HEWLETT-PACKARD

Technical Reference Manual

HP-94
Handheld Industrial Computer

HP-94 Handheld Industrial Computer

Technical Reference Manual

HEWLETT PACKARD

Edition 1 February 1987

Reorder Number
82521-90001

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Printing History **Edition 1** February 1987 Mfg. No. 82521-90002

Contents

Chapter 4 Operating System Functions

Chapter 5 Hardware Control and Status Registers

Illustrations

Part 1 Operating System

Part 2 BASIC Interpreter

Part 3 Hardware Specifications

Appendixes

Tables

Part 1 Operating System

Part 2 BASIC Interpreter

Part 3 Hardware Specifications

Appendixes

Introduction to the Technical Reference Manual

The *HP-94 Technical Reference Manual* provides software and hardware reference information about the HP-94 Handheld Industrial Computer. This information should allow software developers to write assembly language programs for controlling the HP-94 hardware resources, and hardware developers to design accessories that connect to the machine. This manual assumes a certain level of familiarity with the HP-94 and 8088 assembly language programming, and that the user will be using Microsoft assembly language development tools (MASM and LINK) or their equivalents. It is a supplement to the HP 82520A *HP-94 Software Development System* (SDS), which includes other information necessary to fully understand the product, as well as software utilities needed to convert and transfer assembly language programs to the machine. The manual is divided into four major parts:

- Operating System
- BASIC Interpreter
- Hardware Specifications
- Appendixes

The first section describes the built-in operating system, which manages and provides programmatic access to the HP-94 hardware: memory, interrupt system, keyboard, display and backlight, serial port, bar code port, internal timers, power switch and power control, low battery detection, real-time clock, and beeper. This section includes topics such as memory management, program execution, writing user-defined handlers (device drivers) for controlling the serial and bar code ports, and using operating system functions to simplify hardware control from assembly language programs.

The second section describes the internal operation of the built-in BASIC interpreter, which provides the ability to execute BASIC programs that were developed on a development system computer using the HP-94 SDS. This section does not discuss the syntax of the BASIC language, or the operation of each BASIC keyword; that information is contained in the *BASIC Language Reference Manual*. Instead, the section discusses the structure and operation of BASIC programs, data structure of BASIC variables, writing new BASIC keywords, and using BASIC interpreter utility routines to simplify the interaction of BASIC and assembly language programs.

The third section contains hardware specifications for the HP-94 in four categories: electrical (voltage and current levels, HP-94 operating conditions), mechanical (dimensions and connector pinouts), environmental (conditions under which the HP-94 will perform properly), and accessory (electrical and mechanical characteristics of plug-in cards, level converter, cables, etc.).

The final section is appendixes containing summaries of reference information for developers. This includes documentation for the utility subroutines on the disc with this manual, and for the built-in assembly language debugger.

Part 1

Operating System

Introduction to the Operating System

This section of the *HP-94 Technical Reference Manual* describes the built-in operating system, which manages and provides programmatic access to the HP-94 hardware. This section includes topics such as memory management, program execution, writing user-defined handlers (device drivers) for controlling the serial and bar code ports, and using operating system functions to simplify hardware control from assembly language programs.

This section also describes the HP-94 hardware: what major hardware elements are present in the machine, what they do, and how to operate them under software control. The major hardware elements are as follows:

- System ROM
- Read/Write Memory (RAM)
- Control and Status Registers
- CPU
- Interrupt Controller
- Keyboard
- Display with Electroluminescent Backlight
- Serial Port
- Bar Code Port
- Timers
- Power Switch
- Nickel-Cadmium (NiCd) Battery Pack
- Lithium Backup Batteries
- Real-Time Clock
- Beeper
- Reset Switch

All these items will be discussed in subsequent chapters. The following is a block diagram showing the major hardware elements and their relationships.

1

Memory Management

Contents

Chapter 1 Memory Management

1

Memory Management

This chapter describes memory in the HP-94: its possible configurations, how it is organized, and the memory management software.

Hardware Overview

The HP-94 is available in three memory configurations: HP-94D with 64K RAM, HP-94E with 128K RAM, and HP-94F with 256K RAM. Inside the 94 is a single slot for optional memory accessories. The 94D and 94E allow either the HP 82411A 40K RAM Card or HP 82412A ROM/EPROM Card (holding 32 to 128K of ROM or EPROM) to be plugged in. In addition, the 94E can be expanded to 256K (equivalent to a 94F) with the HP 82410A 128K Memory Board (service upgrade only), which also occupies the accessory slot. The 94F cannot be expanded. The following table summarizes HP-94 memory configurations.

Table 1-1. HP-94 Memory Configurations

Machine	Built-In RAM	40K RAM Card Allowed	ROM/EPROM Card Allowed	128K Memory Board Allowed
HP-94D	64K	Yes	Yes	No
HP-94E	128K	Yes	Yes	Yes
HP-94F	256K	No	No	No

The maximum total user memory in the HP-94, RAM and ROM/EPROM combined, is 256K. This limit is imposed by both hardware and software.

Software Overview

The memory management software in the HP-94 provides a directory structure for major contiguous blocks of memory, such as built-in memory and plug-in memory (RAM and ROM/EPROM cards). Within each directory is a file system that supports four different file types and files in RAM or ROM. BASIC programs (type B), assembly language programs (type A), and user-defined I/O port handlers (type H) execute in place, whether in RAM or ROM. Data files (type D) can be created and deleted dynamically while programs are running, and expand when written to in fixed- or variable-length increments. The operating system also provides for allocation and release of scratch areas, and verifies memory integrity using checksums at power off and power on.

Memory Organization

HP-94 memory is organized into contiguous blocks called directories. The directories fall into three major categories: main memory (built-in memory plus the 128K memory board), plug-in memory (40K RAM and ROM/EPROM cards), and system ROM (built-in operating system and BASIC interpreter). Each block of memory has a fixed-length table at the beginning that describes each file in that block of memory. Since the directory table is fixed-length, the maximum number of files that the directory can contain is also fixed. The directory table also identifies what type of memory it is (main, plug-in RAM, plug-in ROM) and how much memory is encompassed by the directory. Below is a table summarizing important information about HP-94 memory. followed by a memory map that shows the organization of all memory in the HP-94. Note that in the map, the main memory RAM quantities include the RAM for the smaller memory configurations, and the ". . ." indicates unused address space.

Table 1-2. Summary of Memory Information

Name of Memory Area	Memory Size	Directory Number(s)	Max. No. of Files	Min. System Overhead
Main Memory	64K	0	63	3.5K *
	128K	0	63	3.5K *
	256K	0	127	4.5K *
40K RAM Card	40K	1	31	0.5K
ROM/EPROM Card	32K	1-4	31	0.5K
	64K	1-3	31	0.5K
	96K	1-2	63	1K
	128K	1	63	1K

* If a BASIC program is running, there will be an additional 2K used by the BASIC interpreter, plus space for the data in the BASIC program variables.

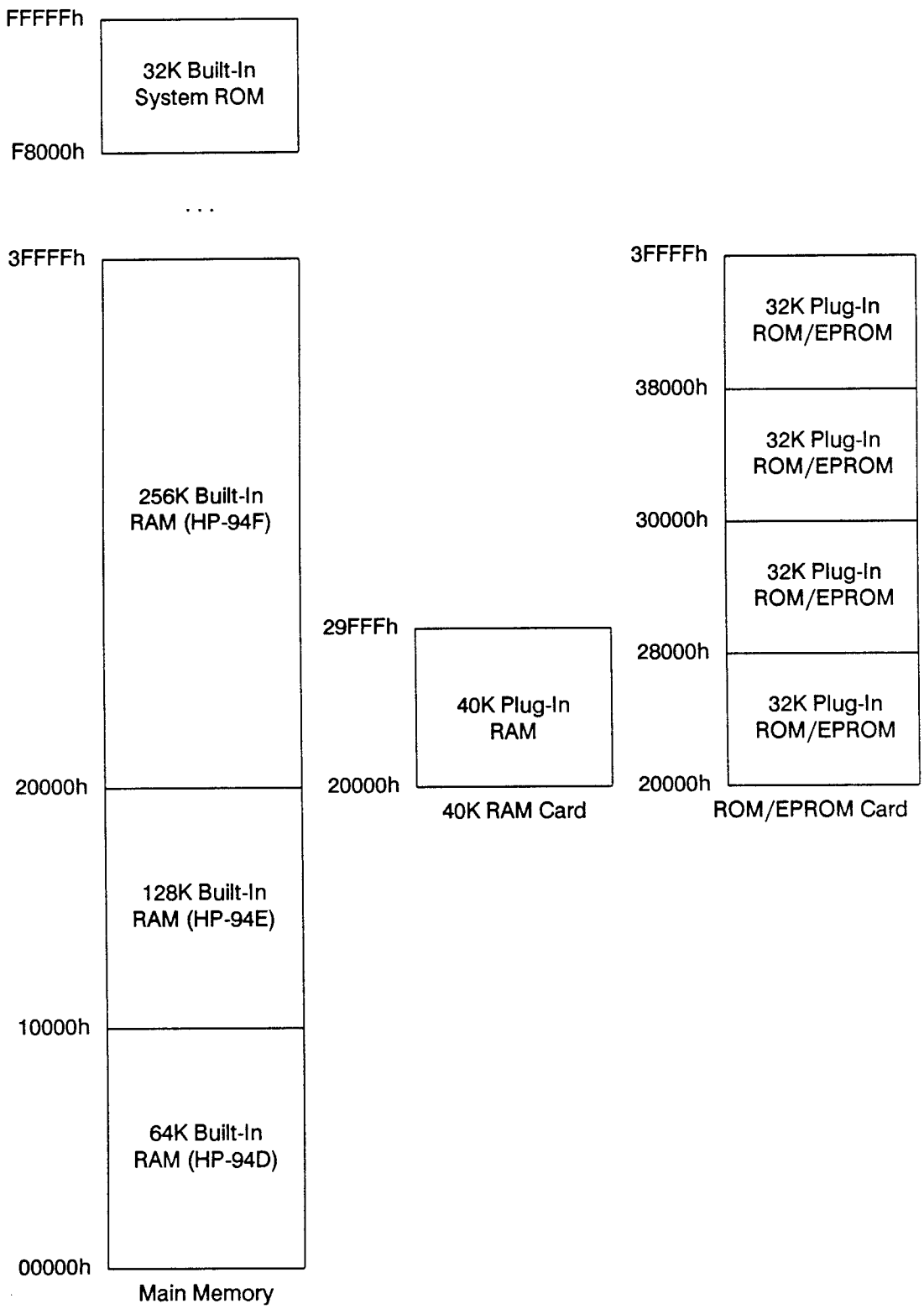


Figure 1-1. Memory Map of the HP-94

Main Memory

Main memory is the first major block of memory, and is called directory 0. It can be 64, 128, or 256K, depending on the memory configuration (94D, 94E, or 94F). Even though the 128K memory board that is used in the 94F or added to the 94E occupies the accessory slot, it is still treated as main memory because it cannot be installed or removed by the user the way the plug-in cards can. The number of files main memory can contain are 63, 63, and 127 respectively for the three memory configurations.

Below is a map of main memory. The pointers on the right side of the memory map correspond to segment addresses maintained in the directory table header (the first entry in the directory table), and will be discussed under "Directory Table".

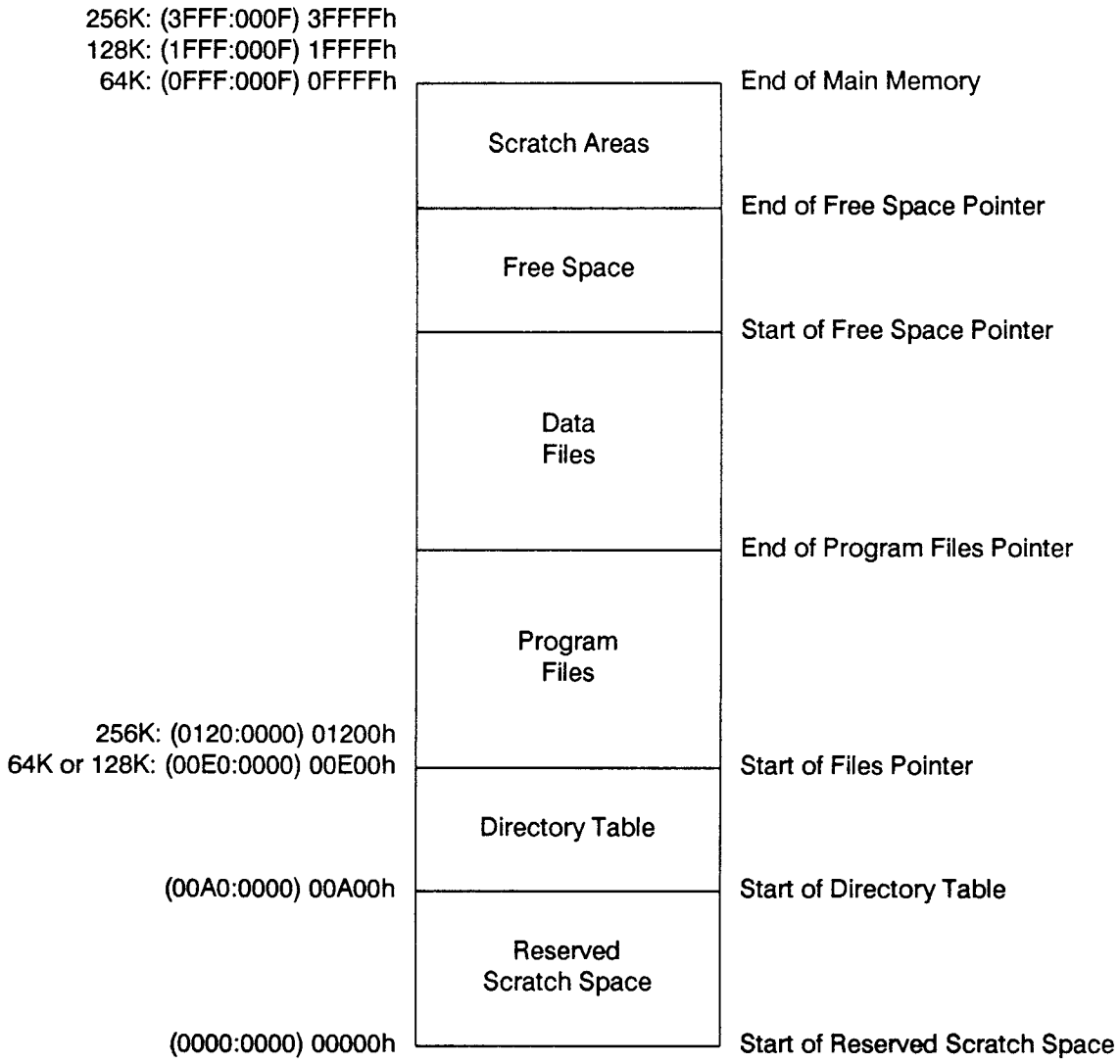


Figure 1-2. Memory Map of Main Memory

The major blocks of memory shown in the memory map are described briefly below. They will each be the subject of a separate section of this chapter.

- Reserved Scratch Space
  This area contains the interrupt vectors for the hardware and software interrupts for the CPU. This area is also used by the operating system to maintain information about the current state of the 94, and for pointers into that information. This area comprises 2.5K of the system overhead.
- Directory Table
  This block describes main memory and all the files contained in it. Files begin immediately after the end of the directory table. This area comprises 1K or 2K of the system overhead, depending on the memory configuration.
- Program Files
  This block is where all non-data files are stored; that is, file types A, B, and H. All program files appear first in the file system. The size of this block changes while programs are loaded, but does not expand or contract at run time.
- Data Files
  This block is where data files are stored. Data files expand by allocating memory from free space, expanding toward higher addresses. When data files are deleted, all their space is returned to the free space area.
- Free Space
  This block is the pool of available memory from which data files are created and expanded and scratch areas are allocated.
- Scratch Areas
  Scratch areas are requested by the built-in BASIC interpreter and by user-written assembly language programs and handlers, and are created by allocating memory from free space, building toward lower addresses. When scratch areas are released, they are returned to free space. Scratch areas are only created in main memory, regardless of which directory contains the program requesting the scratch area. They comprise any additional system overhead requirements.

40K RAM Card

The HP 82411A 40K RAM card is one of the two types of plug-in memory, and is called directory 1. It is 40K long, and can contain a maximum of 31 files. The organization of the RAM card is a subset of the main memory organization — it contains only a directory table, files, and free space. No scratch areas are available, since scratch areas are only allocated in main memory.

Here is a memory map of the 40K RAM card. The pointers on the right side of the map have the same meaning as for main memory.

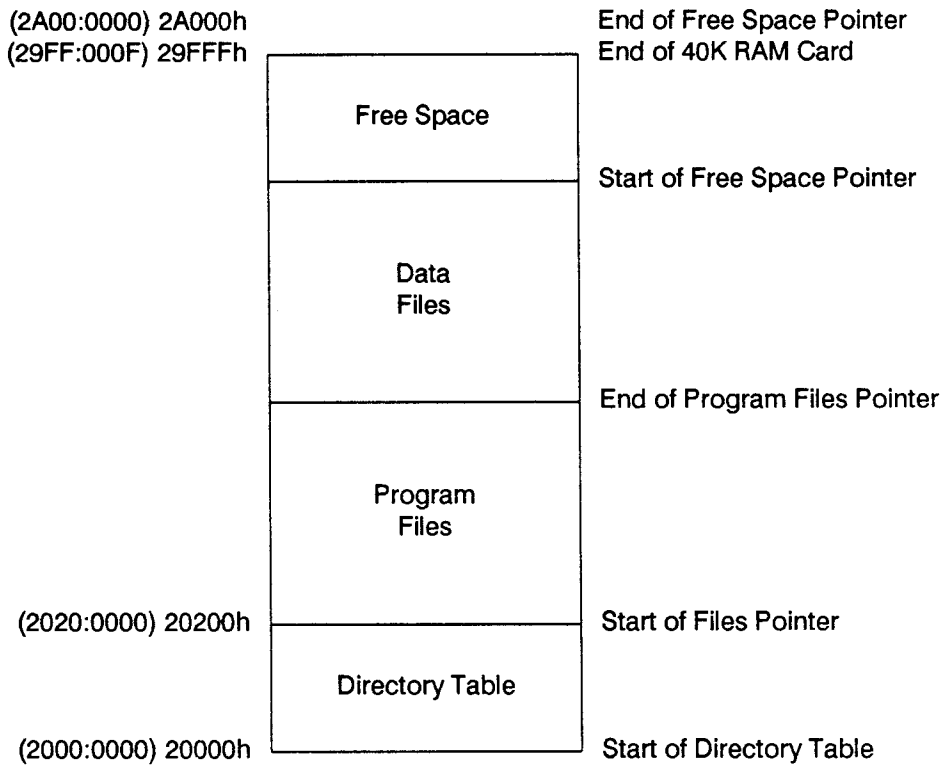


Figure 1-3. Memory Map of the HP 82411A 40K RAM Card

The major blocks of memory shown in the memory map are described below.

- Directory Table
  This block describes the RAM card and all the files contained in it. Files begin immediately after the end of the directory table. This area comprises the 0.5K RAM card overhead.
- Program Files
  This block is where all non-data files are stored; that is, file types A, B, and H. All program files appear first in the file system. The size of this block changes while programs are loaded, but does not expand or contract at run time.
- Data Files
  This block is where data files are stored. Data files expand by allocating memory from free space, expanding toward higher addresses. When data files are deleted, all their space is returned to the free space area.
- Free Space
  This block is the pool of available memory from which data files are created and expanded.

ROM/EPROM Card

The HP 82412A ROM/EPROM card is the other type of plug-in memory, and can contain directories 1 through 4. Files can be put in ROM or EPROM in blocks of four different sizes: 32, 64, 96, and 128K. The number of files each block can contain is 31, 31, 63, or 63 respectively, depending on the ROM or EPROM size. The memory map of the ROM/EPROM card will be discussed in detail under "Logical ROMs" (a *logical ROM* is a ROM in one of the different possible sizes, not necessarily related to the physical IC size actually placed on the ROM/EPROM card).

The organization of each of the four directories within the ROM/EPROM card is similar to the RAM card. They each contain only a directory table, files, and free space. No scratch areas are available, since scratch areas are only allocated in main memory (and could not be allocated in ROM or EPROM anyway).

The memory map of an individual ROM within the ROM/EPROM card is essentially the same as for the 40K RAM card. Unlike the RAM card, data files can only be read — they cannot be created, deleted, or written to. Also, the free space in a ROM or EPROM cannot be used.

The pointers that are shown on the RAM card memory map have the same meaning for an individual ROM or EPROM, but their values vary depending on the size and directory number of the ROM. This will also be discussed in "Logical ROMs".

