphics Cursor

Entry: AH = A (function code)

Exit: BX = word 2 of graphics cursor
CX = word 3 of graphics cursor
DX = word 1 of graphics cursor

INT 52h, Function B - Store Graphics Cursor

Entry: AH = B (function code)
BX = word 2 of graphics cursor
CX = word 3 of graphics cursor
DX = word 1 of graphics cursor

Exit: No registers specified

Since the internal representation of the graphics cursor is unknown outside of the BIOS, the fetch and store graphics cursor routines are not intended to allow the caller to operate on the cursor. Instead, these routines are provided solely to allow the caller to preserve and later restore the contents of the graphics cursor.

INT 52h, Function C - Return Aspect Ratio

Entry: AH = C (function code)

Exit: BX = 256 * (aspect ratio)
DX = 256 / (aspect ratio)

The aspect ratio of the screen is defined as the ratio of the Y distance between two vertically adjacent pixels to the X distance between two horizontally adjacent pixels, where the distances are measured from the center of the first pixel to the center of the second pixel. Typically this ratio is less than 1, so a given number of pixels spans a shorter distance in Y than in X.

The aspect ratio can be used by callers to compensate for unequal X and Y pixel spacing when drawing graphics on the screen. For example, the CIRCLE statement in GW-BASIC uses the aspect ratio to draw spatially round circles.

The aspect ratio can be empirically determined as follows:

1. Turn on a box of pixels of dimension N by N, where N is the minimum of the screen's pixel resolution in X and Y. (In other words, draw the largest box the screen can accommodate having the same number of pixels in X and Y.)
2. Measure the Y (vertical) dimension of the box.
3. Measure the X (horizontal) dimension of the box.
4. Divide the Y dimension of the box by the X dimension to obtain the aspect ratio of the screen.

INT 52h, Function D - Get Pixels

Entry: AH = D (function code)

Exit: AL = pixel states (see description below)
BL = number of pixels returned

This routine returns a byte in AL representing the state of the pixels at and to the right of the pixel addressed by the graphics cursor. Bit 7 (MSB) of AL corresponds to the pixel addressed by the graphics cursor, bit 6 corresponds to the pixel immediately to its right, and so on.

Normally this routine returns 8 pixels so BL contains 8. However, if the right screen edge is encountered prior to getting 8 pixels, the count in BL is less than 8 and the unused low order bits of AL are undefined.

Normally this routine leaves the graphics cursor pointing to the first pixel to the right of the last pixel returned. However, if the last pixel returned is the rightmost pixel of the screen, the graphics cursor is left pointing to it rather than pointing beyond the right edge of the screen.

This is a speed critical routine, it should be coded accordingly.

INT 52h, Function E - Put Pixels

Entry: AH = E (function code)
AL = pixel data
BH = operation code
0 - Off
1 - On
2 - XOR
3 - AND
4 - OR
BL = number of pixels to affect
1 through 8, inclusive

Exit: No registers specified

This routine affects the states of BL pixels at and to the right of the graphics cursor.

Operation code 0 performs a logical NOT on the pixel data in AL, and then sets the appropriate screen pixels accordingly.

Operation code 1 simply sets the appropriate screen pixels to the pixel data contained in AL.

Operation codes 2, 3, and 4 perform the following steps:

1. Read the current state of the affected pixels from the screen.

2. Perform the specified logical operation on the data obtained in step 1 and the data in AL. The pixel data in AL is left aligned in the register, bit 7 corresponds to the current cursor location, bit 6 to the pixel to the right of the current cursor, etc.

3. Set the state of the affected pixels to the result of the logical operation performed in step 2.

Upon exit, the graphics cursor points to the pixel to the right of the last one affected by the routine. However, if the last pixel affected is the rightmost pixel on the screen, the graphics cursor is left pointing to it rather than pointing beyond the edge of the screen.

This is a speed critical routine, it should be coded accordingly.

INT 52h, Function F - Set Pixel at Specified X,Y to Specified Attribute

Entry: AH = F (function code)
AL = attribute
0 - off
1 - on
CX = X coordinate
DX = Y coordinate

Exit: No registers specified

Upon exit, the graphics cursor points to the pixel specified by the X,Y coordinate pair, and the graphics attribute is the one that was passed to the routine in AL.

Calling this routine is the simplest way to plot individual points on the screen. It is intended for use by applications that do not require maximum speed. It may therefore be coded as calls to the routines that map an X,Y coordinate pair to the graphics cursor, set the graphics attribute, and set the current pixel to the current attribute.

Cassette Support - Unsupported Software Interrupt

The cassette block format is as follows:

<driver-header><block-type><block-data><driver-trailer>

1. The format of the driver-header is specified by the OEM. Typically it includes synchronization information and the length of the block.

2. The block-type and block-data are supplied by the caller when the block is written to tape. (See the description of the write block routine below.)

3. The format of the driver-trailer is specified by the OEM. Typically it includes a checksum for the block.

If cassette hardware is not available, the cassette functions should return the Device Not Available error code.

The following interrupts support cassette I/O:

INT 53h, Function 0 - Request Cassette Device

Entry: AH = 0 (function code)

Exit: AL = Standard BIOS error codes

This function is called before any other cassette functions. Its purpose is to mark the cassette unit as being in use.

INT 53h, Function 1 - Initialize Cassette

Entry: AH = 1 (function code)

Exit: AL = Standard BIOS error codes

This function code is called after a successful request has attached the cassette device. It is called once before any I/O to initialize the cassette.

INT 53h, Function 2 - Turn Off Cassette Motor

Entry: AH = 2 (function code)

Exit: No registers specified

INT 53h, Function 3 - Turn On Cassette Motor

Entry: AH = 3 (function code)

Exit: No registers specified

INT 53h, Function 4 - Write Block

Entry: AH = 4 (function code)
AL = block-type
CX = byte count (does not include block-type)
DX = segment address of data buffer
BX = offset address of data buffer

Exit: AL = Standard BIOS error codes

This routine should perform the following steps:

1. Start the motor and delay until the motor is at full speed. (The length of the delay required depends on the characteristics of the cassette drive. For inexpensive cassette recorders a delay of one-half second is usually adequate.)

2. Write the OEM specified block-header.

3. Write the block-type byte.

4. Write the specified number of bytes from the specified buffer.

5. Write the OEM specified block-trailer.

6. Turn the motor off and return.

INT 53h, Function 5 - Read Block

Entry: AH = 5 (function code)
CX = buffer length
DX = segment address of data buffer
BX = offset address of data buffer

Exit: AL = Standard BIOS error code
AH = block-type (read from tape)
CX = number of bytes of data placed in the buffer

This routine should perform the following steps:

1. Start the motor and delay until the motor is at full speed. (The length of the delay required depends on the characteristics of the cassette drive. For inexpensive cassette recorders a delay of one-half second is usually adequate.)

2. Read the OEM specified block-header.

3. Read the block-type byte.

4. Read the data into the buffer, checking for buffer overflow.

5. Read the OEM specified block-trailer, and verify the checksum.

6. Turn the motor off and return.

INT 53h, Function 6 - Deactivate Cassette Device

Entry: AH = 6 (function code)

Exit: AL = Standard BIOS error code

This function is called once after all cassette I/O is completed. Its purpose is to perform any clean-up required after using the cassette, before it is released.

INT 53h, Function 7 - Release Cassette

Entry: AH = 7 (function code)

Exit: AL = Standard BIOS error code

This function will be called after the completion of cassette I/O processing. Its purpose is to mark the cassette device as being not busy.

Calendar, Time of Day Clock, and Alarm Support

The format of the date and time is identical to that used by MS-DOS. Portions of the time specification that are not supported by the hardware should be ignored by set time calls and returned as zero by get time calls.

The power on alarm should only be active if the machine power is off. The high level applications will handle all alarm functions in software whenever the power is on. If a hardware alarm occurs while the machine is on, it should be ignored. Alarm warm-start feature works during power-off, however, alarm interruption is ignored during power-on.