Reserved Scratch Space

The reserved scratch space is the first 2.5K of main memory. The first 0.5K contains interrupt vectors for CPU, hardware, and software interrupts. It also contains pointers to the next 2K, which is the operating system scratch space. Here is a memory map of the reserved scratch space. The ". . ." indicates unused interrupt vector locations.

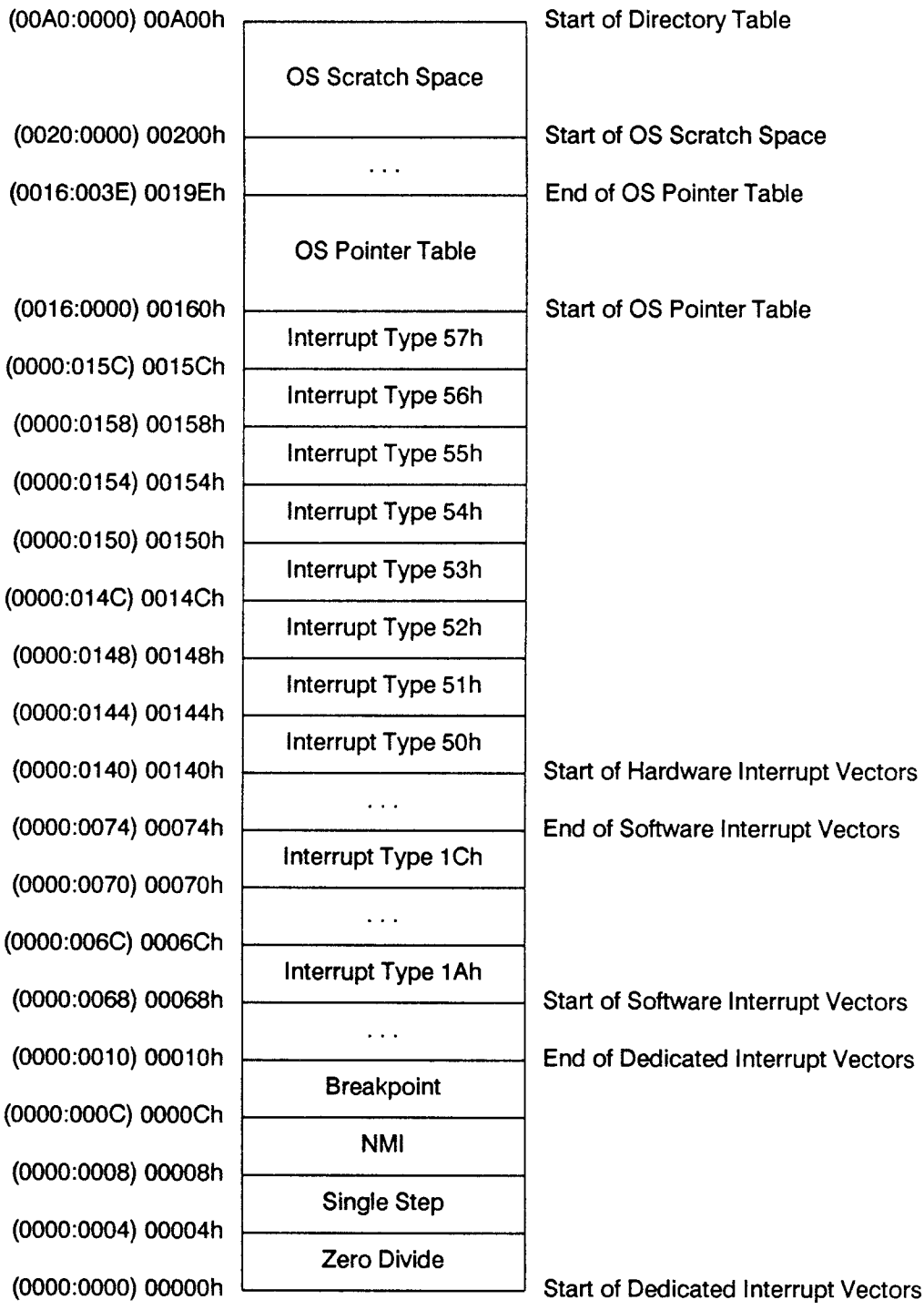


Figure 1-4. Memory Map of Reserved Scratch Space

The major items in the reserved scratch space are described below. The information at the end of each description are the chapters or appendixes where further information can be found about that interrupt. General information about the hardware interrupts (types 50h-57h) is in the "Interrupt Controller" chapter.

- Zero Divide
  Dedicated interrupt vector for divide-by-zero condition. Points to the same location as the breakpoint interrupt vector (appendix A).
- Single Step
  Dedicated single step interrupt vector used for single-stepping the resident debugger (appendix A).
- NMI
  Dedicated non-maskable interrupt vector used to invoke the resident debugger. Points to the same location as the breakpoint interrupt vector (appendix A).
- Breakpoint
  Dedicated breakpoint interrupt vector used for breakpoints in the resident debugger (appendix A).
- Interrupt Type 1Ah
  Software interrupt vector used to invoke the operating system functions (chapter 4).
- Interrupt Type 1Ch
  Software interrupt vector used for the one-second background timer (chapter 12).
- Interrupt Type 50h
  Hardware interrupt vector for system timer (chapter 12).
- Interrupt Type 51h
  Hardware interrupt vector for bar code port timer (chapters 11 and 12).
- Interrupt Type 52h
  Hardware interrupt vector for bar code port transitions (chapter 11).
- Interrupt Type 53h
  Hardware interrupt vector for serial port (82C51 data received) (chapter 10).
- Interrupt Type 54h
  Hardware interrupt vector for low main battery voltage (chapter 14).
- Interrupt Type 55h
  Hardware interrupt vector for power switch pressed (chapter 13).
- Interrupt Type 56h
  Reserved hardware interrupt vector 1 (chapter 7).
- Interrupt Type 57h
  Reserved hardware interrupt vector 2 (chapter 7).
- OS Pointer Table
  These are pointers to various parts of the operating system scratch space. The main pointer of interest to assembly language programmers is the one that points to the handler information table. Refer to the "User-Defined Handlers" chapter for details.
- OS Scratch Space
  This is the space in which the operating system keeps important information about the current state of the HP-94. This area is 2K long. The operating system stack is in this area. It varies in length as it is used, up to a maximum of approximately 600 bytes.

CAUTION The operating system does not initialize or use the overflow interrupt (dedicated interrupt vector 04h, at address 04h * 4 = 00010h). A program that uses the INTO instruction (*interrupt on overflow*) must initialize this interrupt vector to a location in its own program space.

Directory Table

The directory table is organized as a series of 16-byte entries, one per file. The first entry is the directory table header. It identifies the directory, the type of memory (main memory, 40K RAM card, or ROM/EPROM card), and the total amount of memory encompassed by the directory. The header also contains the *pointers* shown on the memory maps. Since all memory areas start and end on paragraph boundaries (a *paragraph* is a block of 16 bytes), pointers are stored in the directory table as segment addresses only.

The contents of the directory table header are shown below. The numbers on the left are hex offsets relative to the start of the header.

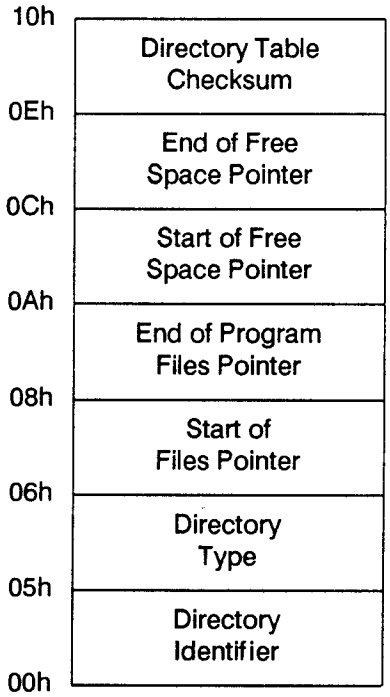


Figure 1-5. Directory Table Header Contents

Refer to the memory maps to see the areas of memory that the pointers refer to.

- Directory Identifier
  The directory identifier always contains the characters `*DIR*`. The operating system uses this to help verify memory integrity.

- Directory Type
  The directory type is the character `M` for main memory, `A` for a 40K RAM card, or `O` for a ROM/EPROM card.

- Start of Files Pointer
  This segment address points past the end of the directory table, and is the beginning of all files. Program files always appear first in the file system.

- End of Program Files Pointer
  This segment address points past the end of the program files, which is the beginning of the data files. Nothing below this address within the directory will move at runtime.

- Start of Free Space Pointer
  This segment address points past the end of the data files, which is the beginning of the free space. Free space is used for data files and scratch areas in main memory, for data files only in a RAM card, and is not available for use in a ROM or EPROM.

- End of Free Space Pointer
  This segment address points past the end of free space. For main memory, it also marks the beginning of scratch areas available for assembly language programmers. If no scratch areas have been allocated, this pointer points past the last byte in main memory — to 1000:0000 (64K), 2000:0000 (128K), or 4000:0000 (256K).

  For the 40K RAM card, this pointer points past the end of the card, since there are no scratch areas. For the same reason, in a ROM, this pointer points past the end of the logical ROM.

- Directory Table Checksum
  This is where the checksum of the directory table is saved when the machine is turned off.

The other entries in the directory table identify the different files. The contents of the directory table entries for files is shown below. Again, the numbers are hex offsets from the start of the entry.

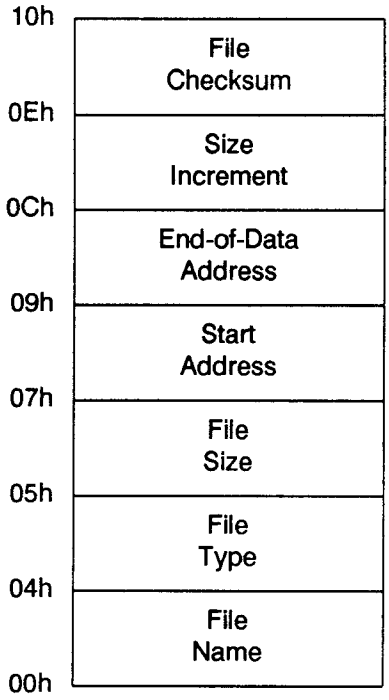


Figure 1-6. Directory Table Entry Contents

- File Name
  This is the name of the file. File names are 1-4 characters long, padded with blanks. If the file had a checksum error at power on, the high bit is set in the first character of the file name (except in ROM files). If a directory table entry is unused, the first byte of this field is set to null (00h).
- File Type
  This is either an A, B, D, or H.
- File Size
  This is the current length of the file in paragraphs. All files are padded with nulls (00h) to the nearest paragraph boundary.
- Start Address
  This segment address is the location where the file starts.
- End-of-Data (EOD) Address
  For data files, this is the offset of the end-of-data within the file, relative to the start of the file. For program files, this is a pointer to the end of the program, which may not be the end of the file because of the null padding. The EOD address is a 24-bit value stored as a two-byte offset and a one-byte segment (low word followed by high byte).
- Size Increment
  For data files, this is the expansion increment, in paragraphs, used when data is written past the end-of-file. It is 0 for program files in RAM and for all files in ROM.

- File Checksum
  This is where the checksum of the file is saved when the machine is turned off.

The space reserved for the directory table is fixed-length, and varies with the total amount of memory. Because the first entry is always reserved for the directory table header, there will be space for one less user file than the size of the directory table would otherwise indicate. The directory sizes and number of files available are shown below.

Table 1-3. Directory Table Sizes

Name of Memory Area	Memory Size	Directory Table Size	Number of Files
Main Memory	64K	1K	63
	128K	1K	63
	256K	2K	127
40K RAM Card	40K	0.5K	31
ROM/EPROM Card	32K	0.5K	31
	64K	0.5K	31
	96K	1K	63
	128K	1K	63

File System

The HP-94 file system allows for multiple files of different types to coexist simultaneously. User files can reside in any of the five user directories (0-4), whether RAM or ROM.

File Names

Each file is identified by a 1-4 character name. File names are composed of uppercase alphabetic characters and numbers only, and must start with a letter. A file name can only exist once in any directory. It is not possible to have the same name but a different type in the same directory. However, the same file name can exist in different directories, with either the same or different type.

File Types

There are four possible file types:

- Assembly Language Program — Type A
  Assembly language programs are either new BASIC keywords, invoked with the %CALL statement, or are entire assembly language applications.
- BASIC Program — Type B
  BASIC programs are a collection of "tokens" that are can be executed by the BASIC interpreter. They are produced by HXC from a BAS file during the file conversion process.

- Data File — Type D
  Data files are simply contiguous blocks of memory.
- User-Defined Handler — Type H
  A handler is a special assembly language program that controls the I/O ports, such as the serial and bar code ports. It has a structure similar in concept to a UNIX or MS-DOS device driver.

Erasing and Loading Files

When files are erased from command mode with the E (*erase*) operating system command, their memory is returned to free space, and files higher in memory move down to fill in the hole. When files are loaded with the C (*copy*) operating system command, existing files with the same name are erased first, and the memory they occupied is reclaimed for other uses. Then memory for the new file is allocated from free space (assuming there is enough room). This ensures that neither file space nor free space are fragmented while erasing or loading files. When data files are deleted with the DELETE function (14h), the memory they occupied is also reclaimed.

Reserved File Names

There are four files with reserved names that must not be used for anything except their current use:

- SYBI — built-in BASIC interpreter
- SYBD — BASIC debugger
- SYFT — user-defined font
- SYOS — built-in operating system

When the BASIC interpreter searches for user-defined keywords with %CALL, the 12 built-in keywords starting with SY will be *not* be overridden by new keyword files of the same name (SYAL, SYBP, SYEL, SYER, SYIN, SYLB, SYPO, SYPT, SYRS, SYRT, SYSW, and SYTO).

In general, Hewlett-Packard uses SY as the first two characters of all its assembly language utilities, and HN as the first two characters of all its user-defined handlers. If you use file names starting with SY or handler names starting with HN, you may have a name conflict. Consequently, you should not use names starting with those characters.

Maximum Number of Files

The maximum number of files that can be placed in any directory was indicated in "Memory Organization" and "Directory Table". The maximum total number of files would occur in a 94D or 94E with a ROM/EPROM card containing four 32K ROMs — 63 files for main memory plus 4 * 31 files for the ROM/EPROM card, for a total of 187 files.

Data Files

Data files are contiguous blocks of memory with a 1-4 character file name name, and file type D. They have no explicit record structure associated with them — it is the responsibility of the application program to impose any record structure needed, and read and write data from the appropriate position within the file. They always appear after all program files in whichever directory the data file resides — between the end of program files pointer and the start of free space pointer.

Data files are created using the CREATE function (11h). When a data file is created, the space requested is taken from free space at the end of the current data files, the directory table header pointers are adjusted, and one entry in the directory table is used to identify the file. Once a file is created, it must be opened with the OPEN function (0Fh) before data can be read or written. Data files are automatically closed at cold start. Data files that were open when the machine was turned off remain open at warm start.

Data files have two characteristics that are defined by the program that creates them (*file size* and *size increment*) and two that are defined automatically (*end-of-data address* (EOD) and *file access pointer*).

File Size

This is the initial size of the file, which is the amount of memory that will be reserved for the file when it is created. It is specified in paragraphs and ranges from 0000h to FFFFh (although the maximum file size is limited by available memory). The space used for the file is automatically initialized to all nulls (00h). A file size of 0 means that the file initially occupies no space, even though the directory table entry still exists to identify the file.

Data files cannot be created in a ROM or EPROM, or in any read-only directory (main memory or the 40K RAM card may be set to be read-only if a checksum error occurred in their directory tables at power on).

Data files can also be created on the development system. Like all development system files, they are converted to Intel MDS format by HXC for transmission to the 94. When no file size is specified, HXC automatically sets it to the actual file size on the development system, rounded up to the nearest paragraph boundary. The 0 to 15 bytes needed to pad the file are automatically set to nulls (00h).

For RAM data files, HXC allows specifying a file size that is larger than the actual size. That way a file could be defined to have a certain amount of data in it, and a fixed amount of unused space in the file. This option is not available for ROM data files, since a program cannot write to unused space in a ROM or EPROM.

Size Increment

This is the expansion increment used to increase the file size when the WRITE function (13h) attempts to write past the end of the file (that is, when the current file size is exceeded). It is specified in paragraphs, and ranges from 0000h to FFFFh (although the maximum expansion is limited by available memory). When a program writes to a data file, and there is no room for the data being written, the operating system will attempt to expand the file by the number of size increments needed, and then the data will be written to the file. For example, a file with a size increment of three (3) paragraphs will expand by as many three-paragraph blocks of memory (48 bytes) as needed to accommodate the

data being written.

Note that the 94 may run out of memory during any of the expansions, leaving a file that has been expanded, but not enough to hold the data to be written. In this situation, no data will be written to the file — data is only written to a file if there is enough room for all it.

When a data file expands, all data files higher in memory move up to accommodate the increased file size. This is illustrated below.

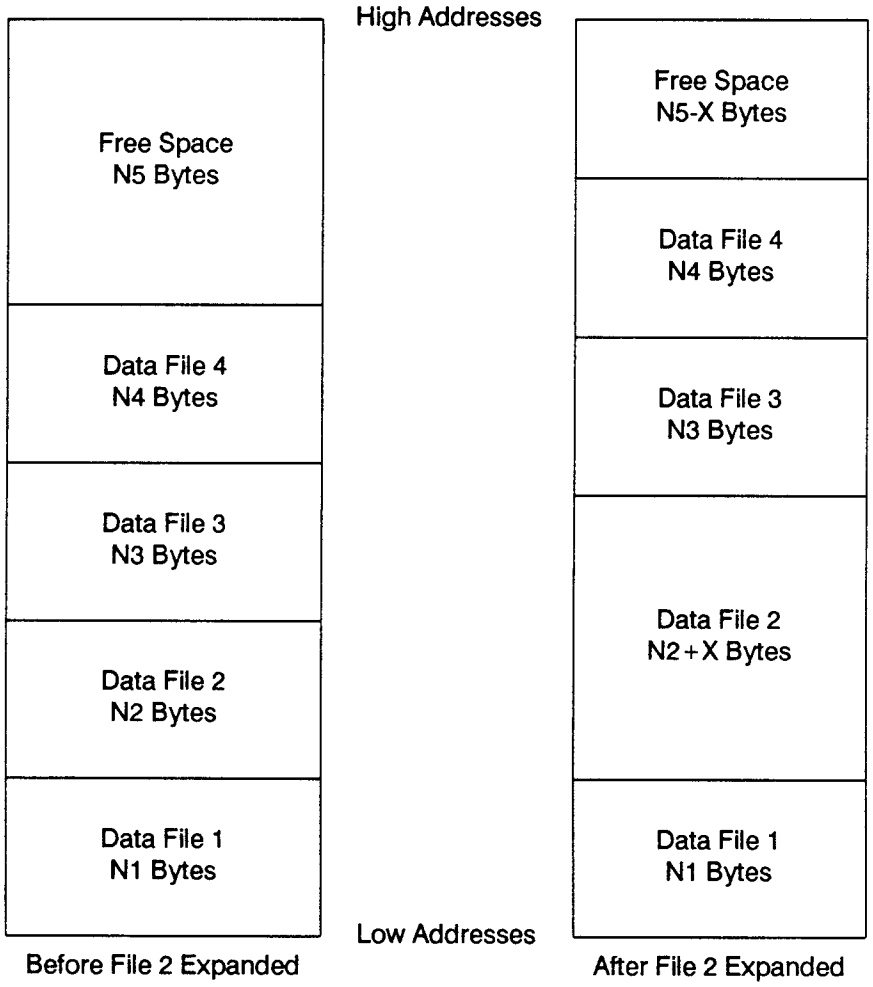


Figure 1-7. File Movement During Data File Expansion

Expansion space added to the file is automatically initialized to all nulls (00h). A size increment of 0 means no expansion will take place — the file will never grow past its allocated size. A size increment of 0 can be specified for any RAM data file; HXC automatically sets it to 0 for ROM data files, since they cannot expand.

File writes are not buffered — they immediately modify the file, provided space is available.

End-of-Data Address

The EOD address is a pointer in the directory table to the location in the data file just past the last byte of data. It is usually not equal to the end of the file (EOF) because files always end on a paragraph boundary. For data files from the development system, HXC sets the EOD address past the last byte of data, even if there is padding to the paragraph boundary or unused space specified beyond the actual file size.

Every time a file write operation writes data past the current EOD or EOF, the EOD is automatically adjusted to reflect the new end-of-data location.