The following interrupts support the time of day clock and alarm:

INT 54h, Function 0 - Get Current Date

Entry: AH = 0 (function code)

Exit: AL = day of week (0 = Sun, 1 = Mon, . . ., 6 = Sat)
CX = year (1980 through 2099)
DH = month (1 through 12)
DL = day (1 through 31)

INT 54h, Function 1 - Set Current Date

Entry: AH = 1 (function code)
CX = year (1980 through 2099)
DH = month (1 through 12)
DL = day (1 through 31)

Exit: AL = return code
0 - no errors
FF - invalid date, operation aborted

INT 54h, Function 2 - Get Current Time

Entry: AH = 2 (function code)

Exit: CH = hours (0 through 23)
CL = minutes (0 through 59)
DH = seconds (0 through 59)
DL = hundredths of seconds (0 through 99)

INT 54h, Function 3 - Set Current Time

Entry: AH = 3 (function code)
CH = hours (0 through 23)
CL = minutes (0 through 59)
DH = seconds (0 through 59)
DL = hundredths of seconds (0 through 99)

Exit: AL = return code
0 - no errors
FF - invalid time, operation aborted

INT 54h, Function 4 - Get Alarm Date

Entry: AH = 4 (function code)

Exit: AL = day of week (0 = Sun, 1 = Mon, . . ., 6 = Sat)
CX = year (1980 through 2099)
DH = month (1 through 12)
DL = day (1 through 31)

INT 54h, Function 5 - Set Alarm Date

Entry: AH = 5 (function code)
CX = year (1980 through 2099)
DH = month (1 through 12)
DL = day (1 through 31)

Exit: AL = return code
0 - no errors
FF - invalid date, operation aborted

INT 54h, Function 6 - Get Alarm Time

Entry: AH = 6 (function code)

Exit: CH = hours (0 through 23)
CL = minutes (0 through 59)
DH = seconds (0 through 59)
DL = hundredths of seconds (0 through 99)

INT 54h, Function 7 - Set Alarm Time

Entry: AH = 7 (function code)
CH = hours (0 through 23)
CL = minutes (0 through 59)
DH = seconds (0 through 59)
DL = hundredths of seconds (0 through 99)

Exit: AL = return code
0 - no errors
FF - invalid time, operation aborted

INT 54h, Function 8 - Enable Alarm

Entry: AH = 8 (function code)

Exit: no registers specified

INT 54h, Function 9 - Disable Alarm

Entry: AH = 9 (function code)

Exit: no registers specified

INT 54h, Function A - Get Timer Resolution

Entry: AH = 0Ah (function code)

Exit: AX = Timer interrupt resolution

This function returns the resolution of the timer interrupt. The number to be returned in AX is 65535/N, where N is the number of interrupts that occur per second.

For example, if timer interrupts occur 18.2 times per second, the number returned in AX would be 65535/18.2 or 3601.

Real Time Clock: (RTC) with 64-bytes built-in RAM is used as follows:

Offset	I/O Address	Contents	Initial Value
0 - 17		Used by RTC itself	
18	C012 (Hex)	Printer Timeout 500 msec./unit	40 (20 sec.)
19	C013 (Hex)	Break Sending Period 25 msec./unit	30 (750 msec.)
20	C014 (Hex)	Reserved	
21	C015 (Hex)	Print Code Translate 0: no translation 1: translate	0 (no translation)
22	C016 (Hex)	Carrier Detect Timeout 500 msec./unit	4 (2 sec.)
23	C017 (Hex)	Power ON Mode 0: Jump to MS-Works FFH: Jump to top of Removable Rom	0 (to MS-Works)
24 - 63	C018 - C03F (Hex)	User free	

Sound Support

The following interrupts provide sound support:

INT 55h, Function 0 - Play Tone

Entry: AH = 0 (function code)
CX = frequency in Hertz
DX = duration in 2.5 millisecond increments

Exit: No registers specified

This function should generate the specified tone for the specified duration. If a tone is already being generated when this function is called, the current tone should be completed before the next tone is begun. If the user types the break key while sound generation is in progress, the sound should be stopped and control returned to the calling program.

The Tandy 600 has only one hardware timer (81C55) which is used for both playing tones and setting the RS232 baud rate. The ROM-BIOS ignores the tone function while using the timer as a baud rate generator. The tone function is, therefore, only available when the timer is not used as a baud rate generator.

The tone function can produce the OHz tone (i.e. Pause).

Printer Support

The following interrupts support the printer:

INT 56h, Function 0 - Request Printer

Entry: AH = 0 (function code)

Exit: AL = Standard BIOS error code

This function called prior to accessing the printer. Its purpose is to mark the printer device as in use.

INT 56h, Function 1 - Initialize Printer

Entry: AH = 1 (function code)

Exit: AL = Standard BIOS error code

This function is called once after successfully requesting the printer. Its purpose is to perform any initialization required before sending an output stream to the printer.

INT 56h, Function 2 - Print Character

Entry: AH = 2 (function code)
AL = character to print

Exit: AL = Standard BIOS error code

INT 56h, Function 3 - Print Screen

Entry: AH = 3 (function code)

Exit: AL = Standard BIOS error code

This function may be implemented as either a graphic print screen if graphic output on the printer is supported, or as an alpha only print screen if graphic printer output is not supported. If graphic output is supported, correction for different aspect ratios between LCD screen and printer may be provided, but is not required.

INT 56h, Function 4 - Deactivate Printer

Entry: AH = 4 (function code)

Exit: AL = Standard BIOS error code

This function is called once after a stream of printer output is completed, and before the printer is released. It is intended to perform any clean-up necessary when releasing the printer.

INT 56h, Function 5 - Release Printer

Entry: AH = 5 (function code)

Exit: AL = Standard BIOS error code

The purpose of this function is to mark the printer as being not in use.

Disk Support

The Disk Support functions have been designed to provide compatibility with MS-DOS style disks. Several of the terms used in the descriptions of the functions are taken from MS-DOS, and it is recommended that the BIOS implementor be familiar with the MS-DOS 2.0 Adaptation Guide.

Several of the disk support function calls refer to a BPB. The BPB used by the HH O/S is identical to that used by MS-DOS 2.X and described in the MS-DOS 2.0 Adaptation Guide. Even if the BPB on the boot sector was not the same as one on the BIOS-ROM, the BIOS will not return an error because the Tandy 600 is capable of reading another logical formatted disk. The description of the structure of a BPB is repeated here:

WORD	Sector size in bytes must be a multiple of 64
BYTE	Sectors per allocation unit must be a power of 2 (1, 2, 4, etc.)
WORD	Number of reserved sectors may be 0
BYTE	Number of FATs
WORD	Number of directory entries
WORD	Total number of sectors includes reserved sectors
BYTE	Media descriptor byte must be unique for each unique BPB
WORD	Number of sectors occupied by a single FAT

The following interrupts provide the disk support functions:

INT 57h, Function 0 - Request Disk Device

Entry: AH = 0 (function code)

Exit: AL = Standard BIOS error code

This function will be called before any disk requests are made. Its purpose is to mark the disk device as in use.

INT 57h, Function 1 - Initialize Disk Device

Entry: AH = 1 (function code)

Exit: AL = Standard BIOS error code
DS:SI = Address of BPB pointer array
CL = Number of disks

This function initializes the disk device. It is called before the first access to the disk. It is also used as a disk reset function when an error occurs.

The BPB pointer array is an array of FAR pointers, one for each supported disk drive. The size of the scratch buffer passed to the Build BPB function call is determined from the BPB's passed in this call. The buffer size will be taken as the size of the largest sector size returned from this call.

INT 57h, Function 2 - Build BPB

Entry: AH = 2 (function code)
AL = Drive number (starting with 0)
DS:SI = Address of scratch buffer
CL = Media descriptor byte

Exit: AL = Standard BIOS error code
DS:SI Pointer to new BPB

The scratch buffer is available for whatever use the BIOS may require. On entry, it is not initialized. The size of the buffer is determined from the default BPB's returned from function 1, and will be as large as the largest sector size specified.

INT 57h, Function 3 - Read Sector(s)

Entry: AH = 3 (function code)
AL = drive number (starting with 0)
BL = Media descriptor byte
CX = number of sectors to read
DX = starting logical sector number
DS:SI = starting memory address

Exit: AL = Standard BIOS error code
12 - data error
14 - write protect (does not apply to read)
15 - drive not ready
16 - seek error
17 - sector not found
18 - write fault (does not apply to read)
0FFh - other error

NOTE: Read/Write Sector(s) Functions: The FDD circuit is automatically powered off after 2 or 3 seconds of non-operation. The Read Sector(s) and Write Sector(s) functions turn on the power of FDD circuit and its motor when their power has been turned off automatically.

INT 57h, Function 4 - Write Sector(s)

Entry: AH = 4 (function code)
AL = drive number (starting with 0)
BL = Media descriptor byte
CX = number of sectors to write
DX = starting logical sector number
DS:SI = starting memory address
FLAGS.C = verify flag
0 - no verify
1 - verify after write

Exit: AL = Standard BIOS error code
(See function 2 for pertinent error codes)

INT 57h, Function 5 - Get Format Table Size

Entry: AH = 5 (function code)

Exit: DX = Size of format table

Many disk controllers require a data table which contains formatting information, (such as sector ID fields). This call should return the size of the buffer needed to build this table. The specified amount of memory will be allocated, and the address of this table passed to the Format Disk function.

INT 57h, Function 6 - Initialize Disk Formatting

Entry: AH = 6 (function code)
AL = Drive Number (starting with 0)
DS:SI = Sector size scratch buffer
ES:DI = Pointer to format parameter string

Exit: AL = Standard BIOS error code
BL = FAT ID byte
BH = Change disk flag
0 = don't change disk
0FFh = change disk
DS:SI = Pointer to BPB

This function is called once at the beginning of formatting. It specifies the location of a sector buffer for use in transferring a boot sector to the newly formatted disk. The data placed into this sector buffer will be provided in the Format Disk function call for placement on the formatted disk.

The parameter string specified in ES:DI is a zero terminated ASCII character string. It is specified by the user from within the FORMAT application program. The actual contents of this string is OEM specified, and may have any meaning desired. Microsoft does not make use of this string in any way. The BIOS-ROM allows the following two parameters:

V = verify option:	Verify formatted disk. The formatting takes longer with the verify option, however, it gives higher reliability.
B = Copy boot sector option:	Usually only the BPB table is copied into the boot sector of a formatted disk, however, all data of the boot sector is copied if this option is specified. Change disk flag:0FFh is returned if this option is specified.

The change disk flag on exit specifies whether to prompt the user to place a new disk in the drive before formatting is begun. If this flag is set on return from the function, the user will be prompted to change the disk before function 7 is called.

The scratch buffer specified in DS:SI will be a sector sized block of memory which may be used in any way. The size of this block is determined from the BPB array returned via Function 1 (Initialize Disk Device) and will be the same size as the largest sector size in the BPB array. This sector may be used in any way desired.

INT 57h, Function 7 - Format Disk

Entry: AH = 7 (function code)
AL = drive number (starting with 0)
DS:SI = Boot sector buffer
ES:DI = Format table buffer

Exit: AL = Standard BIOS error code
BX = Bad sector number
CX = Number of consecutive bad sectors

This routine is called to format the disk. If a bad sector or group of bad sectors is encountered, this function should return with the bad sector number and the count of consecutive bad sectors. This information is used in building a list of bad sectors in the FAT. This function will be called repeatedly until a final return occurs in which CX = 0 (no bad sectors) is reported. At that time, it is assumed that the entire disk has been formatted.

The scratch sector buffer passed in DS:SI is the same buffer as was passed in Function 6 (Initialize for Disk Formatting).

INT 57h, Function 8 - Media Check

Entry: AH = 9 (function code)
AL = drive number
CL = media descriptor byte

Exit: AL = media status
-1 = media has been changed
0 = don't know
1 = media has not been changed

The Media Check function always returns 0 which means the Tandy 600 does not know whether the media has been changed or not. Therefore, the Build BPB function is called frequently. However, this is not a big overhead cost because the ROM-BIOS only returns the fixed BPB without any FDD access. This means that the Tandy 600 only can access FDD formatted media following one of the two standard MS-DOS formats. The other standard MS-DOS format is not available. However, the disk can be accessed by other MS-DOS machines (i.e. compatibility must be resolved by the desk-top MS-DOS machine — not by the Tandy 600).

The disk format is one of the two MS-DOS standard formats.

Number of Tracks:	80
Sectors per Tracks:	9
Bytes per Sector:	512
Reserved sectors:	1
Number of FATs:	2
Sectors per FAT:	2
Total DIR entries:	112
Sectors per DIR:	7
Sectors per allocation unit:	2
Media Descriptor Byte:	F8
Physical Format:	IBM format

Boot Sector Format

	3-bytes JUMP Code to BOOTER		If the JUMP instruction is not on here, the BIOS assumes that this is not its original diskette.
	8-bytes OEM name and version "Tandy 600"		
B.P.B.	W: Bytes per Sector:	512	The Tandy 600 supports only one MS-DOS standard disk format. Another MS-DOS machine can read/write to this diskette because it has BPB on the boot sector.
	B: Sectors per allocation unit:	2	
	W: Reserved Sectors:	1	
	B: Number of FATs:	2	
	W: Number of root DIR entries:	112	
	W: Total number of Sectors includes reserved sectors	720	
	B: Media descriptor:	F8 hex	
	W: Number of Sectors per FAT	2	
Optional B.P.B.	W: Sectors per Track:	9	
	W: Number of Head:	1	
	W: Number of hidden Sectors:	0	
	BOOTER CODE		If there are two NOP (90 hex) codes instead of BOOTER code here, the BIOS assumes that this diskette is not an uncooperative system diskette.

INT 57h, Function 9 - Deactivate Disk Device

Entry: AH = 9 (function code)

Exit: AL = Standard BIOS error code

This function is called once after access to the disk device is completed. It should perform any clean up necessary before the disk device is released.

INT 57h, Function A - Release Disk Device

Entry: AH = A (function code)

Exit: AL = Standard BIOS error code

This function is called when access to a disk drive is to be terminated. The device should be marked as not busy. This function will always turn off the power to the FDD circuit and its motor.

Power On

When the power is turned on, the BIOS must determine the type of start up desired and perform as follows:

1. Cold start - This is indicated by the user holding down a unique, OEM specified key combination while turning the power on. (The key combination should be chosen such that it will not occur accidentally, since cold starting destroys any information stored in the machine.) The BIOS executes its cold start initialization sequence and then transfers control to the entry point of Microsoft's ROM code as discussed below.

2. Warm start resume - This is indicated by no keys being down when the power switch is turned on. In this case, the BIOS executes any warm start code it needs to and then transfers control to the entry point of Microsoft's ROM code as discussed below.

3. Warm start break - This is indicated by the user holding down a unique, OEM specified key or key combination (Microsoft recommends the Break key be used for this) while turning the power on. The BIOS executes any warm start code it needs to and then transfers control to the entry point of Microsoft's ROM code as discussed below. This is intended to be a reset function which will always cause a return to the top level menu. If the machine hangs for any reason, the user can use this function to regain control. See further discussion of Warm Start Break below.

4. Warm start alarm - This occurs when the wake up alarm turns on the power. This always causes a warm start, regardless of what, if any, keys may be depressed at the time. (This prevents accidental cold starts and boot attempts if the machine automatically powers up while an unsuspecting user is "playing" with the keyboard.) In this case, the BIOS executes any warm start code it needs to and then transfers control to the entry point of Microsoft's ROM code as discussed below.

5. Warm start auto answer - This occurs when the modem auto answer feature turns on the power. This always causes a warm start, regardless of what, if any, keys may be depressed at the time. (This prevents accidental cold starts and boot attempts if the machine automatically powers up while an unsuspecting user is "playing" with the keyboard.) In this case, the BIOS executes any warm start code it needs to and then transfers control to the entry point of Microsoft's ROM code as discussed below.