File Access Pointer

This is the single pointer to the current read/write position in the file. The pointer is set to 0 (the start of the file) when the file is opened, and is updated after every file read or write operation. Every time a read or write occurs, the pointer is changed to point past the last byte read or written. Subsequent file read or write operations will begin reading or writing from that updated position. The pointer can be explicitly moved to an arbitrary position between the start of the file and the EOD, or set to the EOD by using the SEEK function (15h). Moves beyond the EOD give an error. It is also possible to force the EOD to be equal to the current file access pointer by performing a zero-length write using the WRITE function (13h). This renders any data after that point inaccessible, but does not collapse the file.

Deleting Data Files

Data files are deleted with the DELETE function (14h), and must be open before they can be deleted. When data files are deleted, all the space occupied by the file is returned to free space. All data files higher in memory move down to fill in the hole. The file space is then available for new data file creation, data file expansion, or scratch area allocation.

Interrupts During File Operations

The power switch and low battery interrupts are disabled during file create, read, write, and delete operations, so they are guaranteed to complete and not be corrupted (unless the reset switch is pressed or the machine turns off automatically because of very low battery). The interrupts are reenabled after the file operation is completed. This disabling and enabling does not change the interrupt status defined by the SET_INTR function (0Ah). What it does is defer the processing (or ignoring) of those interrupts until after the file operation has been completed.

The system timeout only occurs during read operations for channels 0-4 and read/write operations for channels 1-4, so it will not occur during file operations, which use channels 5-15.

File Expansion Example

Assume a data file exists with a current size of 2 paragraphs (32 bytes) and a size increment of 3 paragraphs (48 bytes). The file already contains 25 bytes of data, leaving the EOD at offset 25 relative to the start of the file (the first byte of the file is at offset 0, and the EOD points past the last byte of data). For this example, assume the file access pointer is also at EOD.

When a program tries to write 66 bytes at the file access pointer, there is no room — there are only 7 bytes available. The amount of space required is 66 - 7 = 59 bytes, or 4 paragraphs. Since the size increment is 3, two expansions of 3 paragraphs each will be performed, with a resulting file size of 2 + 2 * 3 = 8 paragraphs (128 bytes). Once the expansion has been completed, the data will be written. The EOD (and the file access pointer) will be moved to offset 25 + 66 = 91, leaving 37 bytes of unused space available at the end. This change to the data file is illustrated below (both decimal and hex offsets are shown).

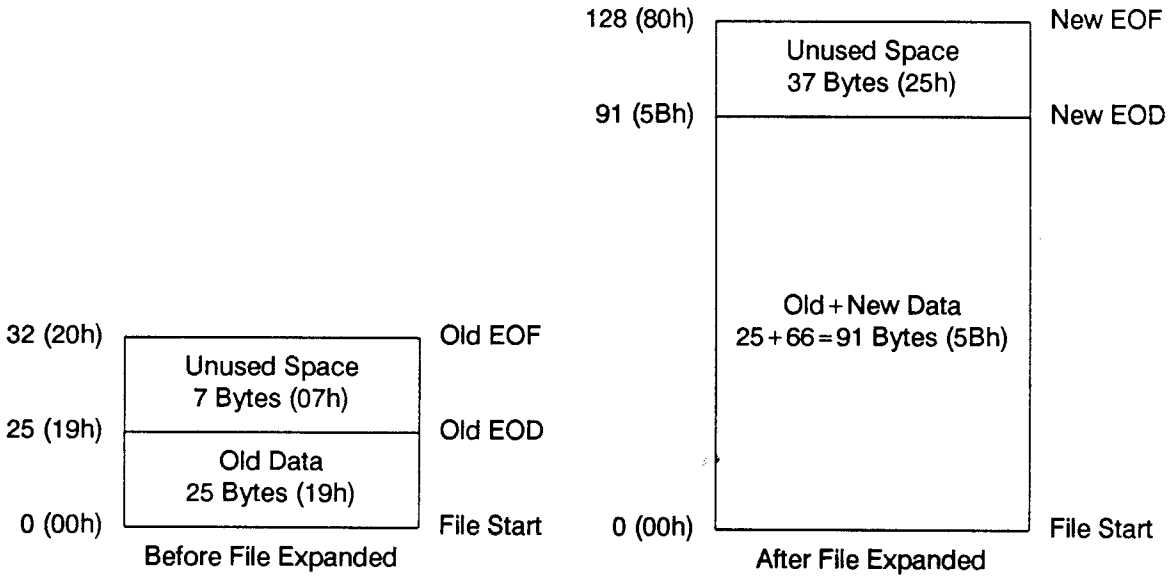


Figure 1-8. Example of Data File Expansion

If the file access pointer had been at the start of the file before the write operation, only a single 3-paragraph expansion would have been needed to accommodate 66 - 32 = 34 bytes.

Free Space

Free space is the pool of available memory from which data files are created and expanded in RAM (main memory and 40K RAM card) and scratch areas are allocated (main memory only). Free space is not available for any use in a ROM or EPROM. It starts at the start of free space pointer in any directory, which is the end of all data files, and ends at the end of free space pointer, which will be at the end of the directory (for main memory only, it could also be at the start of the scratch areas).

In any directory, data files are created and expand by allocating the required memory from the bottom of free space, expanding toward higher addresses. In main memory, scratch areas are created by allocating the required memory from the top of free space, building toward lower addresses, as shown below.

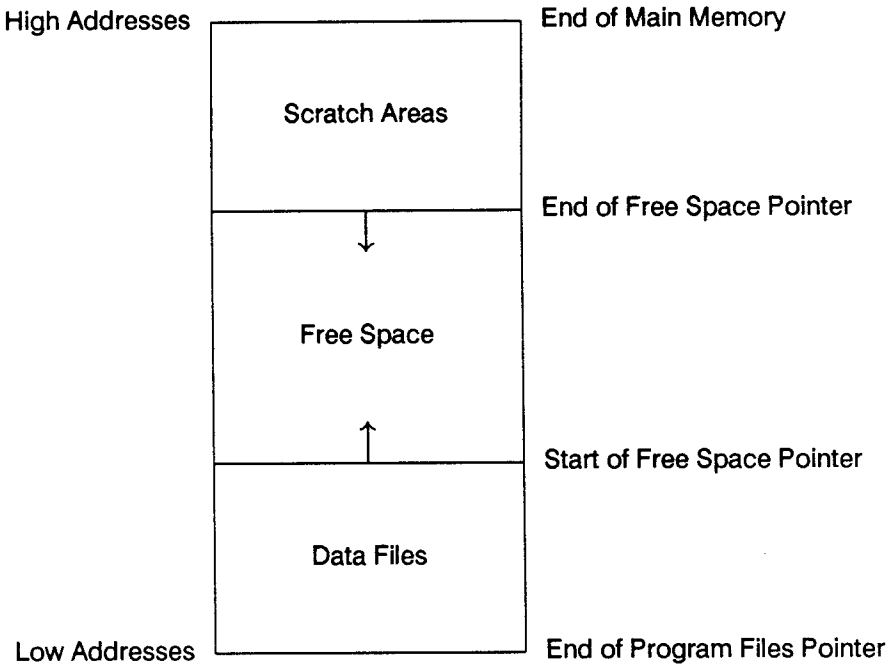


Figure 1-9. Use of Free Space in Main Memory

When the free space goes to zero from either direction, the 94 is out of memory. No data files can be created or expanded, and no more scratch areas can be allocated. The ROOM function (0Eh) reports the amount of free space in any directory; in main memory, it will take into account any existing scratch areas.

Usage in Command Mode

Whenever the operating system enters command mode, all scratch areas in main memory are eliminated, allowing the free space in directory 0 to extend to the end of main memory. The available memory for all directories is then just the size of the free space.

When any RAM file is erased with the E (*erase*) command, the space occupied by that file is returned to free space, and all files higher in memory, regardless of type, are moved down to fill in the hole. When a new file is loaded using the C (*copy*) command, a previously existing file with the same name is erased, and the memory it occupied is reclaimed. Then space for the new file is allocated from free space, and the new file is loaded. If the file loaded is a program file, all files above the end of program files pointer are moved up to make room for the program. If the file loaded is a data file, it is added at the end of the existing data files, and other files do not need to move.

Usage at Run Time

During a running program, there may be scratch areas allocated in main memory, so free space in directory 0 extends only up to the start of the scratch areas. The available memory for other directories is still just the size of the free space.

At run time, program files do not move — only data files and scratch areas interact with free space at run time. When a RAM data file is deleted programmatically, the space occupied by that file is returned to free space, and all data files higher in memory are moved down to fill in the hole. When a new data file is created programmatically, its memory is allocated from free space at the end of the existing data files. When a data file expands because of a write past its end-of-file, the expansion space is allocated from free space, and all data files higher in memory are moved up to make room for the expanded file.

When a scratch area is created, its memory is allocated from free space. When scratch areas are released, their memory is returned to free space only if the area is adjacent to the top of free space. See "Releasing Scratch Areas" for more details.

Scratch Areas

Scratch areas are blocks of memory that a program can reserve for its own use. The built-in BASIC interpreter allocates scratch areas to hold BASIC program variables and subprogram calling information. User-written assembly language programs and user-defined handlers can allocate scratch areas for parameters, status, configuration information, buffer space, space for data returned by operating system functions, or whatever other purpose is required.

Allocating Scratch Areas

The operating system GET_MEM function (0Bh) provides the ability to allocate scratch areas in sizes from 0001h to FFFFh paragraphs (although the maximum expansion is limited by available memory), and returns the segment address of the scratch area. Scratch areas are allocated in main memory only, regardless of which directory contains the program requesting the scratch area: directories 0-4, RAM or ROM. Scratch areas start at the end of main memory and use the space required from free space, building down toward lower addresses. They can also use previously-released scratch areas that have not been returned to free space. This will be discussed later.

Scratch areas are automatically initialized to all nulls. They are all released at cold start, but are preserved at warm start.

When a handler allocates a scratch area during its OPEN routine, the operating system saves the scratch area address in a table based on the channel number of the handler. When the other routines in the handler are called (such as READ, WRITE, etc.), the operating system passes the scratch area address to the routine. (The handler must save this address in the handler information table if it will be needed for an interrupt service routine.)

If a handler allocates more than one scratch area, only the address of the last one allocated will be saved and automatically passed to handler routines. Therefore, when multiple scratch areas are allocated by a handler, the allocation order is important. A handler can allocate scratch areas so that the last one allocated is the one whose address should be passed to handler routines. Alternatively, the

handler can call GET_MEM with the channel number set to 0, and the operating system will not save that scratch area address or pass it to handler routines.

When an assembly language program allocates scratch areas, it is responsible for keeping track of the locations of its scratch areas. The operating system saves scratch area addresses only for user-defined handlers.

The assembler provides the ability to define the offsets within an external scratch area using the SEGMENT AT directive, as shown below.

```
SCR_AREA        segment at 0            ;Addresses start at 0
PARAM1          db      6 dup(?)        ;First parameter needs 6 bytes
PARAM2          db      00              ;Second parameter needs a byte
PARAM3          dw      0000            ;Third parameter needs a word
SCR_AREA        ends
```

Figure 1-10. Defining Scratch Area Data Structure

The SEGMENT AT directive provides an address template that can be imposed on the scratch area. SEGMENT AT causes no code to be generated for the uninitialized data defined within that program segment (in this case, the SCR_AREA segment).

Releasing Scratch Areas

Scratch areas are released using the REL_MEM function (0Ch). The program supplies the address of the scratch area to be released. An error will occur if the program tries to release a scratch area that does not exist by supplying an address that does not point to any defined scratch area.

When a scratch area is released, the operating system will attempt to return the area to free space. This can only occur if the scratch area is adjacent to free space. Consequently, it may not be possible to return a scratch area to free space because of the order that the scratch areas were allocated.

For example, if a handler is opened in a BASIC subprogram, and allocates a scratch area, the area will be adjacent to free space, and will be lower in memory than the scratch area allocated by the subprogram for its variables. When the subprogram ends, the scratch area used for its variables will be released, but will not be returned to free space. It is blocked from being adjacent to free space because of the handler's scratch area. This area is flagged as a free block, available for scratch area allocation, but not for data file creation or expansion since it is not part of free space.

In the diagram below, scratch area 3 was allocated for variables for a BASIC subprogram, and scratch area 4 by a handler.

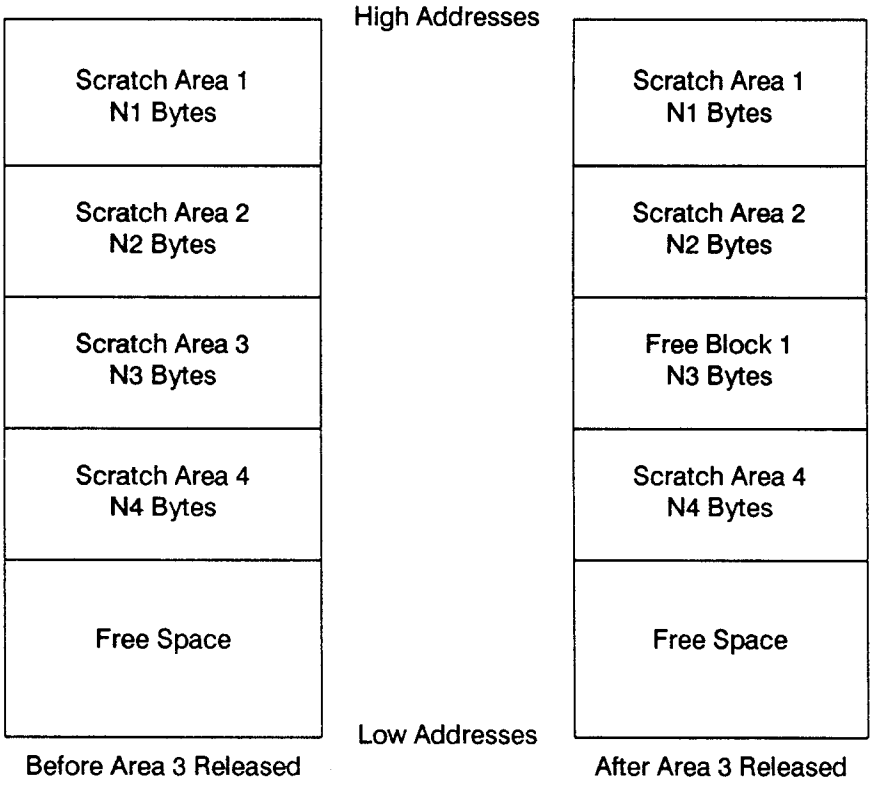


Figure 1-11. Blocking a Released Scratch Area

Scratch area 4 prevents released scratch area 3 from being returned to free space. Scratch area 3 becomes the first free block. It will not be returned to free space until scratch area 4 is released.

To allow this newly-available free block to be reused, regardless of the order in which scratch areas were allocated and released, it will be combined with any adjacent free blocks formed when other trapped scratch areas were released. This coalescing process attempts to form a few large available free blocks, rather than many small ones. This is illustrated below.

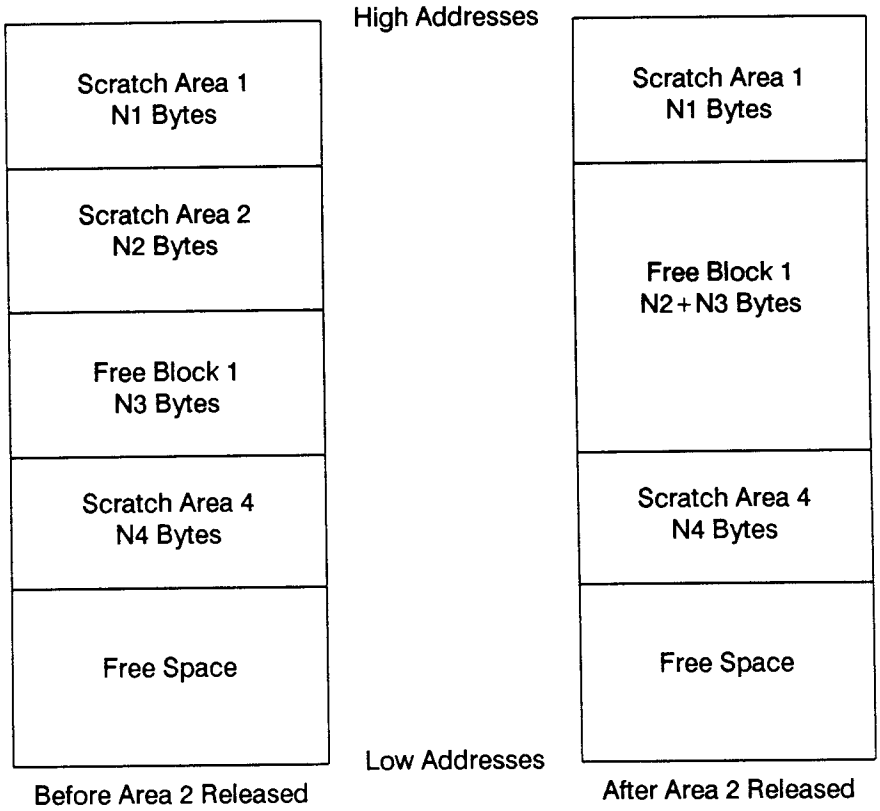


Figure 1-12. Coalescing Adjacent Released Scratch Areas

When scratch area 2 is released, it forms a new free block that cannot be returned to free space. The coalescing process combines this new block with free block 1 that already exists, forming a single free block whose size is the sum of the two smaller blocks. This keeps the number of free blocks to a minimum, since the operating system can only keep track of 20 free blocks.

Subsequent allocation of new scratch areas will use the first free block that is large enough among all those available before allocating additional memory from free space. Only as much of the free block will be used as is required. The remainder will be flagged as a smaller free block.

Data files cannot use free blocks until they are returned to free space — only scratch areas can reuse free blocks. Consequently, free space can go to zero and leave no room for data files creation or expansion, even though there may be free blocks available for reuse when allocating scratch areas.

There is no facility to pack the free blocks together, since many tables and handlers keep track of the segment address of the their scratch areas. Only allocation and release of scratch areas in careful order can help prevent fragmentation of free blocks.

After the coalescing has been completed, if there is an available free block adjacent to free space, it is returned to free space for other uses (data file allocation and expansion or new scratch area allocation when the available free blocks are not large enough).

When the 94 cold starts, all scratch areas and free blocks are automatically returned to free space. This will occur the next time the machine is turned on after a program calls the END_PROGRAM function (00h) and specifies a subsequent cold start. This also occurs whenever the operating system enters command mode, whether because of a program error or because of an explicit call to END_PROGRAM. If a program calls END_PROGRAM and specifies a subsequent warm start, all scratch areas and free blocks are preserved the next time the machine is turned on.

Number of Scratch Areas

A maximum of 34 scratch areas can be allocated in main memory. An error will occur when a scratch area is allocated if 34 scratch areas are already in use.

The BASIC interpreter allocates scratch areas for its own use, for BASIC variables, and for control information. In this sense, the BASIC interpreter can be thought of as another assembly language program, using the facilities within the operating system for scratch space management.

When a BASIC main program is run, two scratch areas are allocated immediately:

- One scratch area for the BASIC interpreter scratch space (2K long).
- One scratch area for the BASIC program variables. The length of this area is shown as "Variable Space Required" in the BMP file produced by HXC (although the length is rounded up to the nearest paragraph boundary). This area will not be allocated in the case of a BASIC main program with no variables.

This leaves a total of 32 scratch areas available for other uses. After that, every time a BASIC subprogram is called with the CALL statement, two scratch areas are allocated:

- One scratch area for the control information save area that contains information passed between programs (32 bytes).
- One scratch area for the BASIC subprogram variables (length shown in the BMP file, not allocated if no variables).

This is why BASIC subprograms can only be nested a maximum of 16 levels deep — scratch area allocation limits permit 32 scratch areas beyond those used for the main program.

Fewer scratch areas may actually be available for BASIC subprogram nesting, since user-defined handlers and assembly language programs can allocate scratch areas also. A high-level and low-level handler combination, for example, may have three scratch areas allocated between them: one for configuration passing and two for scratch and buffer space (one for each handler). Assembly language programs generally allocate one scratch area for scratch and buffer space, but may allocate a second one for configuration passing to handlers. Consequently, BASIC subprogram nesting may be restricted to less than 16 levels.

Optimum Memory Use With Scratch Areas

To allow the most efficient use of memory, scratch areas should be allocated and released in such a way that they do not block other scratch areas from being returned to free space. Long-term scratch areas that must remain in place throughout program execution (such as handler scratch areas) should be allocated when the program begins executing. Short-term scratch areas should be released as soon as they are not needed.

This is particularly important for BASIC programs. BASIC programs should attempt to do tasks that allocate long-term scratch areas in the main program, rather than in subprograms, where they will trap short-term subprogram-related scratch areas. Whenever possible, tasks requiring short-term scratch space should be isolated within a subprogram.