The entry point of Microsoft's ROM code is the lowest ROM location occupied by Microsoft code in a minimum configuration of the machine. When the BIOS jumps to this location the registers must be set up as follows:

AL = type of start up
- 0 - cold start
- 1 - warm start resume
- 2 - warm start break
- 3 - warm start alarm
- 4 - warm start auto answer

DS:BX - Address of ROM pointer table
CX = Number of paragraphs of RAM available to Microsoft's ROM code
DX = Base paragraph address of available RAM

The memory information in CX and DX is used by the internal operating system to initialize the file system during cold start.

The address in DS:BX points to a table of segment addresses which give the base addresses of each separate contiguous ROM region in the system. On cold start, this is used to build the initial file directory. On warm starts, this is used to detect removable ROM's which have been removed, or have different rom's installed. Note, that if during a warm start resume, the O/S detects that there are different ROM's present in the machine than when the power was turned off, the O/S will force a warm start break operation to occur.

The structure of this table is as follows:

```
count          - Number of ROM regions defined
region 1       - Segment address of the first
.                ROM region
.
region n       - Segment address of the last ROM region
```

Power On Operation

Power-on mode	Key combination		Value of Reg. AL
Cold Start	LABEL + ALT + BKSP	+ power-on	AL = 0
Warm Start Resume	no-key	+ power-on	AL = 1
Warm Start Break	SHIFT + PAUSE	+ power-on	AL = 2
Warm Start Alarm	(Auto Power On)		AL = 3
Warm Start Auto-Answer	(Auto Power On)		AL = 4
Boot Uncooperative System	LABEL + ALT + ESC	+ power-on	AL = 5

(Hold key after power-on, and release it a little bit later)

The feature of the Boot Uncooperative System loads the boot sector data into the highest 512 bytes of the standard RAM memory and transfers control to it. Usually, the Tandy 600's diskette does not include any booter and uncooperative system program.

The ROM-BIOS transfers control with CLI (disable INT) state. Therefore, the STI (enable INT) should be done after control is transferred.

Resetting Operation

Key combination at Reset: You can use a RESET switch on the left side of Tandy 600 as follows.

Hold LABEL + ALT + BKSP and Push RESET: Cold-Start
Hold no keys and Push RESET: Warm-Start break
Hold SHIFT + PAUSE and Push RESET: Warm-Start break
Hold LABEL + ALT + ESC and Push RESET: Load unco-op sys.
(Hold keys after pushing RESET, and release them a little bit later.)

NOTE: The resume information used at the warm-start resume is stored after NMI interrupt, i.e. power-off. Therefore, you cannot cause a warm-start resume by using the reset-switch, because of the lack of resume information. If you push only the RESET switch, the Tandy 600 begins warm start break.

Further Considerations Regarding Warm Start Break

Within the O/S, there are critical regions of code that must be allowed to complete once they are begun. These regions include file management operations that cause memory files to move, and database functions that modify the contents of a database file. If one of these regions were entered, but not allowed to complete, the in-memory file system would be destroyed, or a database file would be corrupted. Some of these critical code regions can take sufficiently long to execute, that it is not reasonable to turn interrupts off while executing them. Also, if power off requests occur via NMI, it would not be possible to prevent one from occurring. Because these critical regions of code can be interrupted by a power off request, it is essential that they complete when the power is restored.

This is not a problem if a Warm Start Resume is performed when the power is restored, because the warm start resume returns to whatever was executing when the power off occurred. However, as warm start break is not intended to return to the interrupted process, but break out if it and restart the Top Level program, a potential problem could occur in doing a warm start break. The following discussion describes what happens during a warm start break when one of the critical regions of code in the O/S was interrupted:

1. When a user program calls an O/S function, the O/S pops the user's return address off of the user's stack and saves it in a pair of local variables. This is done so that the segment portion of the return address can be kept correct if the program is running from RAM and the execution of the system call causes the file containing the program code to move.

The O/S then switches to its own stack and begins executing the function call.

When the O/S enters a critical region of code that must complete, it sets a flag that will cause Warm Start Break to resume that system call before the warm start break occurs.

O/S stack	O/S is executing critical region of code.
O/S return address	Power off hardware interrupt occurs

2. The hardware interrupt transfers control to the BIOS, which switches to its own interrupt handler stack.

BIOS stack	The BIOS performs whatever actions are necessary, and then performs an INT 4Eh.
BIOS Return address	

3. The INT 4Eh transfers control to the O/S Power Off Hook, which saves the machine state so that it can be restored during Warm Start Resume.

O/S power off stack	The O/S saves the address of the BIOS power off stack, switches to its own power off stack, and pushes all of the registers. It saves the location of the stack pointer in its own power off stack, and then performs an INT 58h, function 1 to tell the BIOS to turn the power off.
O/S Return address	

4. When the power is turned back on, and the BIOS determines that a Warm Start Break is to be performed, it transfers control to the O/S entry point with AL = 2 to indicate that a Warm Start Break should occur.

5. The O/S tests the flag and determines that a critical region of O/S code was being executed. The O/S patches the variables that contain the return address of the application program that originally called the O/S function, so that when the O/S call completes and performs a return, it will return to the Warm Start Break code instead of the application program. The O/S then transfers to the Warm Start Resume routine which resumes execution of whatever was interrupted by the power off request.

O/S power off stack
O/S Return address

The O/S switches to its power off stack, and restores all of the registers. It then switches back to the stack that was in effect when the O/S power off hook was entered, and performs an IRET to return control to whoever executed the INT 4Eh instruction.

6. The IRET will return control to the BIOS power off interrupt handler. This handler must then restore the stack that was in effect when it was entered and return to the interrupted process.

BIOS Power off stack

The BIOS switches back to the O/S stack that was in effect when the power off request occurred, and performs an IRET to return control to whoever was executing when the hardware power off interrupt occurred.

7. The IRET will return control to the interrupted O/S function which was executing the critical region of code.

O/S stack

The interrupted O/S call resumes execution and completes. It then pushes the patched return address onto the stack and performs a FAR return. The will transfer control back into the O/S Warm Start Break code.

8. The O/S warm start break code will then perform a normal warm start break. It will reinitialize everything except the RAM file system, and build a new directory for whatever ROM's are present in the machine. It will then perform an INT 58h, function 1 to tell the BIOS that its Warm Start Break processing is finished, and that the BIOS should complete its initialization. After executing the INT 58h, function 1, the O/S will then execute the System Manager program which will draw the top level menu.

The most important point of this discussion is that when the power off request interrupted a critical region of the O/S, the O/S Warm Start Break code will perform a Warm Start Resume to allow the interrupted O/S function to complete before doing the Warm Start Break. If any of the BIOS's Warm Start Break initialization would cause a Warm Start Resume to not be performed properly, then the Warm Start Break will fail in this case. The BIOS must not initialize anything internally that would cause a Warm Start Resume to fail until it receives the INT 58h, function 1 request.

General System Control Functions

The following functions are used for controlling various global system functions.

INT 4Eh - Power Low/Off Hook

Entry: No registers specified

Exit: No registers specified

When a power low or power off interrupt occurs, if the BIOS is in the middle of an I/O operation, it should complete the operation, and then execute an INT 4Eh. At any other time, the INT 4Eh should be executed immediately. The INT 4Eh code will save the necessary machine state to allow resumption when the power is restored. At some later time, when the power has been restored, the machine state will be restored, and the INT 4Eh hook will return to its caller. At that time, the CPU state will be the same as when the INT 4Eh hook was entered.

When the Microsoft INT 4Eh code completes it executes an INT 58h Function 0, which causes the BIOS to turn off the power. When the power is restored, and a warm start resume is performed, the INT 4Eh hook will return to its caller with the CPU state unmodified. If a cold start or warm start break is performed when the power is restored, the INT 4Eh hook will not return to the caller, except in the case described above where a critical region of code in the O/S was being executed.

The HH O/S power off hook routine is not re-entrant. The first thing this routine does on entry is to set the INT 4Eh vector to point to a dummy IRET. The HH O/S power on code resets this vector to point back to the proper INT 4Eh power off handler just before it transfers control to the appropriate application program for the type of restart being performed. This reduces the chances of re-entrancy problems if NMI's are used for power off requests.

> **NOTE:** When the Power low/off interrupt occurs during the disk read/write operation, the invocation of Power low/off hook is put off until the end of a read/write operation to keep the media from being damaged.

A previous FDD operation cannot be resumed even if its power is restored, because a disk may be destroyed by resuming a write operation when another disk has been inserted during power-off.

When the Power low/off interrupt occurs during communication via RS232 or a MODEM, the invocation of Power low/off hook is done immediately. When its power is turned on again, however, the previous RS232 and MODEM configurations and connection of the telephone line are fully recovered.

When the Power low/off interrupt occurs during printing, the invocation of Power low/off hook is done immediately. When its power is turned on again, however, printing can be resumed.

INT 58h, Function 0 - Power Off

Entry: AH = 0 (function code)

Exit: Does not return

This function causes the main power to the CPU to be shut off. It will be called by the Power low/off hook (INT 4Eh) as part of the normal power off sequence. It is also called by the Microsoft code to turn the power off if the machine has been idle for a certain length of time, or if the user explicitly turns the power off through software. It does not return. When power is restored, either via the power switch or some automatic mechanism (such as the alarm), control is transferred to the Microsoft code as described in the Power On section of this document.

INT 58h, Function 1 - System Re-Initialization

Entry: AH = 1 (function code)

Exit: none

This function is called during a warm start resume, warm start alarm, or warm start auto-answer, if it is determined that it is not possible to resume whatever was running when the power was shut off. In this case, a warm start break is performed. This function will be called to inform the BIOS that a warm start break is being performed, and that the BIOS should do any initialization that it needs to for a warm start break.

This function is also called when the O/S has completed its Warm Start Break processing, and is ready to execute the System Manager program. See the above discussion of Warm Start Break.

Interval Timer

When the BIOS receives an interval timer interrupt it must execute an INT 4Dh sometime during the interrupt routine.

INT 4Dh - Interval Timer Hook

Entry: No registers specified

Exit: No registers specified

The ROM-BIOS gives interrupt every 31.25 msecs.

Communications/Modem Support

The RS-232 receiver is interrupt driven. The BIOS handles the receiver interrupts and checks for error conditions. If an error is detected within the interrupt service routine, the error code must be saved so that it can be returned as part of the exit conditions for the next call to INT 59h, functions 8 - C. Just prior to placing the data into the queue, the BIOS must execute an INT 4Ch. This allows code to be hooked into the receiver interrupt routine for purposes of data filtering and event trapping.

INT 4Ah - Ring Detect Hook

Entry: No registers specified

Exit: No registers specified

This interrupt should be invoked when an incoming call is detected on the telephone line.

INT 4Ch - Serial Receiver Queue Hook

Entry: AL = Character Received
AH = Serial device the character came from
FLAGS.C = 0

Exit: FLAGS.C = 0 - the character (in AL) should be put into the queue
FLAGS.C = 1 - the character should be discarded (i.e., do not put the character into the queue)
AL = Character to put in queue

There must be at least one serial I/O port, although there may be more if desired. Function 0Fh is used to select which serial port is to become the active port. After a call to function 0Fh, the selected port will become the port acted upon by function calls 07h - 0Eh.

The following interrupts support the serial I/O ports, modem, and telephone dialing:

INT 59h, Function 0 - Disconnect Serial Device from Modem

Entry: AH = 0 (function code)

Exit: No registers specified

INT 59h, Function 1 - Connect Serial Device to Modem

Entry: AH = 1 (function code)
AL = Serial port number

Exit: AL = Standard BIOS error code

The serial port number parameter in AL specifies to which serial port the modem is to be attached. If it is not possible to attach the modem to the requested port, then an error indication should be returned in AL.

It is acceptable to return an error indication in AL if the serial port requested is not configured correctly to allow modem communications to occur (for example: if the baud rate is set too high).

INT 59h, Function 2 - Disconnect Telephone Line

Entry: AH = 2 (function code)

Exit: No registers specified

This function causes the telephone line to go on hook, and terminate the telephone call.

INT 59h, Function 3 - Connect Telephone Line

Entry: AH = 3 (function code)
BX = Device to connect

Exit: AL = Standard BIOS error code

This function call takes the telephone line off hook, and connects an internal device to the line. If the line is already off hook, then it should remain so with no disruption of the call. The following device codes can be specified in BX:

0 = Telephone Dialer

1 = Modem

2 = Voice Handset

In some hardware configurations, some of these devices may be redundant. Connecting a device implies disconnecting any other device which may be connected.

INT 59h, Function 4 - Select Pulse Dialing

Entry: AH = 4 (function code)
AL = dialing rate (0 = slow, 1 = fast)

Exit: No registers specified

If the hardware does not support variable dialing rates for pulse dialing, the parameter in AL can be ignored.

INT 59h, Function 5 - Select Tone Dialing

Entry: AH = 5 (function code)
AL = dialing rate (0 = slow, 1 = fast)

Exit: No registers specified

If the hardware does not support variable dialing rates for tone dialing, the parameter in AL can be ignored.

INT 59h, Function 6 - Dial Telephone Number

Entry: AH = 6 (function code)
AL = length of dial string
DX = segment address of dial string
BX = offset address of dial string

Exit: AL = Standard BIOS error code
If an error occurs, DX:BX points to the illegal character in the dial string.

The dial string is ASCII encoded. The following characters are supported:

Digits 0 through 9 - dial the number.
and * - if tone dialing, dial the tone
if pulse dialing give error
P - switch to pulse dialing
T - switch to tone dialing
+ - delay 1 second before dialing the next digit

All other characters are illegal and cause an error to be returned.

INT 59h, Function 7 - Configure Serial Device

Entry: AH = 7 (function code)
AL = Configuration parameters
bits 0-1 - number of data bits
00 = 5
01 = 6
10 = 7
11 = 8
bits 2-3 - number of stop bits
00 = 1
01 = 1.5
10 = 2
11 = not allowed
bits 4-6 - parity selection
000 = none
001 = ignore
010 = even
011 = odd
100 = mark
101 = space
bit 7 - XON//XOFF flag
0 = XON/XOFF disabled
1 = XON/XOFF enabled
BH = XON character
BL = XOFF character
CX = baud rate/100
(except 110 baud, which is specified by CX = 0)

Exit: FLAGS.C = return code
0 - no errors
1 - illegal configuration parameter(s)

INT 59h, Function 8 - Return Receiver Status

Entry: AH = 8 (function code)

Exit: AL = Standard BIOS error code
CX = number of bytes of free space in queue
DX = number of bytes in queue

INT 59h, Function 9 - Fetch Receiver Data

Entry: AH = 9 (function code)

Exit: AL = Standard BIOS error code
FLAGS.Z = 0 - data available (data is in AL)
1 - no data available
AH = data byte (if FLAGS.Z = 0)

INT 59h, Function A - Transmit Data

Entry: AH = A (function code)
AL = data to transmit

Exit: AL = Standard BIOS error code

It is recommended that a timeout be associated with the transmitter for failsafe reasons. If the transmitter does not become ready to accept the character within the timeout period, then an error should be returned. The recommended timeout period is 250 msec.

INT 59h, Function B - Transmit Break

Entry: AH = B (function code)

Exit: AL = Standard BIOS error code

INT 59h, Function C - Configure Modem

Entry: AH = C (function code)
CH = data rate
1 = 300 baud
3 = 1200 baud
CL = mode
0 = originate
0FFh = answer

Exit: AL = Standard BIOS error code

The data rate parameter specified in CH specifies the encoding method to use.

Note: For all of the above routines, when an error is returned the BIOS must clear its error status so subsequent calls will only return an error if a new error has occurred.

INT 59h, Function D - Set Serial Control Lines

Entry: AH = D (function code)
AL = State to set control lines
Bit 0 - RTS
1 - DTR
2-7 - reserved for future use

Exit: No registers specified

This function is used to set the serial handshaking lines to a specified state. A zero in a bit position means to set the corresponding control line to a spacing (0) state, a one bit means to set the corresponding control line to the marking (1) state.