Logical ROMs

The HP 82412A ROM/EPROM card accommodates ROMs or EPROMs of different sizes: 32, 64, 96, or 128K. These different sizes are considered to be "logical ROMs" for two reasons:

- A logical ROM of size *N* does not have to contain *N* bytes of program and data files; it can contain less than *N* bytes. For example, a 64K logical ROM may only contain 44K of program and data files.
- A logical ROM of size *N* does not have to be placed in a ROM or EPROM integrated circuit (IC) of size *N*. For example, a 96K logical ROM can be contained in either three 32K ICs or two 64K ICs.

Logical Structure of the ROM/EPROM Card

Below is a memory map of the ROM/EPROM card.

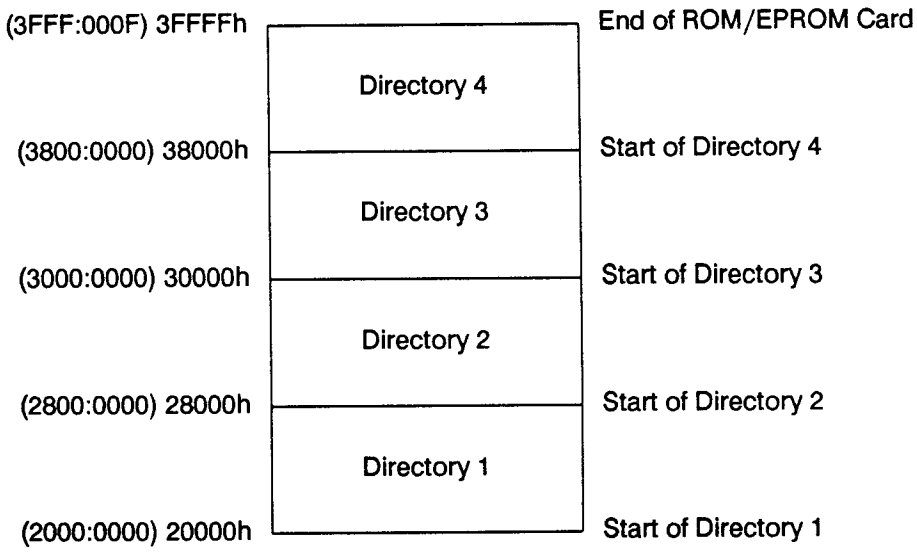


Figure 1-13. Memory Map of the HP 82412A ROM/EPROM Card

This memory map illustrates an important aspect of logical ROMs. Each directory begins on a 32K address boundary within the ROM/EPROM card address space (20000h to 3FFFFh). Each logical ROM is assigned a directory number corresponding to the 32K address boundary where the ROM will start. A logical ROM larger than 32K will span more than one 32K block of addresses. The pointers in

the directory table header created by HXC will reflect that the starting address is on a 32K boundary, and that the logical ROM space for large ROMs spans multiple 32K blocks. (For ROMs that span more than one directory, the directory number specified when the ROM is created is the starting directory number.)

For example, a 96K logical ROM starting at directory 1 will span directories 1, 2, and 3, leaving one 32K block of addresses, directory 4, available for a single 32K logical ROM. Similarly, a 64K logical ROM starting at directory 3 will span directories 3 and 4, leaving two 32K block of addresses, directories 1 and 2, available. These can be filled by either another 64K logical ROM starting at directory 1, or two 32K logical ROMs, one starting at directory 1, and the other starting at directory 2. A 96K logical ROM could not start at directory 3, nor could a 64K logical ROM start at directory 4, because they would have to span into a 32K block of addresses not available to the ROM/EPROM card.

Combining Logical ROMs of Different Sizes

Logical ROMs of different sizes can be combined in many different ways, subject to the following restrictions:

- The total number of logical ROMs cannot exceed four.
- The total number of directories spanned by all the logical ROMs cannot exceed four.
- The total space required by all logical ROMs, regardless of the amount of code they contain, cannot exceed 128K.

This is illustrated by the following diagram, which shows the possible logical ROM combinations for filling 128K of ROM space. Of course, a ROM/EPROM card does not have to be full — that is, it can contain fewer than four logical ROMs, span fewer than four directories, and contain less than 128K total ROM.

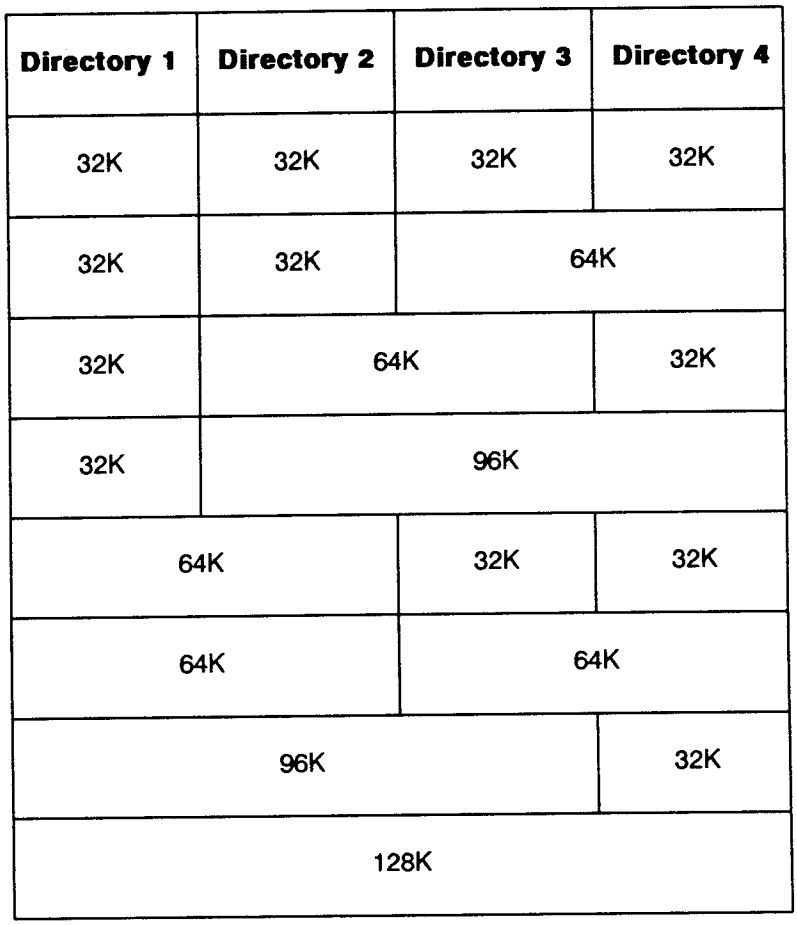


Figure 1-14. Possible Logical ROM Configurations

The memory map of an individual ROM within the ROM/EPROM card is essentially the same as for the 40K RAM card. The major difference is the values of the pointers — these can vary depending on the starting directory number, the directory table size, and the logical ROM size. Below is a memory map of a 32K logical ROM starting at directory 2.

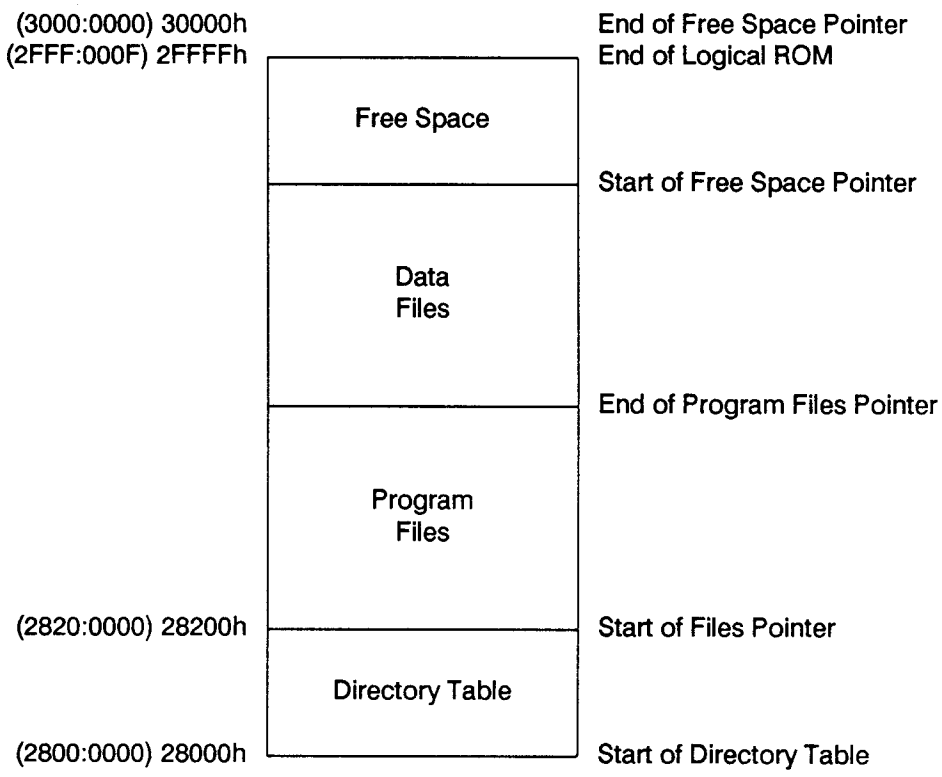


Figure 1-15. Memory Map of a 32K Logical ROM in Directory 2

Rather than provide memory maps for all the possible logical ROMs in directories 1-4, the addresses of the start and end of the logical ROM and for the start of program files (end of directory table) are shown in the following table.

Table 1-4. Addresses for All Logical ROM Sizes in Directories 1-4

Logical ROM Size	Directory Number	Start of Logical ROM	Start of Program Files Pointer	End of Free Space Pointer
32K	1	2000:0000	2020:0000	2800:0000
	2	2800:0000	2820:0000	3000:0000
	3	3000:0000	3020:0000	3800:0000
	4	3800:0000	3820:0000	4000:0000
64K	1	2000:0000	2020:0000	3000:0000
	2	2800:0000	2820:0000	3800:0000
	3	3000:0000	3020:0000	4000:0000
96K	1	2000:0000	2040:0000	3800:0000
	2	2800:0000	2840:0000	4000:0000
128K	1	2000:0000	2040:0000	4000:0000

Selecting a Logical ROM Size

From the different possible logical ROM sizes, select those best for a specific application based on its particular needs. Some of the items to consider are the total number of program and data files needed, maximum file size, total ROM space required for directory tables (which decreases available ROM for the application), and segmentation of code into blocks of different sizes. Below is a comparison of the differences in organizing a 96K application in three different ways: three 32K ROMs, one 64K ROM and one 32K ROM, or one 96K ROM.

Table 1-5. Different Organizations of a 96K Application

Logical ROM Sizes	Total Number of Files	Maximum File Size	Directory Table Overhead	Segmentation Required
3 32K ROMs	3 * 31 = 93	31.5K	3 * .5 = 1.5K	three separate groups of files that each fit in 32K
1 64K ROM + 1 32K ROM	31 + 31 = 62	63.5K, 31.5K	.5 + .5 = 1K	one group of files that fits in 64K and one group of files that fits in 32K
1 96K ROM	63	95K	1K	none

The same reasoning can be applied to other size applications and other logical ROM choices. The results of this analysis should be matched up against the requirements of the application to select the best way to organize it.

ROM and EPROM IC selection is another factor to consider, and will be discussed later.

Physical Layout of the ROM/EPROM Card

The ROM/EPROM card contains a circuit board with three sockets on it for ROM or EPROM ICs. The sockets can accommodate either 32K ICs or 64K ICs (a jumper on the board selects which IC size is being used). Different IC sizes cannot be mixed and matched — the board can hold either up to three 32K ICs or up to two 64K ICs. A diagram of the card is shown below.

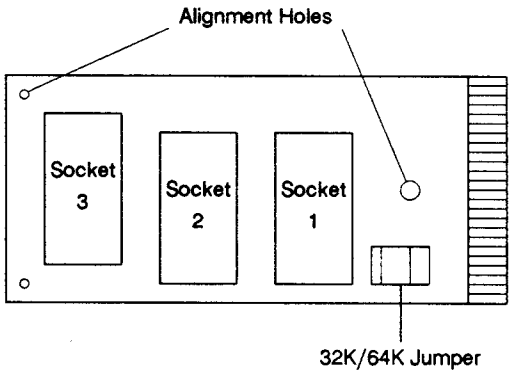


Figure 1-16. HP 82412A ROM/EPROM Card Circuit Board

The socketed jumper on the board selects between 32K ICs and 64K ICs. Underneath the jumper are the legends [256] and [512], meaning 256 Kbits (32 Kbytes) or 512 Kbits (64 Kbytes). To select the 32K ICs, insert the jumper so its solid metal strips connect jumper pins whose mating holes on the board are marked with the [256] symbol. (This is the configuration shown in the diagram.) To select 64K ICs, insert the jumper to use the holes marked with the [512] symbol.

Each socket on the board begins on a 32K address boundary within the ROM/EPROM card address space corresponding to the 32K blocks of address space in which logical ROMs reside. Socket 1 corresponds to directory 1, 2 to 2, and 3 to 3. A 32K IC can therefore be placed in any socket on the board (1, 2, or 3). A 64K IC will span more than one 32K block of addresses. Consequently, 64K ICs can be placed only in sockets 1 and 3. Placing a 64K IC in socket 3 gives access to the fourth 32K block of addresses — this is the "fourth" socket on the board for directory 4.

This means that using 32K ICs, 96K of physical ROM space is the maximum available, and using 64K ICs, the full 128K is available.

Selecting an IC Size

The directory numbers selected for the different logical ROMs will depend on where the logical ROMs will be placed on the board in the ROM/EPROM card. Some of that will depend on which IC size is chosen. The following items should be considered when making an IC size selection:

- Application size
- Price
- Availability
- Correct electrical specifications
- Supported by EPROM programmer (EPROMs only)

Refer to the "Hardware Specifications" for information about electrical and environmental specifications and manufacturers for the different IC sizes.

Placing Logical ROMs Into Physical ICs

In addition to the previous restrictions on combining logical ROMs, and the fact that IC sizes cannot be mixed, there is one more restriction that applies when placing logical ROMs into physical ICs: the physical IC must be placed in the socket on the board which corresponds to the directory number for the logical ROM contained in that IC.

Logical ROMs and physical ICs can both span 32K address boundaries, but this spanning is independent of each other (with the above restriction). This fact yields two important results. First, a logical ROM can cross physical IC boundaries; if it could not, logical ROMs larger than 32K would not be possible. Second, it does not matter what part of a logical ROM occupies a given physical IC as long as the logical ROM's starting directory number corresponds with the socket it occupies on the board, and the different pieces of the logical ROM are kept in the proper order.

Continuing the previous example of a 96K application, below are the ways that the logical ROMs could be placed in physical ICs. Each row of the tables represents a different way to place the particular logical ROM in the ICs.

Table 1-6. Placing a 96K Application Into Three 32K ICs

Logical ROM Sizes	Which Part of Logical ROM Put In 32K IC In Socket 1	Which Part of Logical ROM Put In 32K IC In Socket 2	Which Part of Logical ROM Put In 32K IC In Socket 3
3 32K ROMs	one entire 32K ROM	one entire 32K ROM	one entire 32K ROM
1 64K ROM + 1 32K ROM	first half of 64K ROM	last half of 64K ROM	entire 32K ROM
	entire 32K ROM	first half of 64K ROM	last half of 64K ROM
1 96K ROM	first third of 96K ROM	middle third of 96K ROM	last third of 96K ROM

Table 1-7. Placing a 96K Application Into Two 64K ICs

Logical ROM Sizes	Which Part of Logical ROM Put In 64K IC In Socket 1		Which Part of Logical ROM Put In 64K IC In Socket 3	
	First Half of IC	Last Half of IC	First Half of IC	Last Half of IC
3 32K ROMs	one entire 32K ROM	one entire 32K ROM	one entire 32K ROM	
		one entire 32K ROM	one entire 32K ROM	one entire 32K ROM
	one entire 32K ROM		one entire 32K ROM	one entire 32K ROM
	one entire 32K ROM	one entire 32K ROM		one entire 32K ROM
1 64K ROM + 1 32K ROM	first half of 64K ROM	last half of 64K ROM	entire 32K ROM	
	first half of 64K ROM	last half of 64K ROM		entire 32K ROM
	entire 32K ROM	first half of 64K ROM	last half of 64K ROM	
		first half of 64K ROM	last half of 64K ROM	entire 32K ROM
	entire 32K ROM		first half of 64K ROM	last half of 64K ROM
		entire 32K ROM	first half of 64K ROM	last half of 64K ROM
1 96K ROM	first third of 96K ROM	middle third of 96K ROM	last third of 96K ROM	
		first third of 96K ROM	middle third of 96K ROM	last third of 96K ROM

As the tables indicate, the segmentation of an application across logical ROM boundaries has no bearing on the way the ROMs are segmented to fit into physical ICs, as long as the starting directory number corresponds with the socket number, and the different pieces of the logical ROM are kept in the proper order.

The same reasoning can be applied to other size applications and other logical ROM choices. The results of this analysis should be matched up against the requirements of the application to select the best way to organize it.

System ROM

The system ROM is 32K of EPROM located in directory 5 in the upper 32K of the CPU address space. While this directory can be examined in command mode, it cannot be referenced by number or by any of its files during a running program. During a running program, the OPEN, FIND_FILE, and FIND_NEXT functions (0Fh, 16h, and 17h) will only find files in directories 0-4. The system ROM

contains four major blocks, shown in the memory map below.

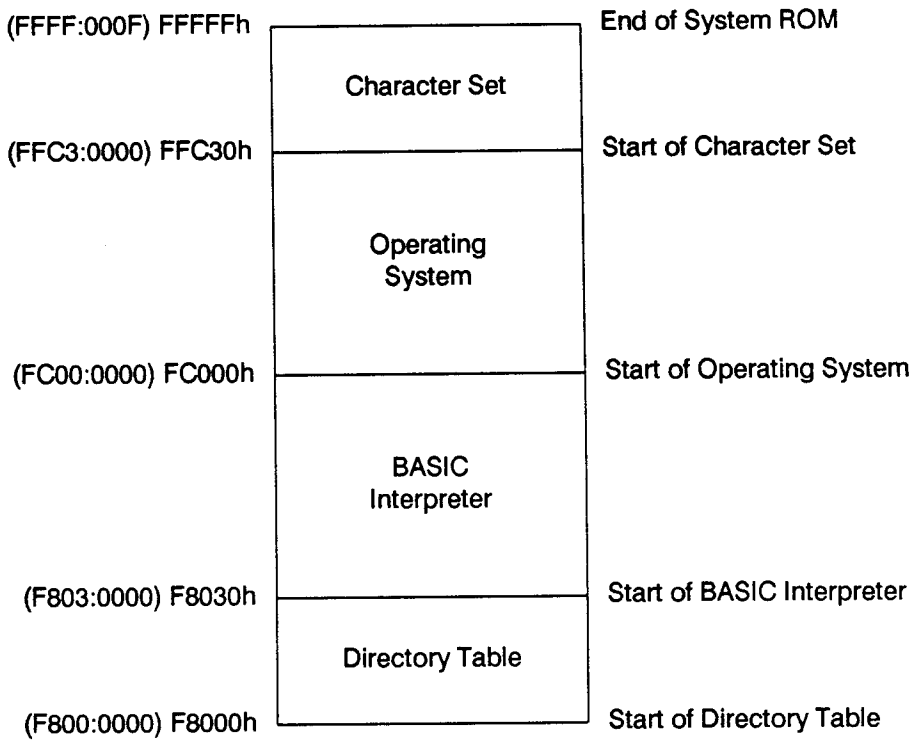


Figure 1-17. Memory Map of the System ROM

- Directory Table
  This contains only three entries: directory table header, BASIC interpreter file entry (SYBI), and operating system file entry (SYOS).
- BASIC Interpreter
  This is file SYBI.
- Operating System
  This is file SYOS.
- Character Set
  This is the dot pattern for the Roman-8 character set.

Memory Integrity Verification

The operating system computes and saves checksums of various areas of memory when the 94 is turned off. When the 94 is turned back on, the checksums are recomputed and compared with the saved values. Any changes indicate that memory integrity has not been preserved, and an error message is issued. Checksums are computed such that the sum of all words in the block being verified, plus the

checksum, will equal zero.

The major blocks of memory for which checksum errors are reported are directory tables, files, reserved scratch space, and free space. In addition, a checksum is made of the system ROM, and the reserved scratch space is tested extensively. These operations are discussed below.

Checksums Computed at Power Off

At power off, checksums for all RAM areas (main memory and 40K RAM card) are computed and saved. Checksums for ROM/EPROM card are not computed, since they are fixed in ROM, but they are saved in the reserved scratch space for comparison at power on. The system ROM checksum is also not computed.

Memory Integrity Tests at Power On

At power on, the operating system checks the main NiCd battery voltage. If it is below the low battery interrupt level, the machine is immediately turned off. If the voltage is OK, integrity tests are performed in the order shown in the following table. If any of the first three tests fail, the machine will not enter command mode. If any of the other tests fail, the machine will enter command mode and issue an error message. Any program run at that time will cold start.