INT 59h, Function E - Get Serial Device Control Status

Entry: AH = E (function code)

Exit: AL = State of control lines
Bit 0 - CTS
1 - DSR
2 - CD
3-7 - reserved for future use

This function is used to test the state of the serial handshaking lines.

INT 59h, Function F - Select Active Serial Device

Entry: AH = F (function code)
AL = Device number to select

Exit: AL = Standard BIOS error code

This function specifies which serial device is active. The functions for configuring, reading from and writing to serial device affect the active port.

INT 59h, Function 10h - Request Serial Device

Entry: AH = 10h (function code)
AL = Device number to attach

Exit: AL = Standard BIOS error code

This function will be called before accessing a serial device. It does not select the device, but marks it as being in use. Attempting to select a serial device via function 0Fh before attaching it via this function is an error.

INT 59h, Function 11h - Release Serial Device

Entry: AH = 11h (function code)
AL = Device number to release

Exit: AL = Standard BIOS error code

This function code is called after an application is finished using a serial device. This function marks the specified device as not in use. This call does not affect which serial device is currently selected.

Bar Code Reader Support — Unsupported Software Interrupt

The data returned by the read block function should be fully decoded. There are a number of encoding schemes used with bar code readers, and the one expected to be the most common, or standard, for the particular machine should be implemented. Different encoding schemes can be supported by supplying additional bar code drivers which the user may install as needed.

If bar code reader hardware is not available, the bar code reader functions should return the Device Not Available error code when called.

The following interrupts support bar code reader input:

INT 5Ah, Function 0 - Request Bar Code Reader

Entry: AH = 0 (function code)

Exit: AL = Standard BIOS error code

This routine will be called before any access to the bar code reader. Its purpose is to mark the bar code reader as in use.

INT 5Ah, Function 1 - Initialize Bar Code Reader

Entry: AH = 1 (function code)

Exit: AL = Standard BIOS error code

This function is called before attempting to read bar code data. It should perform any initialization necessary to prepare the bar code reader to receive data.

INT 5Ah, Function 2 - Read Block

Entry: AH = 2 (function code)
CX = buffer length
DX = segment address of data buffer
BX = offset address of data buffer

Exit: AL = Standard BIOS error code
CX = number of bytes of data placed in the buffer

This function should read a block of bar code data and place it in the specified buffer. A count of the number of bytes transferred should be returned in CX.

INT 5Ah, Function 3 - Turn off Bar Code Reader

Entry: AH = 3 (function code)

Exit: AL = Standard BIOS error code

This function will be called to indicate that access to the bar code reader is complete. This function should perform any necessary clean up.

INT 5Ah, Function 4 - Release Bar Code Reader

Entry: AH = 4 (function code)

Exit: AL = Standard BIOS error code

This routine will be called at the end of bar code reader access. Its purpose is to mark the bar code reader as not in use.

Touch Panel Support — Unsupported Software Interrupt

The touch panel is considered to be an interrupt driven device. It should be scanned for a change of state at the same time that the keyboard is being scanned.

The touch panel is assumed to have a resolution equal to the size of one character on the LCD display. Touch panel coordinates are given as row and column, which corresponds to the character location on the LCD which is being touched.

It is recommended that a touch cursor be supported which highlights the location currently being touched to give the user a positive indication of the selected location on the screen. This cursor must be removed from the display when the touch panel is not being touched.

A touch panel hook is specified. This hook should be called by the BIOS whenever it has been discovered that the touch state of the touch panel has changed. A change in touch state occurs when the touch panel goes from 'not being touched' to 'being touched', or from 'being touched' to 'not being touched'.

If a touch pannel is not supported by the hardware, the following values should be returned:

AL = 0
BL = 0

INT 5Bh - Touch Panel Hook

Entry: BL = Current touch state
0 = Not touched
0FFh = Touched
DH = Column at which state change occurred
DL = Row at which state change occurred

Exit: No registers specified

INT 5Bh, Function 0 - Touch Panel Status

Entry: AH = 0 (function code)

Exit: AL = State change flag
0 = No change since last call
0FFh = State has changed since last call
BL = Current state
0 = Touch panel is not currently being touched
0FFh = Touch panel is currently being touched
DH = Column at which state changed (only if AL = 0FFh)
DL = Row at which state changed (only if AL = 0FFH)

This function is called to determine if the state of the touch panel has changed. The flag in AL indicates if the touched/not touched state has changed since the last time this function was called. If the flag in AL indicates that the state has changed, then DH-DL should contain the Column-Row at which that state change occurred. If a state change has not occurred, then DH-DL are undefined. In all cases, BL indicates the current touch state of the touch panel.

INT 5Bh, Function 1 - Touch Panel Location

Entry: AH = 1 (function code)

Exit: BL = Current state
0 = Touch panel is not currently being touched
0FFh = Touch panel is currently being touched
DH = Column
DL = Row

This function is called to determine the current state of the touch panel. The flag in BL indicates if the touch panel is currently being touched. If the panel is being touched, then DH-DL should contain the Column-Row that are currently being touched. If the panel is not currently being touched, then DH-DL are undefined.

INT 5Bh, Function 2 - Disable Touch Panel

Entry: AH = 2 (function code)

Exit: No registers specified

After this function is called, the BIOS no longer needs to scan the touch panel at interrupt level. The touch panel can still be polled via functions 0 and 1. While the touch panel is disabled, the touch panel cursor should also be disabled.

INT 5Bh, Function 3 - Enable Touch Panel

Entry: AH = 3 (function code)

Exit: No registers specified

After this function is called, the BIOS should begin scanning the touch panel at interrupt level. The touch panel cursor should also return to the enabled/disabled state that it had prior to the last call to function 2.

INT 5Bh, Function 4 - Enable Touch Cursor

Entry: AH = 4 (function code)

Exit: No registers specified

This function is called to enable the display of the touch panel cursor. The touch cursor should be displayed at the current touch position, and should be removed when the touch panel is no longer being touched.

INT 5Bh, Function 5 - Disable Touch Cursor

Entry: AH = 5 (function code)

Exit: No registers specified

This function disables the touch panel cursor so that it is not displayed when the touch panel is being touched. If a touch panel cursor is being displayed when this function is called, it must be removed from the screen.

BIOS Special Extended Functions

Following functions are provided for installable printer drivers:

INT 71h, Function 0 - Clear BIOS BREAK Flag

Entry: none

Exit: none

INT 71h, Function 1 - Check BIOS BREAK Flag

Entry: none

Exit: [CF] = 1 - BREAK
= 0 - not BREAK

INT 71h, Function 2 - Get VRAM Display Start Address

Entry: none

Exit: DX - start address

APPENDICES

Appendix A

SUMMATION OF O/S FUNCTIONS

Special Function Keys and Pop-ups

Calculator (Control F2)

From all applications run pop-up Calc under the following conditions:

a. Normal operating environment;

b. Low and 0 memory conditions;

c. While in EDIT mode.

From System Manager perform a RUN on CALC. By RUNNING Calc it is possible to pop up Calc from Calc.

Cold boot the computer and then run FILE on ENVIRON.SYS. (System Environment file). While you are doing this, pop up Calc and store a value in memory.

Alarm (Control F3)

From all applications run pop-up Alarm under the following conditions:

a. Normal operating environment;

b. Low and 0 memory conditions;

c. While in EDIT mode.

From System Manager perform a RUN on Alarm. By RUNNING Alarm it is possible to pop up Alarm from Alarm.

Set an alarm to go off with Calendar then quit Calendar. Wait for the Alarm to go off then start Calendar back up. Pop up alarm and change the Start time and/or the Reminder time. Quit Alarm to return to Calendar, the changes made to the Start time and/or Reminder time should be reflected on the screen.

Set an alarm to go off with Calendar then quit Calendar. Wait for the Alarm to go off then start File on CALENDAR.CAL. Pop up alarm and change the Start time and/or the Reminder time. Quit Alarm to return to File, the changes made to the Start time and/or Reminder time should be reflected on the screen.

User Defined Pop-ups (Control F4 - F8)

Notes on User defined pop-ups:

a. It is possible to make any valid application in memory a pop-up.

b. User definable pop-up keys are from F4 to F8.

c. By making certain applications pop-ups it forces others to contend with file sharing. One such example would be to make Calendar a pop-up, RUN Alarm then Pop up Calendar and modify the data that Alarm had displayed. Situations like this can cause nasty problems.

d. All the data needed to configure pop-ups is kept in the System Environment file (ENVIRON.SYS).

e. Remember, some pop-ups need extra memory to run so if there is a Low or 0 system memory condition in the system the pop-up may not be invoked.

Turn CALENDAR into a pop-up:

a. Run File on the System Environment file (ENVIRON.SYS).

b. Move to the NEW record in the NAME field and type the string $$FK5.

c. Move over to the TEXT field in the same row and type CALENDAR.

d. Push the enter key then Quit FILE. Wait for the System Manager to display.

e. Push the Warm Start Break button.

f. Now Push the Control F1 key to get a display of the available Pop-ups. The 5 should have the word Calendar above it.

g. Push Control F5. If the above steps were done properly, Control F5 should pop up Calendar.

h. Repeat the above steps using different Function keys (valid keys are from 4-8) and application.

Run Previous (Control F9)

Run previous between the same application.

a. Start an application on a file then Quit.

b. Start the same application on another file then perform a Run previous.

c. Perform run previous 64 times. Each time run previous is performed, wait until the screen is redrawn before performing the next.

d. Repeat the above steps for each application.

Run previous between different applications.

a. Start an application on a file then Quit.

b. Start another application on a different file then perform a Run previous.

c. Perform run previous 64 times. Each time run previous is performed, wait until the screen is redrawn before performing the next.

d. Repeat the above steps for as many application combinations as time allows for. Also try to run different applications on the same file and switch between the two. Modify the data with one application then switch back to the previous.

Programs of special concern are TELCOM, BASIC and WORD. Try opening the COM port in BASIC and then connecting to the same port with TELCOM. Also printing to COM is possible if you have a serial printer. Try opening COM in BASIC and then switching between BASIC and an application that is printing to COM. Another area is running two invocations of FILE on the same database.

Try Run previous in various low and 0 memory conditions.

Quit (Control F10)

Quit from all applications from menu options and then return to the applications using both run previous and RUN to reinvoke them. Pay special attention to Quitting with little or no available system memory.

Quit from pop-ups. This should return to the application that the pop-up was invoked from. Also pay special attention to Quitting with little or no available system memory.

Disk I/O

Try printing to the disk with various applications.

Break in mid print

Quit in mid print

Run previous in mid print

Try Merging from a disk file with Word or Loading, Chaining, Running, Merging from a disk file with BASIC.

Using a disk with physical flaws try to read from it and write to it.

If possible remove the disk from the drive during I/O and reinsert another disk that is different.

Different density

Has bad sectors

Has different number of sides

Do I/O on disks of various formats including:

Double & Single density

Double & Single sided

Different number of tracks per sector

Different physical flaws

Different physical drive location

Different file count

Device I/O

COM

This is a Read/Write device.

By configuring the port with Telcom it is possible to hook a device to any external COM port that the computer may have attached and use it as an output device for applications to print to. Note that certain signals may have to be applied to the port for data to pass through it.

PRN

This is a Write only device.

Try printing to PRN with and without a printer connected.

CON

This is a Read/Write device.

a. Read comes from the keyboard.

b. Write goes to the screen.

Print to CON using various applications. Check that the screen is redrawn after printing.

Try Merging from CON with Word or Loading from CON with BASIC.

LCD

LCD is a write only device.

All output to LCD goes to the screen. Print to LCD using various applications. Check that the screen is redrawn after printing.

Try Merging from LCD with Word or Loading from LCD with BASIC.

KYBD

KYBD is a read only device.

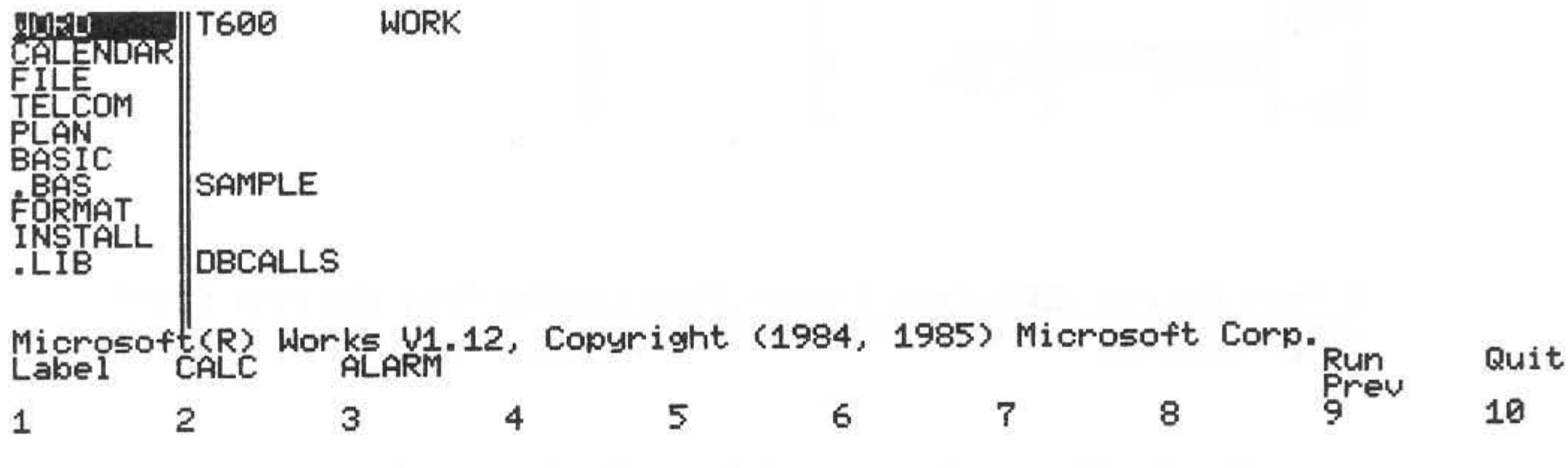
All data from KYBD comes from the keyboard. Try Merging from KYBD with Word or Loading from KYBD with BASIC.

Try to Print to KYBD with various applications.

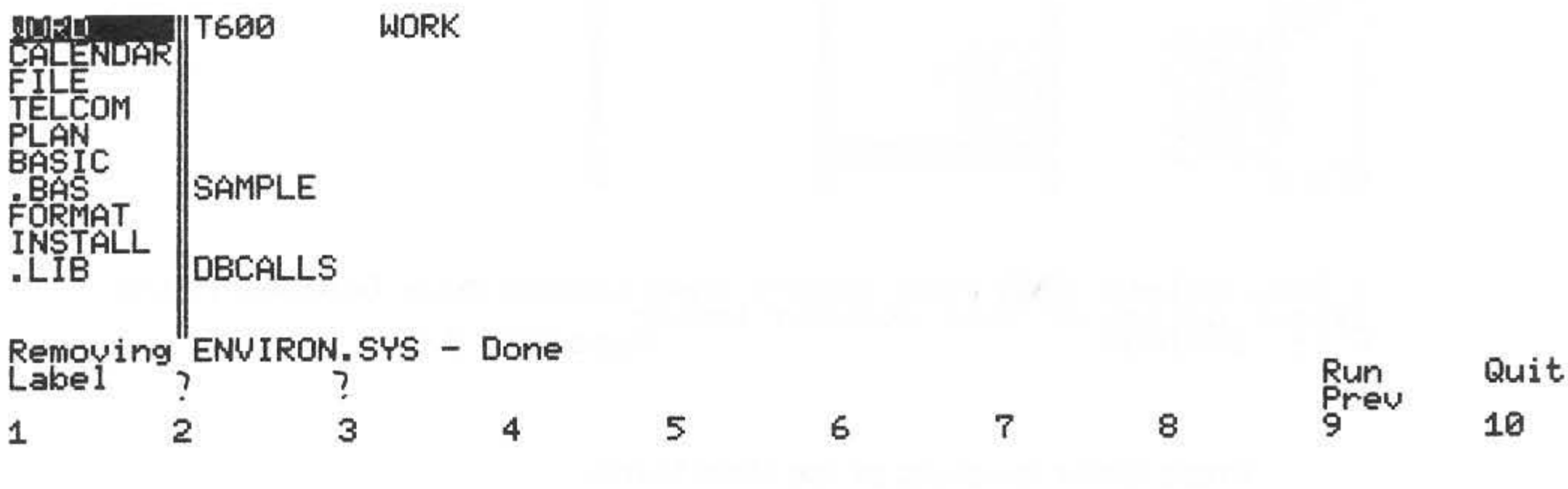
Appendix B

CHANGING MAIN MENU LABELS

Pressing Ctrl-Label will result in the following:



Removing Environ.sys file, then pressing Ctrl-Label will result in the following:



Pressing Warm-Start-Break will return the Environ.sys file, and pressing Ctrl-Label will give you same results as shown above.