Table 1-8. Memory Integrity Errors

Integrity Test Performed	Main Memory Error	40K RAM Card Error	ROM/EPROM Card Error
System ROM Checksum	low beep	—	—
Reserved Scratch Space Read/Write	high beep	—	—
Valid RAM Configuration	high beep and memory map	—	—
Directory Table Header Consistency	212 and require `I 0`	213 and require `I 1`	—
Reserved Scratch Space Checksum *	214	—	—
Free Space Checksum *	215	—	—
Directory Table Checksums *	212 and make type O	213 and make type O	213
File Checksums *	216 and set MSB of name	217 and set MSB of name	217

* Not computed at power off or power on if power turned off by pressing the reset switch or by automatic turn-off 2-5 minutes after the low battery interrupt.

These tests and their results are described below.

- System ROM Checksum
  If the stored checksum in the system ROM does not match the computed checksum, the operating system will issue a continuous low tone beep, and will not enter command mode.

- Reserved Scratch Space Read/Write
  If every byte in the reserved scratch space cannot be read and written, the operating system will issue a continuous high tone beep, and will not enter command mode.

- Valid RAM Configuration
  The RAM configuration is checked by reading and writing the first word of every RAM IC. If there is any other configuration of built-in RAM than 64K, 128K, 256K, or the RAM card has other than 40K, the operating system will issue a continuous high tone beep, and will not enter command mode.

  In addition, a memory configuration map will be displayed indicating the incorrect RAM ICs. The map is in the form "`Error `" followed by eight hex characters. The bits in each character represent individual RAM ICs. Reading from right to left, each bit will be a 1 if the IC was present, and a 0 if the IC was not present. For example,

  ```
  Error FFFFFFDF
  ```

  indicates that the sixth RAM IC was not present (the last 8 bits of the map are 11011111). Shown below is what the memory configuration map would be if the different configurations were correct. (These patterns will never appear, because only an incorrect pattern will be displayed.)

Table 1-9. Configuration Map for Valid Memory Configurations

Memory Configuration	Configuration Map if Configuration Correct
64K	`Error 000000FF`
128K	`Error 0000FFFF`
256K	`Error FFFFFFFF`
64K+40K RAM Card	`Error 001F00FF`
128K+40K RAM Card	`Error 001FFFFF`

After this test, the operating system will check the keyboard. If any keys are down other than CLEAR and ENTER, the machine will turn back off immediately. This is to prevent accidental turn on (while in a full briefcase, for example).

- Directory Table Header Consistency
  This verifies the consistency of the directory table headers for main memory and the 40K RAM card. The `*DIR*` directory identifier must be intact and the different pointers must point to successively higher addresses. If not, error 212 or 213 is issued, and the directory table is flagged such that the user must initialize the directory with the `I` (*initialize*) command (`I0` or `I1`). This also occurs if the size of main memory has changed (by adding or removing the 128K memory board).

- Reserved Scratch Space Checksum
  This is the checksum of the interrupt vector area and the operating system scratch space. If this checksum error occurred, error 214 will be issued.

- Free Space Checksum
  This is the checksum of the free space (and scratch areas, if any) — everything higher in main memory than the end of free space pointer. If this checksum error occurred, error 215 will be issued.

- Directory Table Checksums
  These are the checksums of the directory tables in any directory. If a directory table checksum error occurred for main memory or the 40K RAM card, error 212 or 213 will be issued, and the directory type in the directory table header will be changed to O (ROM directory). This makes the directory read-only, allowing the data to be retrieved, but not changed. To make the directory table type **M** or **A** again, the user must initialize the directory with the `I` command (`I0` or `I1`) after retrieving any desired data.

  For the ROM/EPROM card, only the error (213) will be issued — the directory type is already type O. Any checksum error in a ROM or EPROM (especially an EPROM) implies that the IC had one or more bits change state, and the IC should be replaced.

  The operating system recognizes that a card has been plugged in or removed, or that ROMs were changed on the ROM/EPROM card, because the number and contents of the directory tables has changed. When these conditions occur, they will not cause a checksum error, but will cause the machine to cold start.

- File Checksums
  These are the individual checksums for each file in any directory. If a file checksum error occurred for main memory or a RAM card, the MSB of the first character of the file name will be set. This will cause the file names to be displayed with a leading asterisk (*) when the D (*directory*) or M (*memory*) operating system commands are executed. If a file name has already been flagged as being corrupted, its checksum will not be computed at power on.

  If a file checksum error occurred in a ROM/EPROM card, the file name will not be altered, so no asterisk will appear when using the D or M commands. Any checksum error in a ROM or EPROM (especially an EPROM) implies that the IC has had one or more bits change state, and the IC should be replaced.

  Even with the MSB set in the file name, all normal file operations can still be performed: open, close, read, write, delete, find, execute, etc. All these operations are risky (especially running corrupted programs) because the state of the file is unknown. Unless the program or the user has the ability to reconstruct corrupted data, the safest action would be to erase the corrupted files and either replace them (program files) or recreate them (data files).

After all memory integrity tests have been performed, the operating system checks the lithium backup battery voltages. If the voltages are too low, the machine will enter command mode, and issue error 210 (main memory) and/or 211 (128K memory board or 40K RAM card).

2

Program Execution

Contents

Chapter 2 Program Execution

2

Program Execution

This chapter describes program execution in the HP-94: behavior at run time, cold start and warm start, program structure, and restrictions.

Running Programs

Program files are any of the non-data files — file types A, B, or H. They can reside in RAM or ROM/EPROM, and have some characteristics that are described here. Details on new BASIC keywords (type A) and user-defined handlers (type H) are in the BASIC interpreter and handler sections of this manual. BASIC programs are discussed in the BASIC interpreter section of this manual, as well as in the *BASIC Language Reference Manual.*

Autostart

When the HP-94 cold starts (discussed later), the operating system will automatically run the first file called `MAIN` that it finds. It searches directories 0-4 in ascending order, and if the first `MAIN` file encountered is type A or type B, it will be run; if not, an error will be issued. This search order allows a `MAIN` program in directory 0 (main memory) to override a `MAIN` file in directories 1-4 (40K RAM card or ROM/EPROM card).

Programs can also be run using the S (*start*) operating system command. Programs run with S will always cold start.

In-Place Execution of Programs

Program files are executed in place, regardless of where they are located in memory. Programs in ROM do not have to be copied into RAM before being executed. Space for BASIC program variables and scratch areas for assembly language programs and handlers are allocated from main memory, regardless of which directory the program resides in.

Behavior at Run Time

Program files always appear first in the file system for each directory, as illustrated in the memory maps. This placement occurs regardless of the order in which files are loaded. The C (*copy*) command ensures that all RAM-based program files are located before any data files. HXC ensures the same condition for ROM-based programs.

This is important because program files do not move at run time. All files lower in memory than the

end of program files pointer will not move at run time. However, because the order programs are loaded may vary, it is not known until run time exactly where each file may be located (and therefore what the initial CS will be). There is no segment fixup performed as is true for MS-DOS programs. Consequently, all references to addresses within program files must be relative to the start of the file — there can be no far calls or far jumps. This is particularly important for assembly language programs; HXBASIC and HXC handle this for BASIC programs.

Data files, however, can move at run time, since they can expand and be deleted. Since the operating system assumes that programs do not move at run time, data files must appear after all program files so that data file expansion and deletion will not change the location of programs.

Behavior of Reserved Files

There are four files with reserved names that must not be used for anything except their current use:

- SYBI — built-in BASIC interpreter
  If this file is run with the S (*start*) command, the operating system will immediately return to command mode.
- SYBD — BASIC debugger
  If this file is run with the S command, the operating system will immediately return to command mode (with the side effects shown in Table 2-3 for a FAR RET).
- SYFT — user-defined font
  If this file is run with the S command, the data in the file will be treated as code, which will have unpredictable (and possibly harmful) side effects.
- SYOS — built-in operating system
  If this file is run with the S command, the operating system will immediately turn the machine off.

When the BASIC interpreter searches for user-defined keywords with %CALL, the 12 built-in keywords starting with new keyword files of the same name SY will be *not* be overridden by new keyword files of the same name (SYAL, SYBP, SYEL, SYER, SYIN, SYLB, SYPO, SYPT, SYRS, SYRT, SYSW, and SYTO).

Cold Start and Warm Start

The HP-94 supports two methods of running programs when the machine is turned on: *cold start* and *warm start*. The fundamental difference is where the program starts running.

At cold start, the program starts running at the beginning. All conditions are reset to their default state. At warm start, the program continues running from the point at which it turned the power off. Most conditions are preserved in the state they were in while the program was previously running, although a few are reset to their default state. The warm start state is seen by user-defined handlers when their WARM routines are called.

The details of what state the machine is in at cold and warm start are described below. Notice that there are several items at the beginning of the table that behave identically, regardless of cold or warm start. This is particularly important for handlers. In the WARM routine of a handler, the handler must restore I/O devices to their required state (power, interrupt vector addresses, and interrupt enable/disable status) since they are always set to their default state, even at cold start.

Table 2-1. HP-94 Status at Cold and Warm Start

Item	Status at Cold Start	Status at Warm Start
Display	Cleared	Cleared
Input/Output	Halted	Halted
Interrupt Vector Addresses	Set to Default *	Set to Default *
Interrupt Enable/Disable Status	Set to Default †	Set to Default †
Copy of Main Control Register	00h	00h
Copy of Interrupt Control Register	31h †	31h †
Serial Port Power	Off	Off
Built-In Serial Port Buffer	Cleared	Cleared
Bar Code Port Power	Off	Off
Bar Code Port Transitions	Disabled	Disabled
Key Buffer	Cleared	Cleared
Beeper	Turned Off	Turned Off
User-Defined Characters	Available	Available
Access to Directory 5	Disabled	Disabled
MAIN Program	Starts at Beginning	—
Current Program	—	Restarts at Power Off Point
System Timeout Value	120 s	Unchanged
Display Backlight Timeout Value	120 s	Unchanged
Display Backlight	Turned Off	Unchanged
Cursor Status	On	Unchanged
Cursor Type	Underline	Unchanged
Keyboard Status	Unshifted	Unchanged
Low Battery Behavior	Halt Program With Error 200	Unchanged
Power Switch Behavior	Turn Off Machine	Unchanged
Timeout Behavior	Turn Off Machine	Unchanged
Allocated Scratch Areas	Returned to Free Space	Preserved
Available Free Blocks	Returned to Free Space	Preserved
BASIC Variable Contents	Lost	Preserved
Open Data Files	Closed	Left Open
File Access Pointers	Reset to Zero	Unchanged
Handler Information Table	Cleared	Unchanged
Open Channel 1-4 Handlers	Closed	Left Open ‡
Channel 1-4 Handler Configurations	Lost	Preserved ‡
Channel 1-4 Buffers	Lost	Preserved ‡
Open Built-In Serial Port Handler	Closed	Left Open, Serial Port On
Built-In Serial Port Configuration	Set to Default §	Unchanged
Stack Pointer	Points to OS Stack	Unchanged

* System timer (50h), serial port data (53h), low main battery voltage (54h), power switch (55h), operating system function (1Ah), user timer (1Ch), and dedicated (00h-03h) interrupt vectors all point to their operating system interrupt service routines. All others point to a dummy FAR RET.

† System timer, low main battery voltage, and power switch interrupts are enabled. All others are disabled.

‡ Exact warm start behavior depends on user-defined handler. The handler must restore the I/O device to its proper state (power, interrupt vector addresses, and interrupt enable/disable status).

§ 9600 baud, 7ES, XON/XOFF enabled, no terminate character, null strip disabled.

When Cold Start Occurs

The 94 will cold start a program under the following conditions:

- After default power off, either because the machine timed out or because the program turned it off with the END_PROGRAM function (00h) and specified cold start.
- After pressing the reset switch.
- After the automatic power off occurs 2-5 minutes after low battery interrupt.
- If any memory integrity error occurred at power on.
- After entering command mode, either when a program ends or by pressing [CLEAR] and [ENTER] at power on.
- If the program is run using the S (*start*) operating system command.
- If main memory size changes (128K memory board added or removed).
- If 40K RAM card changed to ROM/EPROM card, or vice-versa.
- If number or size of directories in ROM/EPROM card changed.

When Warm Start Occurs

The 94 will warm start the program if the program turned the machine off with the END_PROGRAM function and specified warm start, and none of the cold start conditions occurred.

Operating System Activities During Cold Start

When the 94 cold starts, it begins by performing the normal power-on initialization (check memory integrity, determine memory configuration, etc.). The operating system looks for a file called MAIN by searching directories 0-4 in ascending order. If MAIN exists, the status defined in the previous table is set. If no MAIN file is found, or if MAIN is not type A or B, the machine cannot autostart, so it enters command mode.

If MAIN is type A, the operating system does a FAR CALL to the main entry point of the program — the segment address of the start of the program and an offset of 6 (past the end of the program header). This implies that an assembly language program can end with a FAR RET — see the section on "Ending Programs" for further information.

If MAIN is type B, it will be executed by the BASIC interpreter. The operating system searches for a BASIC interpreter (SYBI) in directories 0-5 in ascending order. Error 100 is issued if none is found, or if the one found is not type A. Once the interpreter is found, control is transferred to it. It allocates and initializes its scratch area and the variable space required by the program, sets default values for various BASIC program conditions (shown below), and begins interpreting the program.

Table 2-2. Cold Start Status of BASIC Programs

Item	Initial Status
BASIC Numeric Variables and Arrays	Set to zero
BASIC String Variables and Arrays	Set to null string
SYEL Value	120 seconds
SYER Value	Error trapping disabled
SYLB Value	Default low battery behavior
SYRS Value *	9600 baud, 7ES, XON/XOFF enabled, no terminate character, null strip disabled
SYSW Value	Default power switch/timeout behavior
SYTO Value	120 seconds
* These values override any values specified by the B (*baud*) operating system command.	

Operating System Activities During Warm Start

When the 94 warm starts, it begins by performing the normal power-on initialization (check memory integrity, determine memory configuration, etc.) and executes the WARM routines of any open handlers. Then the operating system transfers control to where the program was running when the power was turned off, and the program continues running.

Ending Programs

Assembly language programs can end in one of two ways. They can either turn the power off, or they can leave the power on and enter command mode. Command mode is where the user can type operating system commands such as C (*copy*) or D (*directory*), and is usually reached by turning on the machine on while holding down the CLEAR and ENTER keys.

The END_PROGRAM function (00h) is used to end a program and turn the power off, specifying that the next power on be cold or warm start. For warm start, the CPU registers are saved on the operating system stack for use when the machine next turns on. If the program has used the operating system stack for its own data, the data will be destroyed when the CPU registers are saved. Therefore, a program cannot specify warm start unless it uses its own stack. If it specifies warm start while using the operating system stack, END_PROGRAM will issue error 219 and enter command mode.

There are two ways to enter command mode from a program. The first way is with a FAR RET, since the program was executed with a FAR CALL. The second way is to use the END_PROGRAM function, specifying to enter command mode. There are subtle differences in the operating system behavior with these two approaches, summarized below.

Table 2-3. Ending a Program With END_PROGRAM or FAR RET

Item	Behavior Using END_PROGRAM	Behavior Using FAR RET
CPU Interrupt Flag	Set (STI)	Unchanged
Access to Directory 5	Enabled	Disabled
Open Files	Closed	Not Closed
Handler CLOSE Routines	Called	Not Called *

* The handler will have no opportunity to restore interrupt vectors or status. Power will be continue to be supplied to the serial port, level converter, and bar code port if they were enabled.

Because of these differences, the END_PROGRAM function is the preferred method of ending a program and entering command mode.

Operating System Activities When Entering Command Mode

When the operating system enters command mode, it initializes certain things to their default values, as shown below.

Table 2-4. HP-94 Status in Command Mode

Item	Status
Input/Output	Halted *
Interrupt Vector Addresses	Unchanged *
Interrupt Enable/Disable Status	Unchanged *
Copy of Main Control Register	Unchanged *
Copy of Interrupt Control Register	Unchanged *
Serial Port Power	Off *
Built-In Serial Port Buffer	Cleared
Bar Code Port Power	Off *
Bar Code Port Transitions	Disabled *
Key Buffer	Unchanged
Beeper	Unchanged
User-Defined Characters	Not Available
Access to Directory 5	Enabled †
System Timeout Value	120 s
Display Backlight Timeout Value	120 s
Display Backlight	Turned Off
Cursor Status	On
Cursor Type	Block
Keyboard Status	Shifted
Low Battery Behavior	Halt Program With Error 200
Power Switch Behavior	Turn Off Machine
Timeout Behavior	Turn Off Machine
Allocated Scratch Areas	Returned to Free Space
Available Free Blocks	Returned to Free Space
BASIC Variable Contents	Lost
Open Data Files	Closed
File Access Pointers	Reset to Zero
Handler Information Table	Cleared
Open Channel 1-4 Handlers	Closed
Channel 1-4 Handler Configurations	Lost
Channel 1-4 Buffers	Lost
Open Built-In Serial Port Handler	Closed †
Built-In Serial Port Configuration	Set to Default ‡
Stack Pointer	Points to OS Stack

* Whether or not these conditions are true depends on the what the program does before it ends and the behavior of the CLOSE routines in any user-defined handlers in use (assuming the routines are called before the program ends). The CLOSE routines will be executed automatically when entering command mode with the END_PROGRAM function (rather than a FAR RET).

† Only if the END_PROGRAM function was used to enter command mode (rather than a FAR RET).

‡ 9600 baud, 7ES, XON/XOFF enabled, no terminate character, null strip disabled.

Program Structure

The three different types of programs (types A, B, and H) have a simple structure consisting of a program header followed by the code. Assembly language programs (type A) have a six-byte header, then the executable code. Handlers (type H programs) have a six-byte header, a jump vector table, then the code pointed to by each of the jump vectors. BASIC programs (type B) have a 16-byte header, then the program tokens.

Program Headers

Assembly language programs start with a six-byte header, shown below with hex offsets on the left side. Note that the order of this illustration is with the lowest offset at the top, which is the order the entries would be placed in the source code for the handler.

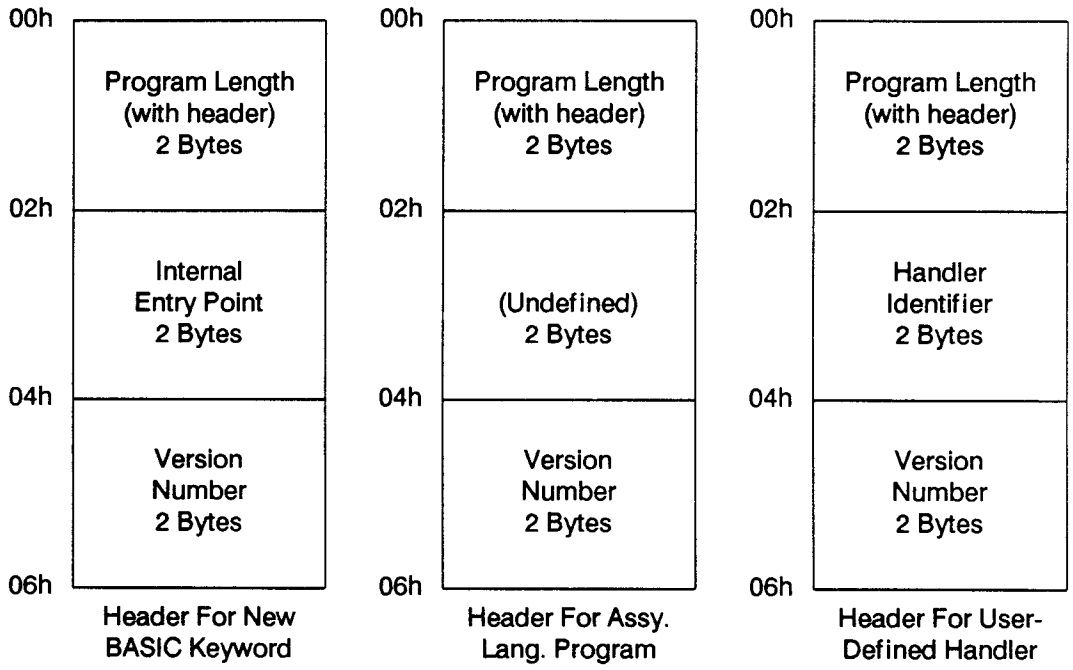


Figure 2-1. Program Headers

There are three fields in the header:

- Program Length
  This field is the length of the program, including the length of the header itself.
- Internal Entry Point
  For type A programs that are new BASIC keywords, this field is the offset of the processing block relative to the start of the program. This assumes a particular BASIC keyword structure which will be described shortly. If a BASIC keyword does not use this structure, this field can be set to point to the first byte after the header, to a dummy FAR RET instruction, or be used for other purposes.

- (Undefined)
  For type A programs that are not BASIC keywords, the place to start executing the program is immediately after the header, so the value of the internal entry point field does not matter — it will never be called by another program. It can therefore either be set to point to the first byte after the header, to a dummy FAR RET instruction, or be used for other purposes.

- Handler Identifier
  The second field in the header has a slightly different meaning for handlers. It contains a two-character identifier that is returned by the identify handler I/O control function (00h).

- Version Number
  This is used for revision control by the programmer. It is a two-byte binary number representing a decimal fraction of the form *II.FF*, where the *II* is the integer part of the version, and the *FF* is the fractional part of the version. The statement VERSION dw 0103h would designate a version number of 1.03, and the statement VERSION dw 0212h would define version 2.18 of the software. This can also be defined in decimal as db 18,2, where the fractional part precedes the integer part.

For type A programs, the program code starts after the header. For type H programs, the jump vector table that follows the header defines the locations of the executable code.

BASIC Keyword Structure

BASIC keywords can be written so that they are accessible from both BASIC and assembly language programs. This requires a keyword structure in which there are two distinct blocks: an I/O block in which all interaction with BASIC variables occurs, and a processing block in which the function of the keyword is implemented. Once the I/O block has read and validated the supplied variables, it calls the processing block. When the processing block is done, it returns its results to the I/O block, which then places them in BASIC variables as appropriate. This structure is shown below.

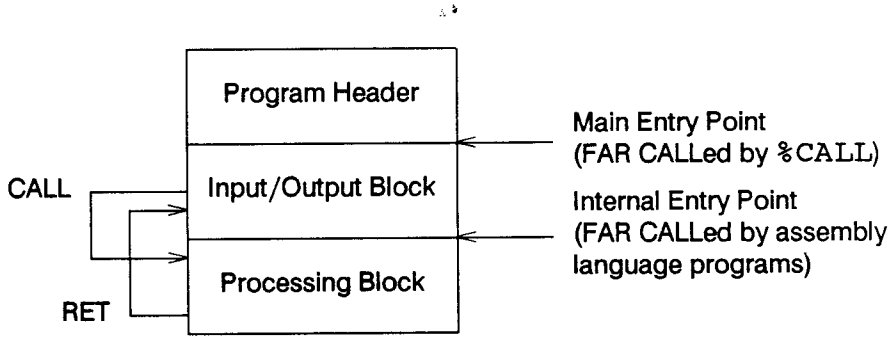


Figure 2-2. BASIC Keyword Structure

The internal entry point in the program header would point to the start of the processing block. This allows both BASIC and assembly language programs access to the functionality implemented by the keyword. BASIC programs execute new keywords with %CALL, which FAR CALLs the main entry point at the end of the header. Assembly language programs execute the processing block only via the internal entry point. They find the program, read the internal entry point from the header, set up appropriate parameters, and FAR CALL the processing block.

Errors should be reported differently depending on which entry point is called. If the main entry point is called (which implies the keyword was called by a BASIC program), non-numeric errors should be reported using the ERROR BASIC interpreter utility routine (offset 34h). This will cause a non-numeric error to be issued by the BASIC interpreter, and the BASIC program will halt. If the internal entry point is called (which implies the keyword was called by an assembly language program), numeric errors should be returned in the AL register (00h if no errors).

The main entry point of a BASIC keyword can also be called from command mode with the S command. This condition should be recognized by BASIC keywords. If the keyword was called from a BASIC program using %CALL, the CS register will be the same as the DS register. If the keyword was called from command mode with the S command, the CS register will be different than the DS register.

There are two possible ways to handle this condition. One approach is for the keyword to end immediately if the keyword is called from command mode. Another approach is to implement an input/output block for interacting with command mode, analogous to the input/output block for interacting with the BASIC interpreter.

Program Restrictions

Programs can start on any paragraph boundary, depending on where the program was loaded and what other files were loaded or deleted. Once they begin to run, they do not move — there is no run-time relocation. Consequently, there should be no far calls or jumps to absolute addresses in type A or H programs. (HXBASIC and HXC ensure this for type B programs.)

Valid EXE Format

When EXE files are created, they should not contain any MS-DOS-style relocation entries. HXC will reject any EXE file if it contains a relocation table. An EXE file, to be accepted by HXC, must have the following characteristics:

- EXE file size of 512 bytes or greater.
- Valid EXE identifier.
- 512-byte header.
- No relocation entries.
- Initial CS = 0000h.

It is recommended that source files use byte alignment by specifying SEGMENT BYTE at the beginning of each program segment. The assembler's default alignment is on paragraph boundaries, causing each object file to be padded with 1-15 bytes. Byte alignment eliminates this unused space. HXC will pad the entire EXE file only once, not once for each object file.

Use of Operating System Stack

A program can use the operating system stack for its own use. The stack varies in length, depending on how the program was called (from the operating system or from another program), up to a maximum of approximately 600 bytes. If a program turns off the machine and specifies a subsequent warm start (see "Cold Start and Warm Start"), it must not use the operating system stack. The END_PROGRAM function (00h) will issue error 219 if the program is using the operating system stack. Consequently, if a program wants to use the warm start option, it must put its stack in its own data space.

Programs in ROM or EPROM

Programs can be in RAM or ROM, and execute in place in either location. ROM programs have additional restrictions. There can be no data space in the code itself if the program is to have the option of running in ROM. The operating system provides scratch area allocation and release functions to allow ROM programs to get needed data space.

The assembler provides the ability to define the offsets within an external scratch area using the SEGMENT AT directive, as shown below.

```
SCR_AREA        segment at 0            ;Addresses start at 0
PARAM1          db      6 dup(?)        ;First parameter needs 6 bytes
PARAM2          db      00              ;Second parameter needs a byte
PARAM3          dw      0000            ;Third parameter needs a word
SCR_AREA        ends
```

Figure 2-3. Defining Scratch Area Data Structure

The SEGMENT AT directive provides an address template that can be imposed on the scratch area. SEGMENT AT causes no code to be generated for the uninitialized data defined within that program segment (in this case, the SCR_AREA segment).

3

User-Defined Handlers

Contents

Chapter 3 User-Defined Handlers

3

User-Defined Handlers

User-defined handlers, or handlers for short, allow BASIC or assembly language programs simple access to the HP-94 I/O ports — the devices associated with channels 1-4. In particular, user-defined handlers can be written for the serial port (channel 1) and bar code port (channel 2); channels 3 and 4 are reserved, and currently have no I/O port associated with them. Handlers are assembly language program files that are assembled and linked into EXE files on the development system. Then they are processed by HXC and given file type H before being copied into the HP-94.

Handlers are similar in concept to UNIX or MS-DOS device drivers. They are a collection of routines to handle various activities associated with I/O devices, such as initializing the port for use, reading and writing data to it, and releasing control of the port. Handlers have a special structure that allows the individual routines to be called, either from BASIC or assembly language, solely by supplying the name of the handler being used when the channel is opened.

This chapter will discuss handler organization in general, how handlers interact with the channel-oriented input and output of the HP-94, the different types of handlers, passing configuration parameters and registers to handler routines, and what tasks handler routines perform.

Handler Structure

Handlers contain three major components: the program header, the jump table, and the executable code for each of the handler routines.

Program Header

Handlers, like all assembly language programs, start with a six-byte header. The first two bytes are the length of the handler, including the header. The next two bytes are a two-character handler identifier that is returned by handlers that implement function 00h of the IOCTL routine (discussed later). The last two bytes of the header are the software version number. It is a two-byte binary number representing a decimal fraction of the form *II.FF*, where the *II* is the integer part of the version, and the *FF* is the fractional part of the version. The statement `VERSION dw 0103h` would designate a version number of 1.03, and the statement `VERSION dw 0212h` would define version 2.18 of the software. This can also be defined in decimal as `db 18,2`, where the fractional part precedes the integer part.

Jump Table

Immediately following the header is a jump table with 10 entries of three bytes each. Each entry contains a JMP instruction to one of the handler routines. Each routine must end with a FAR RET. The header and jump table, showing the order in which the jump table must appear in the program, is shown below. The hex offsets from the start of the program are along the left side. Note that the order

of this illustration is with the lowest offset at the top, which is the order the entries would be placed in the source code for the handler.

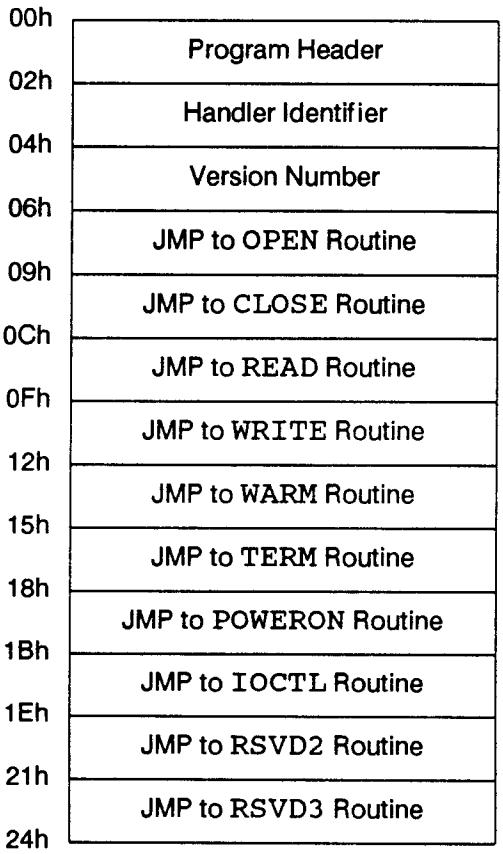


Figure 3-1. Handler Header and Jump Table

The purpose of the different handler routines are listed briefly below.

- OPEN Routine — initializes the port.
- CLOSE Routine — releases control of the port.
- READ Routine — reads data coming into the port.
- WRITE Routine — writes data to the port.
- WARM Routine — allows reinitialization of the port at warm start.
- TERM Routine — allows I/O to be terminated because of the power switch or low battery.
- POWERON Routine — allows initialization at machine power-on.
- IOCTL Routine — controls actions of handler.
- RSVD2 Routine — for future use.
- RSVD3 Routine — for future use.

Entries in the jump table are required for all handler routines. However, not all handlers will implement all routines. If a routine is not implemented, the jump table entry should just JMP to a dummy FAR RET.

There is no jump table entry for the handler's interrupt service routine. The address of that routine is placed in the appropriate interrupt vector in the reserved scratch space. For details on using interrupts, refer to the "Interrupt Controller" chapter.

The tasks performed by the different handler routines will be discussed later in this chapter. The next sections will describe general information relevant to all handlers and handler routines.

Channel Input and Output

The HP-94 operating system performs input and output through 16 different logical channels, each of which is associated with different physical devices. The channels being used for I/O are defined by opening them. From an assembly language program, this is done with the OPEN function (0Fh); from a BASIC program, this is done with the OPEN # statement (which calls the OPEN function). Both the OPEN function and the OPEN # statement take the channel number to open and a file name as their parameters. The table below summarizes the uses of the 16 logical channels, and the meaning of the file name for the different channels.

Table 3-1. Channel Number Assignments

Channel Number	Physical Device	File Name Meaning
0	Console *	Ignored
1	Serial Port	Name of User-Defined Handler (Type H)
2	Bar Code Port	Name of User-Defined Handler (Type H)
3-4	Reserved	Name of User-Defined Handler (Type H)
5-15	Data Files	Name of Data File (Type D)
* The console is the keyboard for input operations and the display for output operations.		

Below is more information about the different channels.

- Channel 0
  The console is always opened by the operating system. A program can specify a file name as a parameter when opening channel 0, but the name will be ignored — user-defined handlers for channel 0 are not allowed.
- Channel 1
  The built-in serial port handler is specified by supplying the null string ("") for the file name. If a user-defined device handler name is supplied and no such handler exists in memory, the default handler will be used.
- Channels 2-4
  There is no default handler for these channels. If the null string is used as the file name, or there is no handler in memory matching the file name supplied, an error will be reported.

- Channels 5-15
  When a data file is opened, the file access pointer is reset to the start of the file. Only one channel at a time can be assigned to a single file. Multiple channels cannot be open to the same file simultaneously.

Once a channel has been opened, an error will occur if it is reopened without first being closed.

File Search Order

The OPEN function will search for the specified file name in directories 0-4 in ascending order. If the file name includes a directory number (e.g., `"1:HNBC"`), only that directory will be searched. If the file name is found, but is an illegal type, (not type H for channels 1-4, or not type D for channels 5-15), an error will be issued. If it is a legal type, it will be opened.

Types of Handlers

There are two types of handlers: high-level and low-level. These support the concept of layered software, in which successively higher layers become more hardware-independent.

Low-Level Handlers

Low-level handlers interact only with the I/O port hardware. They take care of the characteristics of the I/O port on the HP-94 only. An example of this is HNBC, a low-level bar code port handler supplied with the *HP-94 Software Development System* that does low-level I/O with the bar code port. Low-level handlers usually include one or more interrupt service routines for the hardware interrupts associated with the I/O port.

High-Level Handlers

High-level handlers interact only with low-level handlers, not with the I/O port hardware. They take care of the characteristics of the external device connected to the port, but not of the port itself. An example of this is HNWN, a high-level handler that handles the device-specific features of Hewlett-Packard Smart Wands, but relies on the low-level handlers HNBC or HNSP to perform port-specific activities. High-level handlers do not have interrupt service routines because they do not interact directly with the hardware.

Who Calls Handler Routines

The routines in both types of handlers can be called by operating system functions, which in turn are called by BASIC I/O keywords, assembly language programs, or by the operating system itself. If a high- and low-level handler pair are being used, the operating system will think that only the high-level handler is open. All communication between the two handlers is performed by the high-level handler using *handler linkage routines*. These routines are described later in this chapter, and are available as an include file that can be included with the high-level handler source code (discussed in the appendixes).

The relationship between all the layers of software used for I/O is shown below.

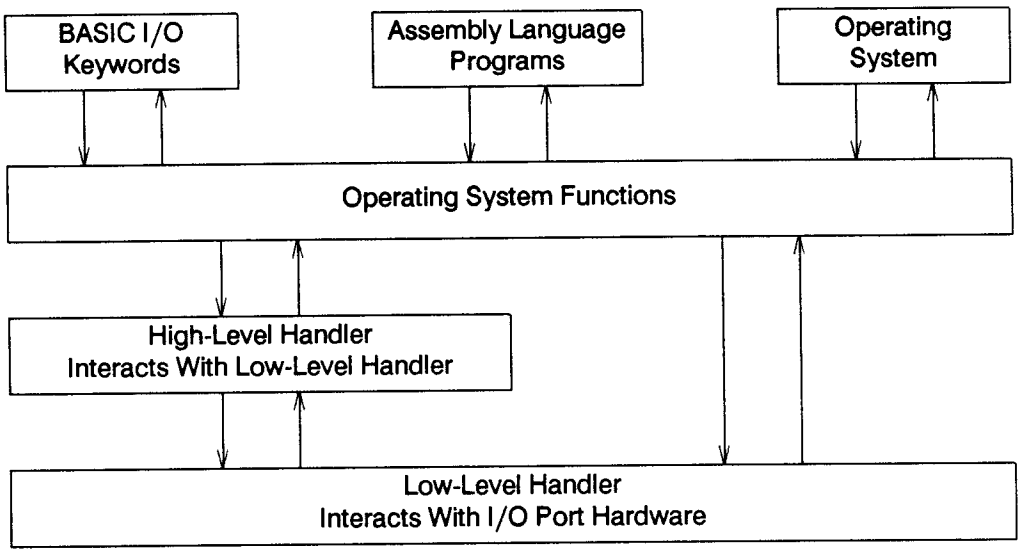


Figure 3-2. Relationship Between High- and Low-Level Handlers

As this diagram indicates, all that is required to perform I/O to a port is a low-level handler. It is not necessary to have or use a high-level handler. If external devices will be used with unique characteristics better accommodated on a driver level than an application level (so the application is more device-independent), then a high-level handler may also be necessary.

Because the high-level handler is totally dependent on the low-level handler to actually move data through the I/O port, high-level handlers cannot stand alone. A low-level handler can be used by itself, but a high-level handler must be used as part of a high- and low-level handler pair.

Handler Information Table

There is a table in the operating system scratch space where handlers keep information about scratch area locations. The table contains five two-byte entries, each of which is associated with a specific channel and has a different meaning depending on whether the handler is closed or open.

Table 3-2. Handler Information Table Entries

Entry Offset	Which Channel	Meaning While Handler Closed	Meaning While Handler Open	Used By Which Interrupt
00h	Bar Code Port	None	Low-Level Handler Scratch Area Address	Bar Code Timer (51h)
02h	Serial Port	Parameter Scratch Area Address	Low-Level Handler Scratch Area Address	Serial Port Data Received (53h)
04h	Bar Code Port	Parameter Scratch Area Address	Low-Level Handler Scratch Area Address	Bar Code Port Transition (52h)
06h	Channel 3	Parameter Scratch Area Address	Low-Level Handler Scratch Area Address	Reserved 1 (56h)
08h	Channel 4	Parameter Scratch Area Address	Low-Level Handler Scratch Area Address	Reserved 2 (57h)

Table Usage While Handlers Are Closed

When a handler is closed, the handler information table is used for the segment address of the parameter scratch area for that channel. When the OPEN routine in either a high- or low-level handler is called, it looks at the appropriate table entry to determine if the parameter scratch area exists and if the information it contains is valid. The procedure for doing this will be discussed later.

Table Usage While Handlers Are Open

Every time a routine in an open handler is called, the operating system automatically passes the segment address of the handler's scratch area to the routine in the DS register. However, the operating system cannot do this when an interrupt causes the handler's interrupt service routine to be executed. To allow the interrupt service routine to locate the scratch area, the handler information table is used for the address of the low-level handler's scratch area. This is done only when the handler is open, for this is the only time that interrupts will be enabled for the handler.

After verifying its parameters, the low-level handler's OPEN routine must save the parameter scratch area address in the handler's scratch area, and place the handler's scratch area address in that table entry. When the handler is closed, the low-level handler CLOSE routine must restore the original parameter scratch area address in that table entry.

Table Entry Offsets

The handler information table entry offsets for a particular handler are *2 * the handler channel number.* Once the handler is open, the entry is read during the handler interrupt service routine. This means that each handler can have one hardware interrupt associated with it. This is not true for the bar code port, since it has both a transition interrupt and a timer interrupt. The primary interrupt for the bar code port is the transition interrupt since it occurs on every transition, so it is associated with the entry for channel 2. The bar code port timer interrupt uses the first entry in the table at offset 0.

Reading and Setting the Handler Information Table

The handler information table is located in the first 10 bytes (5 words) of the operating system scratch space. Using the operating system pointer to locate the scratch space (described in the appendix), the following code will take the channel number in AL and load the table entry for that channel into ES:

```
mov     si,16h              ;get segment address of OS pointers
mov     ds,si               ;put in segment register
xor     ah,ah               ;clear ah
mov     si,ax               ;put channel number in si
shl     si,1                ;2 * channel number
mov     ds,ds:[0000h]       ;get the segment address of OS scratch space
mov     es,word ptr ds:[si] ;get this channel's table entry
```

Figure 3-3. Example of Reading Handler Information Table Entries

Passing Parameters to Handlers

Parameters are passed to a handler mainly to define its operating configuration (such as baud rate for the serial port). The handler uses them to set its configuration when its OPEN routine is called. Parameters can be passed in one of two ways when the handler is opened:

- The parameters can be placed in a *parameter scratch area*. This can be done from a BASIC program with a separate keyword (such as the SYBC keyword that defines parameters for HNBC), or from an assembly language program that allocates and initializes the parameter scratch area before opening the handler. This is the approach used for passing parameters to Hewlett-Packard handlers.
- The parameters can be placed after the handler name that is passed to the OPEN function or the OPEN # statement (e.g., "LLHN 9600,7ES"). The handler OPEN routine then parses the parameters from the name string.

Regardless of which approach is used to pass parameters, the low-level handler must save a copy of them in its scratch area. This is needed by the IOCTL routine of the handler.

Passing Parameters in a Parameter Scratch Area

A parameter scratch area is a one-paragraph scratch area. The upper 8 bytes (bytes 08h-0Fh) are reserved for high-level handler parameters, and the lower 8 bytes (bytes 00h-07h) are reserved for low-level handler parameters. The first byte of each half is used as a *valid data flag* (discussed shortly) to indicate the validity of the parameters. This leaves 7 bytes available for parameters for each high- and low-level handler.

Handlers verify two aspects of configuration parameters: first, that the parameter scratch area exists, and second, that it contains valid configuration information.