To use other slots in Ctrl-Label key (create pop-ups), use the application file and the file called Environ.sys. This is what will appear.

```
ID     ‖NAME        ‖TEXT        ‖NEW
FORM   ‖AAAAAAAAAA  ‖AAAAAAAAAA  ‖AAAAAAAAAA
SORT   ‖            ‖            ‖
FIND   ‖            ‖            ‖
1      ‖$$FK2       ‖CALC        ‖
2      ‖$$FK3       ‖ALARM       ‖
NEW    ‖            ‖            ‖

> Copy Delete Edit Find Insert Jump LookUp Move Options Print
Copyright (1984, 1985) Microsoft Corp.
File: ENVIRON                                  Records: 2/2
```

Edit the file to include any of the applications, such as:

```
ID     ‖NAME        ‖TEXT        ‖NEW
FORM   ‖AAAAAAAAAA  ‖AAAAAAAAAA  ‖AAAAAAAAAA
SORT   ‖            ‖            ‖
FIND   ‖            ‖            ‖
1      ‖$$FK2       ‖CALC        ‖
2      ‖$$FK3       ‖ALARM       ‖
3      ‖$$FK4       ‖WORD        ‖
4      ‖$$FK5       ‖PLAN        ‖
5      ‖$$FK6       ‖TELCOM      ‖
NEW    ‖            ‖            ‖

> Copy Delete Edit Find Insert Jump LookUp Move Options Print
Select option or type command letter
File: ENVIRON                                  Records: 5/5
```

Press Enter to return to the Main Menu.

Now do a Warm-Start-Break to tell the system to use the new Environ.sys file.

```
WORD       ‖T600       WORK
CALENDAR   ‖
FILE       ‖ENVIRON
TELCOM     ‖
PLAN       ‖
BASIC      ‖
.BAS       ‖SAMPLE
FORMAT     ‖
INSTALL    ‖
.LIB       ‖DBCALLS

Microsoft(R) Works V1.12, Copyright (1984, 1985) Microsoft Corp.
> Copy Delete List Name Options Quit Set
Select option or type command letter
System Manager: FILE                  Bytes free: 174768  4/ 4/1985 10:20:44 AM
```

You can kill off the Environ.FIL file.

```
WORD          T600       WORK
CALENDAR
FILE
TELCOM
PLAN
BASIC
.BAS          SAMPLE
FORMAT
INSTALL
.LIB          DBCALLS

Removing ENVIRON.FIL - Done
> Copy Delete List Name Options Run Set
Select option or type command letter
System Manager: WORD                    Bytes free: 176320  4/ 4/1985 10:21:29 AM
```

Final results.

```
WORD          T600       WORK
CALENDAR
FILE
TELCOM
PLAN
BASIC
.BAS          SAMPLE
FORMAT
INSTALL
.LIB          DBCALLS

Microsoft(R) Works V1.12, Copyright (1984, 1985) Microsoft Corp.
Label    CALC     ALARM    WORD     PLAN     TELCOM                    Run      Quit
                                                                       Prev
1        2        3        4        5        6        7        8       9        10
```

To change back to normal kill the Environ.sys file and do a Warm-Start-Break.

Appendix C

PROGRAM TRANSFER AND CONVERSION

All programs (applications or device drivers) written in 8086 code for the Tandy 600 must reside in the HH-DOS file system as "pure binary fines." In other words, these files must be "core image", or in HH*DOS terms — they are "Application Memory Image" files.

Considering the existing "software tools" for the Tandy 600, this presents a major problem for the software developer: How to get a program which was written on a MS-DOS machine to the Tandy 600 and into HH-DOS's file system in the required format?

A simple printer filter for the Tandy 600 solved this problem in the following manner:

1) Write the code on a 2000 following the rules of HH-DOS's device drivers as explained in "16-Bit Hand-held Operating System Programmers Reference Guide", February 25, 1985.

2) Assemble and link using MASM and MS-LINK.

3) Then convert the .EXE file to binary using MICRO-SOFT's utility, EXE2BIN.

4) Convert the binary files to INTEL.HEX file with 16 byte records.

5) Then tranfer the .HEX file from the 2000 to the Tandy 600 via RS232. The comm package to use on 2000 is TELCOM (DESKMATE). The comm package on the Tandy 600 is also TELCOM.

6) At this point the .HEX file exists in the HH-DOS's file system. Using BASIC on the Tandy 600, modify the .HEX to binary conversion program from OKI. Convert the .HEX file to binary using this BASIC program (On the following page).

7) The INSTALL utility on the Tandy 600 is then used to make the binary file, i.e. printer filter, resident.

A sample conversion program,
Intel.hex to binary.
The intel. hex file must have 16 byte records!

```
100     KEY OFF:CLS
200     PRINT:INPUT         "Intel.hex file name: ",HNAME$
300     PRINT:INPUT         "Binary file name: ", BNAME$
400     OPEN HNAME$ FOR INPUT AS #1
500     OPEN BNAME$ FOR OUTPUT AS #2
600     '
700     H$="&H"
800             LINE INPUT #1,L$
900             PRINT L$;" > "
1000            LS=LEN (L$)-11
1100            IF LS=0 GOTO 5000
1200            L$=MID$(L$,10,LS)
1300            PRINT "                 ";L$
1400            FOR     I = 1 TO LS STEP 2
1500                    B = VAL(H$ + MID$(L$,I,2))
1600                    PRINT #2,CHR$(B);
1700            NEXT I
1800    IF LS<>0 GOTO 800
1900    '
5000    PRINT "End of Intel.hex file"
5100    CLOSE
5200    END
```

Appendix D

MS-WORKS UTILITIES FOR DEVELOPMENT

If you want a copy of the MS-Works Utilities Disk for applications development on the Tandy 600, please forward your request and a blank 5¼" Tandy 1000 disk or a PC-DOS disk (360K type) to the following address:

Tandy Third Party Support
1300 One Tandy Center
Fort Worth, Texas 76102

The following files are available on the MS-Works Utilities Disk exclusively for applications development on the Tandy 600. The use of these files is subject to the terms and conditions of the end user license (located at the beginning of the Tandy 600 Owner's Manual).

Note: These files are unsupported by Microsoft, and Radio Shack, and the simulator will not support a printer or modem.

Object Files:

HHSIM.EXE	BLDROM.EXE
CALC.!00	WORD.!30

Data Files:

DEBUG.ROM	HHOS.SYM
HHSIM.SYM	DEBUG.GEN
DEBUG.PRM	MAKEDBG.BAT
EXECNV.EXE	DEBUGENV.SYS
HHDEBUG.BAT	

The following is a listing of the $$SYS00x.sys files.

Note: These files are part of the code licensed with Microsoft Works on the Tandy 600.

$$sys001.sys : Operating system.
$$sys002.sys : Pcode Interpreter (used only by the System Manager and Microsoft applications).

$$sys003.sys : Shared Pcode Library (used only by the System Manager and Microsoft applications).
$$sys004.sys : Math Pack (used only by the System Manager and Microsoft applications).

$$sys005.sys : System Manager.
$$sys007.sys : Shared Strings (used only by the System Manager and Microsoft applications).

$$sys0010.sys : Template environment file.