Verifying Parameter Area Existence

High- and low-level handlers determine if the parameter area exists by reading the handler information table entry for that channel. If the entry is zero, there is no parameter scratch area for the handler. The handler should then allocate a one-paragraph parameter scratch area and place its address in the table entry. If the entry is non-zero, the entry contains the segment address of a parameter scratch area that already exists.

It is important that the address of the parameter area put in the handler information table actually point to a scratch area. If an assembly language program opens a handler and passes it parameters, the address put in the table must not point to parameters on the program's stack, or to fixed parameters embedded in the program code. This is because if the stack vanishes or the program moves, the address in the handler information table will no longer point to valid parameters.

CAUTION When a handler is open, the entry in the handler information table will be the scratch area address of the handler, *not* of the parameter scratch area (see "Handler Information Table"). If a separate configuration program is run after the handler is open, it could misinterpret the handler information table entry, and modify the handler scratch area by mistake. Configuration programs should check if the handler is open before examining the handler information table. See the appendixes for a utility routine that determines if a channel is open or not.

Validating the Contents of the Parameter Scratch Area

High- and low-level handlers validate the contents of the parameter scratch area by looking at the first byte in their respective parts of the area (upper 8 bytes for high-level handlers, lower 8 bytes for low-level handlers). This first byte is a valid data flag that is unique for each handler associated with a particular channel. The valid data flag is set to zero when the scratch area is allocated because the operating system initializes all scratch areas to zero (00h). The flag is then set to a value either by a handler, by the program calling the handler, or by a configuration keyword. The action that a handler should take for different values of the valid data flag is shown below.

Table 3-3. Interpreting the Valid Data Flag

Value of Flag	High-Level Handler Action	Low-Level Handler Action
Zero	Put correct valid data flag and default high-level handler configuration in upper 8 bytes of parameter scratch area.	Put correct valid data flag and default low-level handler configuration in lower 8 bytes of parameter scratch area.
Correct for Handler	Use these parameters to define high-level handler configuration.	Use these parameters to define low-level handler configuration.
Any Other Value	Return an error, since the parameters are not valid for this handler.	Return an error, since the parameters are not valid for this handler.

Handlers should use values for the valid data flag in the range 01h-7Fh. Hewlett-Packard uses values in the range 80h-FFh for its handlers, and 00h is reserved because it indicates uninitialized parameters. Refer to the "Program Resource Allocation" appendix for information about reserving a valid data flag that will not conflict with any other flag in use.

Passing Parameters After the Handler Name

If parameters are passed in-line with the handler name, the handler's OPEN routine must parse and interpret the handler names and parameters. When the handler OPEN routine executes, `ES:BX` points to the start of the entire handler name string. The routine can skip past the handler name in the string to find the beginning of the parameters, and parse them into whatever internal form is required for the handler. The syntax of the name string is as follows:

- High-level handler name
- One or more spaces
- High-level handler parameters separated by commas
- Semicolon
- One or more spaces
- Low-level handler name
- One or more spaces
- Low-level handler parameters separated by commas
- Ending null (00h)

This results in handler and parameter strings that look like the following examples:

`"HNLL 7,2"`	Low-level handler with parameters
`"HNHL 1,3;HNLL 7,2"`	High- and low-level handlers with parameters
`"1:HNHL 1,3;1:HNLL 7,2"`	Same but with directory numbers
`"HNHL;HNLL"`	High- and low-level handlers with no parameters

Restrictions on In-Line Parameters

- If the OPEN # statement is used, the maximum length of the handler names and parameters is 255 characters.
- The OPEN # statement uppercases all characters in the name string, so the name string in OPEN `#1,"llhn 7es"` will be passed as `"LLHN 7ES"`. If a handler that accepts in-line parameters will be opened with the OPEN # statement, the parameters should not be case-sensitive.
- If a high-level handler that accepts in-line parameters calls a low-level handler that accepts parameters in a parameter scratch area (such as Hewlett-Packard handlers), the high-level handler must parse its in-line parameters and put them in the form expected by the low-level handler. Then it must create a parameter scratch area, place the parameters in it, and modify the handler information table before calling the low-level handler.

Handler Linkage Routines

If a high- and low-level handler pair are being used, the operating system will think that only the high-level handler is open. All communication between high- and low-level handlers is performed by the high-level handler using handler linkage routines. These routines are available as an include file that can be included with the high-level handler source code (discussed in the appendixes).

Each handler routine has a corresponding linkage routine that it uses to call the low-level handler. To use the linkage routines, load appropriate values into the registers, put the channel number in AL, and FAR CALL the routine by name. The activities of each high-level handler routine before and after calling the linkage routine will be discussed shortly.

The linkage routines are designed to mimic the way the operating system calls handler routines. A low-level handler will not be able to distinguish that it is being called by a high-level handler rather than by the operating system. Like the operating system, the caller's registers (in this case, the high-level handler's) are saved in a register save area on the stack when the low-level handler is called. Upon return, the registers are popped off in exactly the same manner. This means that low-level handlers must return the error code in AL (00h if no errors), and all other register values in the appropriate location in the register save area.

Below is a summary of the registers passed to and returned by the linkage routines.

Table 3-4. Register Usage By Handler Linkage Routines

Routine Name	Registers Passed		Registers Returned	
	Register	Contents	Register	Contents
LLH_CLOSE	AL	Channel number to close	AL	Error code
LLH_IOCTL	AL AH Others	Channel number IOCTL function code As defined by routine	AL Others	Error code As defined by routine
LLH_OPEN	AL ES BX	Channel number to open Segment address of low-level handler name to open Offset address of low-level handler name to open	AL	Error code
LLH_READ	AL CX ES BX	Channel number to read Number of bytes to read Segment address of read buffer Offset address of read buffer	AL CX	Error code Number of bytes actually read
LLH_RSVD2	AL Others	Channel number Not yet defined	AL Others	Error code Not yet defined
LLH_RSVD3	AL Others	Channel number Not yet defined	AL Others	Error code Not yet defined
LLH_TERM	AL AH *	Channel number Cause of termination 1 = power switch 0 = low battery	AL	Error code
LLH_WARM	AL	Channel number	AL	Error code
LLH_WRITE	AL CX ES BX	Channel number to write Number of bytes to write Segment address of write buffer Offset address of write buffer	AL CX	Error code Number of bytes actually written
All (supplied automatically)	DS † BP DI	Segment address of low-level handler scratch area Stack offset address of register save area Destroyed	BP	Unchanged from value passed to routine

* The TERM routine for high- and low-level handlers will receive the cause of the termination in AL. A high-level handler must move this value into AH and place the channel number in AL before calling LLH_TERM. LLH_TERM will swap them back, thereby passing the cause of the termination to the low-level handler in AL.

† Not passed to LLH_OPEN routine.

Handler Routine Descriptions

Handler routine descriptions consist of the following:

- A brief description of the routine.
- A summary of the parameters passed to the routine.
- A summary of the parameters that the routine must return.
- Details on when the routine is called.
- Supplementary notes and cautions on the use and behavior of the routine.

Registers Passed to Handler Routines

Handler routines are called by the analogous operating system functions. For example, the READ function will FAR CALL the READ routine in the handler that is open to the channel being read. When handler routines are called, either by the operating system or by handler linkage routines, all the registers values that were passed to the operating system function will be passed to the handler routine, with the following exceptions:

- The DS register contains the segment address of the handler scratch area (except for the OPEN routine).
- The BP register contains the offset on the stack where all the caller's registers were saved.
- The DI register is destroyed.

All the caller's original registers are saved in a register save area on the stack. When the handler routine ends (with a FAR RET), the caller (operating system function or handler linkage routine) will automatically pop all the saved registers off the stack *except* AL, which is used to return error codes, and BP, which must be unchanged from the value passed to the routine. Consequently, if a handler wants to return a value in a register other than AL or BP, it cannot just put the value in the register — the register will be lost when the saved register copies are popped off the stack. Instead, the handler routine must place values to be returned into the register save area on the stack.

The order that the registers are saved on the stack is shown below, with the hex offsets on the left.

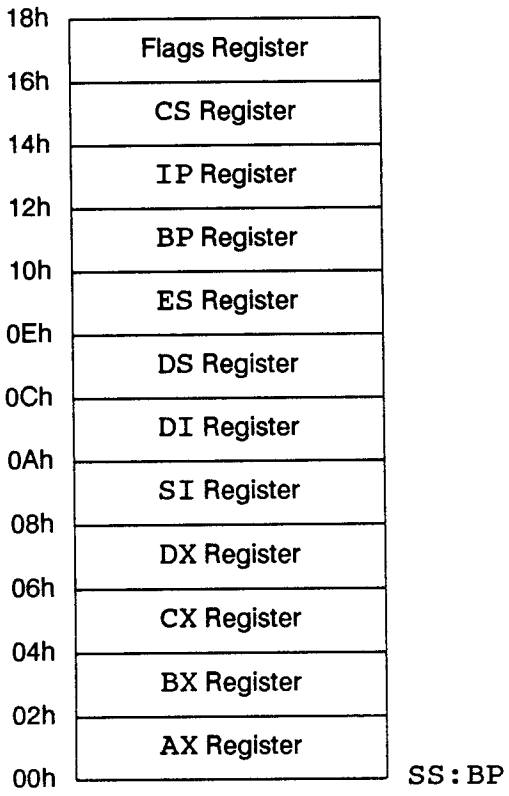


Figure 3-4. Register Save Area

CAUTION Do not alter values in the register save area except those that the handler routine is required to change. Some registers are critical to the proper operation of the calling routines, and changing them can have significant, detrimental side effects (including loss of data).

High-Level Handler Behavior With Unused Registers

Routines in high-level handlers must return to their callers all registers returned by the low-level handler, even if the high-level handler doesn't use or modify any of those registers. The reason is that even if the high-level handler doesn't care about the contents of a particular register, the register may be important to the caller.

This is particularly true of the IOCTL routine, in which the high-level handler may just pass through, unmodified, low-level handler IOCTL requests from an application. If the high-level handler does not similarly pass back the results from the low-level handler, the caller will not see them.

CLOSE

The CLOSE routine in a handler is where the I/O port and the external device are shut down, and control of the port is released by the handler.

Passed to routine:

AL	Channel number to close.

Routine must return:

AL=00h	Successful close.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By the CLOSE function (10h) if a high- or low-level handler name was specified when the handler was opened. The CLOSE function can be invoked either by the BASIC CLOSE # statement or by an assembly language program.
- By a high-level handler using the LLH_CLOSE linkage routine.
- When a program ends and returns to command mode by calling the END_PROGRAM function (00h), the operating system closes all open handlers by calling their CLOSE routines.

Notes:

- Registers specified by the caller of the CLOSE function or the LLH_CLOSE linkage routine are passed to the handler CLOSE routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- When returning to command mode, the operating system calls the CLOSE routines of all open handlers to close them, but does not set AL to the channel number being used. Make sure AL is set to the channel number before calling LLH_CLOSE, or the linkage routine will not call the low-level handler CLOSE routine properly.

  If the high-level handler is only valid for one channel, that valid channel number can be placed in AL before calling LLH_CLOSE. If the high-level handler can be used for more than one channel, the channel number being used should have been saved in the handler's scratch area by its OPEN routine.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Perform device-specific shut down activities.	Disable hardware interrupts for the I/O port.
	Disable and power down the I/O port.
Call low-level handler with LLH_CLOSE linkage routine (see caution below).	Restore original hardware interrupt vectors for the I/O port.
Release high-level handler scratch area.	Restore parameter scratch area address from the low-level handler scratch area into the handler information table.
Return an error code if the routine failed (00h if no errors).	
	Deallocate low-level handler scratch area.
	Return an error code if the routine failed (00h if no errors).

IOCTL

The IOCTL (*I/O control*) routine in a low-level handler allows a program to control the handler operation after the handler has already been opened. This is in addition to providing the handler configuration parameters at open time. High-level handler IOCTL routines only call their low-level handler, since most external devices are controlled by command sequences embedded in data sent to them (via the WRITE function).

Passed to routine: *

AH	IOCTL function code.
AL	Channel number.

Routine must return: *

AL=00h	Successful.
>00h	Error code.
BP	Unchanged from value passed to routine.
AH †	As defined by routine (return in register save area, offset 00h)
BX †	As defined by routine (return in register save area, offset 02h)
CX †	As defined by routine (return in register save area, offset 04h)
DX †	As defined by routine (return in register save area, offset 06h)
SI †	As defined by routine (return in register save area, offset 08h)
DI †	As defined by routine (return in register save area, offset 0Ah)
ES †	As defined by routine (return in register save area, offset 0Eh)

When routine is called:

- By a high-level handler using the LLH_IOCTL linkage routine.
- By an assembly language program using the IOCTL utility routine (see the appendixes). If a high-level handler is called, it passes the call on to the low-level handler by calling LLH_IOCTL.
- Not called by the operating system. IOCTL is one of the three reserved handler routines whose use was not defined until after the operating system was developed; the others are RSVD2 and RSVD3.

Notes:

- Registers specified by the caller of the LLH_IOCTL linkage routine or the IOCTL utility routine are passed on to the low-level handler IOCTL routine with the following exceptions:

* Because each handler implements different handler control functions within its IOCTL routine, other register requirements are defined by the handler itself.

† Returned by high-level handler only.

DS Set to the segment address of the scratch area allocated by the handler.

BP Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.

DI Destroyed.

- Routines in high-level handlers must return to their callers all registers returned by the low-level handler, even if the high-level handler doesn't use or modify any of those registers. The reason is that even if the high-level handler doesn't care about the contents of a particular register, the register may be important to the caller. This is particularly true of the IOCTL routine, in which the high-level handler may just pass through, unmodified, low-level handler IOCTL requests from an application. If the high-level handler does not similarly pass back the results from the low-level handler, the caller will not see them.

Cautions:

- The high-level handler must not change the DS register in the register save area from the caller. Doing so may cause the caller to use the wrong scratch area.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_IOCTL linkage routine.	Perform low-level handler control activities.
Return any registers (in the register save area) that may have been used by the low-level handler routine to the caller.	Return an error code if the routine failed or if a function code was passed that the handler does not implement (00h if no errors).
Return an error code if the routine failed (00h if no errors).	

Reserved IOCTL Functions

Certain I/O control functions have been assigned fixed function codes 00h-06h. Each handler may implement additional functions; refer to the documentation for the particular handler of interest for details. The "Program Resource Allocation" appendix indicates other function codes that have been reserved by other handlers. The fixed function codes are listed below in numeric order.

- IDENTIFY (Function 00h)
  The IDENTIFY function returns two pieces of information to identify handlers: the handler identifier (bytes 2 and 3 of the program header) in CX (byte 2 in CH, byte 3 in CL), and the version number in DX (DH=integer part, DL=fractional part). Hewlett-Packard handlers also return the characters "HP" in BX (BH="H", BL="P"),

- GET_CONFIG (Function 01h)
  The GET_CONFIG function returns the address of the current configuration in ES:DX. Refer to the documentation for each handler for details on the format of the configuration.

The configuration that is returned should be the one saved in the handler's scratch area during its OPEN routine. If the CHANGE_CONFIG function has changed the configuration, the changes would have been made to the saved copy, not the original configuration in the parameter scratch area.

- CHANGE_CONFIG (Function 02h)
  The CHANGE_CONFIG function changes the current handler (and possibly port) configuration while the handler is open. The address of the new configuration is passed in ES:DX. Refer to the documentation for each handler for details on the format of the configuration.

  The configuration that is altered should be the one saved in the handler's scratch area during its OPEN routine. The reason is that configuration changes while the handler is open should not affect the original status defined while it was closed. If a program has initialized a parameter scratch area with certain values prior to opening the handler, the program expects that set of parameters to be unchanged the next time the handler is opened.

- RECEIVE_STATUS (Function 03h)
  The RECEIVE_STATUS function returns the number of bytes in the receive buffer in CX.

- RECEIVE_FLUSH (Function 04h)
  The RECEIVE_FLUSH flushes the receive buffer.

- SEND_STATUS (Function 05h)
  The SEND_STATUS function returns the number of bytes in the send buffer in CX.

- SEND_FLUSH (Function 06h)
  The SEND_FLUSH flushes the send buffer.

The register usage for these functions is summarized below. The AH register is set to the function code, and the AL register is set to the channel number. Like all handler routines, all the registers returned by these functions must be placed in the register save area except AL (for error codes) and BP (which must be unchanged from the value passed to the routine).

Table 3-5. Reserved IOCTL Function Codes

Function Name	Registers Passed		Registers Returned	
	Register	Contents	Register	Contents
CHANGE_CONFIG	AH AL ES DX	02h Channel number Segment address of configuration Offset address of configuration	AL	Error code (00h if no errors)
GET_CONFIG	AH AL	01h Channel number	AL ES DX	00h Segment address of configuration Offset address of configuration
IDENTIFY	AH AL	00h Channel number	AL BX CX DX	00h "HP" * Handler identifier Version number
RECEIVE_FLUSH	AH AL	04h Channel number	AL —	Error code (00h if no errors) Receive buffer cleared
RECEIVE_STATUS	AH AL	03h Channel number	AL CX	00h Number of bytes in receive buffer
SEND_FLUSH	AH AL	06h Channel number	AL —	Error code (00h if no errors) Send buffer cleared
SEND_STATUS	AH AL	05h Channel number	AL CX	00h Number of bytes in send buffer

* Returned by Hewlett-Packard handlers.

OPEN

The OPEN routine in a handler is where the I/O port and the external device are initialized and readied for I/O.

Passed to routine:

AL	Channel number to open.
ES	Segment address of handler name string to open.
BX	Offset address of handler name string to open.
DS	Segment address of parameter area (built-in serial port handler only).
DX	Number of bytes to write.

Routine must return:

AL=00h	Successful open.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By the OPEN function if the caller invoking the function specifies a high- or low-level handler name. The OPEN function can be invoked either by the BASIC OPEN # statement or by an assembly language program.
- By a high-level handler using the LLH_OPEN linkage routine.

Notes:

- Registers specified by the caller of the OPEN function or the LLH_OPEN linkage routine are passed to the handler OPEN routine with the following exceptions:

  BP Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.

  DI Destroyed.

- Handlers allocate one or two scratch areas in their OPEN routine: the parameter scratch area for parameter passing (if not already allocated), and the handler scratch area for its pointers, buffers, etc. The operating system saves the handler's scratch area address in an internal table based on the channel number of the handler (this is not the same as the handler information table). When the other routines in the handler are called (such as READ, WRITE, etc.), the operating system reads the appropriate scratch area address from this internal table, and passes it to the routine.

  If a handler allocates more than one scratch area, only the address of the last one allocated will be saved and automatically passed to handler routines. Therefore, when multiple scratch areas are allocated by a handler, the allocation order is important. A handler can allocate scratch areas so that the last one allocated is the one whose address should be passed to handler routines. Alternatively, the handler can call GET_MEM with the channel number set to 0, and the operating system will not save that scratch area address or pass it to handler routines.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Verify that the channel being opened to is correct for this handler.	Read and verify configuration parameters. If passed in parameter scratch area, use the handler information table and the valid data flag. If passed in-line with the name string, parse the parameters from the string, and convert to the form required by the handler.
Read and verify configuration parameters. If passed in parameter scratch area, use the handler information table and the valid data flag. If passed in-line with the name string, parse the parameters from the string, and convert to the form required by the handler.	Allocate and initialize parameter scratch area if necessary.
Allocate and initialize parameter scratch area if necessary.	Allocate low-level handler scratch area for port-specific needs.
Allocate high-level handler scratch area for device-specific needs.	Save parameter scratch area address from the handler information table in the low-level handler scratch area. This will be needed by the CLOSE routine.
Save channel number the handler is opened to in the high-level handler scratch area. This will be needed by the CLOSE, TERM, and WARM routines. *	Save parameters from the parameter scratch area in the low-level handler scratch area. This will be needed by the IOCTL routine.
Change handler name pointer (ES:BX) to point to the start of the low-level handler name. Skip past the directory number and colon, if any, and any in-line parameters to find the low-level handler name. Return an error if there is no low-level handler name.	Save the low-level handler scratch area address in the handler information table. This will be needed by the interrupt service routine.
Call low-level handler with LLH_OPEN linkage routine. Return an error if no low-level handler with that name exists.	Take over hardware interrupt vectors for the I/O port, and save the previous vector address in the low-level handler scratch area.
Perform device-specific initialization activities.	Initialize the I/O port and provide power to it.
Return an error code if the routine failed (00h if no errors).	Enable hardware interrupts for the I/O port.
	Return an error code if the routine failed (00h if no errors).

* This is only necessary if the high-level handler can be used for more than one channel, such as HNWN. If the handler can be used for only one channel, that channel number need not be saved, since it will always be known.

POWERON

The POWERON routine allows a handler to perform device or port initialization when the machine is turned on, even if the handler is not open.

Passed to routine:

Nothing. *

Routine must return:

Nothing. *

When routine is called:

- Only when the HP-94 is turned on, after all memory integrity checks have been performed, and all battery voltages have been tested. All handlers, whether open or closed, will have their POWERON routine executed at that time. This includes the low-level handler of a high- and low-level handler pair, even though the operating system thinks that only the high-level handler of the pair is open. For this reason, there is no LLH_POWERON linkage routine.
- Not executed if the machine enters command mode because of a memory integrity error or because the [CLEAR] and [ENTER] keys are held down. (The latter prevents an erroneous POWERON routine from permanently preventing access to command mode.) This means that if the machine autostarts MAIN, POWERON will have been executed, but if MAIN is run from command mode using the S (*start*) command, POWERON will not have been executed. POWERON will not have been executed if the program is always started from command mode (e.g., SCOLL to start a program called COLL).

Notes:

- If a high-level handler wants to perform device-specific power-on initialization, it must be done after a low-level handler performs port-specific power-on initialization, or the I/O port may not allow access to the device. The POWERON routines are called in each handler, open or closed, in directories 0-4 in ascending order. Within each directory, handlers are called in ascending directory table entry order. This implies that the low-level handler's POWERON routine would have to be called before the high-level handler's. This will only occur if the low-level handler appears earlier in the same directory as the high-level handler, or in a lower-numbered directory than the high-level handler.
- HP-94 status during the POWERON routine is discussed later.

Cautions:

- Power-on initialization of the HP-94 can be completely altered, with significant, detrimental side effects (including loss or alteration of existing data) if the POWERON routine changes any of the registers that are passed to it. It is therefore imperative that the POWERON routine save and restore any registers that it uses.

* Unlike all other handler routines, no registers are saved before calling POWERON, or restored upon its exit. No registers are passed to the routine, nor are any values expected to be returned by it.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Save any registers used by routine.	Save any registers used by routine.
Perform device-specific power-on initialization activities.	Perform port-specific power-on initialization activities.
Restore original registers.	Restore original registers.

HP-94 Status During POWERON Routine

The machine status when the POWERON routine is called is identical to the status at warm start, *even if the machine was turned off with cold start specified*, with the following exceptions:

- The status of user-defined characters is unchanged. If the machine was turned off during a running program, they will be available (assuming they were present in the machine) during the POWERON routine. If the machine was turned off by pressing the power switch in command mode, they will not be available during POWERON routine.
- Access to directory 5 is enabled. This is the only time when a program is running that directory 5 is accessible.
- The display backlight will be off.

After the POWERON routine is called, the operating system will perform either cold start or warm start initialization, depending on how the machine was turned off. If it should cold start, the cold start status is set, and program MAIN will be autostarted. User-defined characters are located and made available if they exist, access to directory 5 is disabled, and the backlight remains off. If it should warm start, the warm status is left unchanged, and the program continues running at its power-off point. User-defined characters are left in their warm start state, access to directory 5 is disabled, and the backlight is turned on if it was on when the machine turned off.

Because the cold or warm status is set after POWERON is called, some operating system functions cannot be used in the routine. For example, if the machine is going to cold start, all open data files will be closed. If a file was opened during the POWERON routine, it will be closed immediately during cold start initialization. Here is a list of the operating system functions that can be used in the POWERON routine:

Table 3-6. Functions Allowed in POWERON Routine

Function Name	Function Number
BEEP	07h
BUFFER_STATUS	06h
CURSOR	05h
DISPLAY_ERROR	18h
FIND_FILE	16h
FIND_NEXT	17h
GET_CHAR	01h
GET_LINE	02h
MEM_CONFIG	0Dh
PUT_CHAR	03h
PUT_LINE	04h
ROOM	0Eh
SET_INTR	0Ah
TIMEOUT	09h
TIME_DATE	08h

READ

The READ routine in a handler is where the data coming into the I/O port is read and returned to the caller.

Passed to routine:

AL	Channel number to read.
CX	Number of bytes to read.
ES	Segment address of read buffer.
BX	Offset address of read buffer.

Routine must return:

AL=00h	Successful read.
>00h	Error code.
BP	Unchanged from value passed to routine.
CX	The number of bytes actually read (return in register save area, offset 04h).

When routine is called:

- By the READ function if a high- or low-level handler name was specified when the handler was opened. The READ function can be invoked either by the BASIC GET #, INPUT # or INPUT$ statements, or by an assembly language program.
- By a high-level handler using the LLH_READ linkage routine.

Notes:

- Registers specified by the caller of the READ function or the LLH_READ linkage routine are passed to the handler READ routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- The number of bytes to read must not be greater than the actual read buffer length (although it can be less).

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_READ linkage routine. The read buffer specified can be either the caller's buffer or one in the handler's scratch area.	Enable the system timeout. *
	Monitor system events (system timeout, power switch, and low battery) while waiting for incoming data. *
Perform device-specific read activities.	Read the data from the I/O port.
Transfer the data from the high-level handler's buffer (if any) into the caller's read buffer (but no more than the low-level handler returned).	Transfer the data from the low-level handler's buffer (in its scratch area) into the caller's read buffer (but no more than the caller requested).
Return the actual number of bytes read, and an error code if the routine failed (00h if no errors).	Disable the system timeout. *
	Return the actual number of bytes read, and an error code if the routine failed (00h if no errors).
* Refer to the appendixes for information about a utility routine to do this.	

RSVD2

The RSVD2 routine in a handler is the second routine reserved for future use, the first (with a use now assigned) being IOCTL, and the third being RSVD3.

Passed to routine: *

AL	Channel number.

Routine must return: *

AL=00h	Successful write.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By a high-level handler using the LLH_RSVD2 linkage routine.
- Not called by the operating system or by any utility routines.

Notes:

- Registers specified by the caller of the LLH_RSVD2 linkage routine are passed on to the handler RSVD2 routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Activities of routine: *

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_RSVD2 linkage routine.	Return an error code if the routine failed (00h if no errors).
Return an error code if the routine failed (00h if no errors).	

* Because these routines have not yet been defined, other register requirements and activities may be defined at a later date.

RSVD3

The RSVD3 routine in a handler is the third routine reserved for future use, the first (with a use now assigned) being IOCTL, and the second being RSVD2.

Passed to routine: *

AL	Channel number.

Routine must return: *

AL=00h	Successful write.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By a high-level handler using the LLH_RSVD3 linkage routine.
- Not called by the operating system or by any utility routines.

Notes:

- Registers specified by the caller of the LLH_RSVD3 linkage routine are passed on to the handler RSVD3 routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Activities of routine: *

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_RSVD2 linkage routine.	Return an error code if the routine failed (00h if no errors).
Return an error code if the routine failed (00h if no errors).	

* Because these routines have not yet been defined, other register requirements and activities may be defined at a later date.

The TERM routine in a handler is used to halt I/O in progress when low battery or power switch interrupts occur.

Passed to routine:

AL	Cause of termination (0=low battery, 1=power switch pressed).

Routine must return:

AL=00h	Successful.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By the operating system when low battery occurs.
- By the operating system when the power switch is pressed, unless the program disabled the power switch using the SET_INTR function (0Ah).
- By a high-level handler using the LLH_TERM linkage routine.
- *Not* called by the operating system when the system timeout occurs. Since each handler must monitor the system timeout itself, that handler will be the only one waiting on I/O when the timeout expires. Consequently, it is the only one that needs to terminate I/O.

Notes:

- Registers specified by the caller of the TERM function or the LLH_TERM linkage routine are passed on to the handler TERM routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- When low battery or power switch occurs, the operating system calls the TERM routines of all open handlers, but does not set AL to the channel number being used. Instead, it sets AL to the cause of the termination (0=low battery, 1=power switch). Place the cause of the termination into AH, and make sure AL is set to the channel number before calling LLH_TERM, or the linkage routine will not call the low-level handler TERM routine properly. LLH_TERM will swap the values so that the low-level handler's TERM routine will receive the cause of the termination in AL.

  If the high-level handler is only valid for one channel, that valid channel number can be placed in AL before calling LLH_TERM. If the high-level handler can be used for more than one channel, the channel number being used should have been saved in the handler's scratch area by its OPEN routine.

...TERM

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_TERM linkage routine (see caution below).	Halt I/O in progress.
	Clean up incomplete data.
Perform device-specific termination activities.	Return an error code if the routine failed (00h if no errors).
Return an error code if the routine failed (00h if no errors).	

The WARM routine in a handler is where the I/O port and the external device are reinitialized to their open state and readied for I/O when the HP-94 warm starts.

Passed to routine:

AL	Channel number.

Routine must return:

AL=00h	Successful.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By the operating system when the HP-94 turns on with a warm start, after the POWERON routine has been called. The WARM routines of any high- or low-level handlers that were open at power off are called after the operating system performs all memory integrity tests and sets all warm start status, just before returning control to the program that turned the power off. Refer to the "Program Execution" chapter for details on machine status at warm start.
- By a high-level handler using the LLH_WARM linkage routine.

Notes:

- Registers specified by the caller of the WARM function or the LLH_WARM linkage routine are passed to the handler WARM routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- At warm start, the operating system calls the WARM routines of all open handlers, but does not set AL to the channel number being used. Make sure AL is set to the channel number before calling LLH_WARM, or the linkage routine will not call the low-level handler WARM routine properly.

  If the high-level handler is only valid for one channel, that valid channel number can be placed in AL before calling LLH_WARM. If the high-level handler can be used for more than one channel, the channel number being used should have been saved in the handler's scratch area by its OPEN routine.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities *
Call low-level handler with LLH_WARM linkage routine (see caution below).	Take over hardware interrupt vectors for the I/O port, and save the previous vector addresses in the low-level handler scratch area.
Perform device-specific initialization activities.	
Return an error code if the routine failed (00h if no errors).	Initialize the I/O port and provide power to it.
	Enable hardware interrupts for the I/O port.
	Return an error code if the routine failed (00h if no errors).

* The status of I/O devices at warm start is the same as at cold start. It is the responsibility of the handler to restore I/O devices to their proper state (power, interrupt vector addresses, and interrupt enable/disable status).

WRITE

The WRITE routine in a handler is where the data is sent out the I/O port to the external device.

Passed to routine:

AL	Channel number to write.
CX	Number of bytes to write.
ES	Segment address of write buffer.
BX	Offset address of write buffer.

Routine must return:

AL=00h	Successful write.
>00h	Error code.
BP	Unchanged from value passed to routine.
CX	The number of bytes actually written (return in register save area, offset 04h).

When routine is called:

- By the WRITE function (13h) if a high- or low-level handler name was specified when the handler was opened. The WRITE function can be invoked either by the BASIC PRINT #, PRINT #...USING or PUT # statements, or by an assembly language program.
- By a high-level handler using the LLH_WRITE linkage routine.

Notes:

- Registers specified by the caller of the WRITE function or the LLH_WRITE linkage routine are passed to the handler WRITE routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- The number of bytes to write must not be greater than the actual write buffer length (although it can be less).

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Perform device-specific write activities.	Enable the system timeout. *
Call low-level handler with LLH_WRITE linkage routine. The write buffer specified can be either the caller's buffer or one in the handler's scratch area.	Monitor system events (system timeout, power switch, and low battery) while outputting data. *
	Write the data to the I/O port.
Return the actual number of bytes written, and an error code if the routine failed (00h if no errors).	Disable the system timeout. *
	Return the actual number of bytes written, and an error code if the routine failed (00h if no errors).

* Refer to the appendixes for information about a utility routine to do this.

4

Operating System Functions

Contents

Chapter 4 Operating System Functions

4

Operating System Functions

This chapter describes the operating system functions. These functions allow assembly language programs to simplify the interaction between assembly language programs and the HP-94 hardware: memory, keyboard, display (and display backlight), serial port, bar code port, power switch, low battery detection, real-time clock, and beeper. The BASIC interpreter also uses these functions to provide analogous capability to BASIC language programs.

Operating System Function Usage

Operating system functions are called by the following procedure:

- Load function code into register AH.
- Load any other function parameters into the corresponding registers.
- Issue a software interrupt `1Ah`.

When functions end, they pass results back in the registers listed in the function descriptions.

Operating System Function Descriptions

Function descriptions consist of:

- A brief description of the operating system function.
- A summary of the function call parameters.
- A summary of the function return parameters including any possible returned error codes.
- Supplementary notes and cautions on the use and behavior of the function.
- A list of related operating system functions.
- An example of the use of the operating system function. These examples are provided only to illustrate typical use of the various functions. Several of the examples contain data scratch areas embedded in the code, and consequently will only work if executed in RAM — they will not run in ROM or EPROM.

Registers Passed to Operating System Functions

Each operating system function saves the contents of all the registers passed to it, and returns those values to the caller when the function ends. The only registers altered by the functions are those that explicitly return particular values to the caller — all other registers will retain their original values. AL is always used to return error codes.

BEEP

Beep a high or low tone for a specified duration.

Call with:

AH=07h	BEEP function code.
AL=00h	Low tone.
=01h	High tone.
BL	Length of tone in 0.1 second units (0.1 - 25.5 seconds).

Returns:

Nothing.

Notes:

- When AL is greater than 01h, no action is performed.

Cautions:

- As soon as BEEP starts the beeper, the application program will continue to run; that is, the program does not wait for the beep to finish before resuming execution.
- BEEP can be called while the beeper is beeping. If the tone specified is different than the tone in progress, beeping will continue at the high tone and duration - the high tone and its duration will always take precedence, regardless of the order in which the tones are specified. If the tone specified is the same as the tone in progress, beeping will continue at either the remaining duration or the new duration, whichever is longer.

Related functions:

None.

Example:

The following example will do a one-second low beep.

```
BEEP            equ     07h             ;BEEP function code
LOTONE          equ     00h
HITONE          equ     01h
                .
                .
                .
                mov     ah,BEEP         ;BEEP function code
                mov     al,LOTONE       ;low tone...
                mov     bl,10           ;for 1 second
                int     1Ah             ;beep it.
                .
                .
                .
```

BUFFER_STATUS

Get the number of bytes in or flush either the key buffer or the receive buffer for the built-in serial port handler.

Call with:

AH=06h	BUFFER_STATUS function code.
AL=00h	Flush key buffer.
=01h	Get the number of bytes in the key buffer.
=02h	Flush the receive buffer for the built-in serial port handler.
=03h	Get the number of bytes in the receive buffer for the built-in serial port handler.

Returns:

DL	Number of bytes in the key buffer (AL=01h) or the receive buffer for the built-in serial port handler (AL=03h).

Notes:

- The operations performed when AL is 02h or 03h only apply to the buffer for the built-in serial port handler. For user-defined serial port handlers with their own buffers, these operations will not work.
- When AL is greater than 03h, no action is performed.

Related functions:

GET_CHAR, GET_LINE, READ

Example:

The following example will flush any characters in the key buffer and serial port receive buffer.

```
BUFFER_STATUS       equ     06h             ;BUFFER_STATUS function code
KBD_FLUSH           equ     00h
KBD_STAT            equ     01h
SER_FLUSH           equ     02h
SER_STAT            equ     03h
                    .
                    .
                    .
;
;                   initialize the key buffer and serial port receive buffer
;
                    mov     ah,BUFFER_STATUS    ;BUFFER_STATUS function code
                    mov     al,KBD_FLUSH
                    int     1Ah                 ;flush keyboard buffer
                    mov     al,SER_FLUSH
                    int     1Ah                 ;flush serial port receive buffer
                    .
                    .
                    .
```

CLOSE

Close and release an open channel.

Call with:

AH=10h	CLOSE function code.
AL	Channel number to close.

Returns:

AL=00h	Successful close.
=65h (101)	Illegal parameter.
=69h (105)	Channel not open.

Notes:

When closing channels 1 - 4, CLOSE will transfer control to the CLOSE routine of the user-defined handler specified when the channel was opened. The same registers passed to the CLOSE function will be passed to the user-defined handler CLOSE routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Refer to the "User-Defined Handlers" chapter for details.

- When closing channels 1 - 4 and the user-defined handler has returned from its CLOSE routine, the handler will no longer be in control of the device.
- Once a channel is closed, it may not be accessed until it is reopened.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CREATE, DELETE, OPEN, READ, SEEK, WRITE

Example:

The following example procedure will close a file.

```
CLOSE           equ     10h             ;CLOSE function code
                .
                .
                .
;               fclose -- close an open file
;
;               call with:
;                       al = channel #
;
fclose          proc    near
                mov     ah,CLOSE        ;CLOSE function code
                int     1Ah             ;close the file
                or      al,al           ;set status for caller
                ret
fclose          endp
                .
                .
                .
```

CREATE

Allocate initial storage for a data file and build the directory table entry for the file.

Call with:

AH=11h	CREATE function code.
ES	Segment address of file name to create.
BX	Offset address of file name to create.
CX	Initial allocated size in paragraphs.
DX	Size increment in paragraphs.

Returns:

AL=00h	Successful create.
=65h (101)	Illegal parameter.
=66h (102)	Directory does not exist.
=68h (104)	Too many files. The directory is full.
=6Ch (108)	File already exists.
=6Dh (109)	Read-only access.
=6Fh (111)	No room for file.
CX	Start segment address of the file.
BX	Available free space in paragraphs (when AL=6Fh).

Notes:

- The file name must be uppercase.
- The file name must be terminated by either a null (00h) or a space (20h).
- Wild card characters are not allowed in the file name.
- If the file name is longer than 4 characters, only the first 4 characters (plus a leading directory number and colon, if any) will be used.
- If the file name contains a directory specifier, the file will be created in that directory. If the file name does not contain a directory specifier, the file will be created in directory 0 (main memory).
- The size increment is the expansion increment used when data is written past the end-of-file.
- CREATE will fail if an attempt is made to create a data file in a ROM or EPROM.
- A data file must be opened before it can be written to or read from.
- Allocated file space is automatically initialized to nulls (00h).
- If there is not enough free space in the directory to create a file with the specified number of paragraphs, CREATE will return 6Fh (111) in AL along with the number of available paragraphs in BX.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CLOSE, DELETE, OPEN, READ, SEEK, WRITE

Example:

The following example procedure will create a file.

```
CREATE          equ     11h             ;CREATE function code
                .
                .
                .
;               fcreate -- create a file
;
;               call with:
;                       bx = offset address of file name buffer
;                       cx = initial size in paragraphs
;                       dx = size increment in paragraphs
;
fcreate         proc    near
                mov     ah,CREATE       ;CREATE function code
                push    cs              ;segment address of file name to ES
                pop     es
                int     1Ah             ;create the file
                or      al,al           ;set status for caller
                ret
fcreate         endp
                .
                .
                .
```

CURSOR

Move the display cursor or obtain its current position.

Call with:

AH=05h	CURSOR function code.
AL=00h	Get the current display cursor position.
=01h	Move the display cursor.
CL	Cursor column position 0 - 19 (for move cursor).
CH	Cursor row position 0 - 3 (for move cursor).

Returns:

CL	Current cursor column position 0 - 20 (for get cursor).
CH	Current cursor row position 0 - 3 (for get cursor).

Notes:

- When AL is greater then 01h, no action is performed.
- When an attempt is made to move the cursor outside the range of the display window, no action is performed.
- When a character is displayed in the last column of the display, the cursor will remain in that column until another character is displayed. In this case, it is considered to be in column position 20.

Related functions:

PUT_CHAR, PUT_LINE

Example:

The following example will move the cursor to column 0 of the current line.

```
CURSOR          equ     05h             ;CURSOR function code
                .
                .
                .
                mov     ah,CURSOR       ;CURSOR function code
                mov     al,00h          ;to get current cursor position
                int     1Ah             ;get cursor position
                mov     cl,00h          ;set to column 0
                mov     al,01h          ;to set cursor position
                int     1Ah             ;and set the new position
                .
                .
                .
```

DELETE

Delete a currently open data file. The area occupied by the file will be returned to the free space in the directory containing the file.

Call with:

AH=14h	DELETE function code.
AL	Channel number of the open file to delete.

Returns:

AL=00h	Successful delete.
=65h (101)	Illegal parameter. This error will also occur for deletes of channels 0 - 4.
=69h (105)	Channel not open.
=6Dh (109)	Read-only file.
=6Eh (110)	Access restricted.

Notes:

- The released area is merged with the other free space in the directory each time a file is deleted.
- An automatic CLOSE occurs after a DELETE.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CLOSE, CREATE, OPEN, READ, SEEK, WRITE

Example:

The following example will delete a file by first opening and then deleting it.

```
DELETE          equ     14h             ;DELETE function code
OPEN            equ     0Fh             ;OPEN function code
                .
                .
                .
                mov     ah,OPEN         ;OPEN function code
                mov     al,5            ;use channel 5
                push    ds              ;put fname segment address into ES
                pop     es
                                        ;fname address offset into BX
                mov     bx,offset byte ptr fname
                int     1Ah             ;open the file
                or      al,al           ;successful open?
                jne     open_err        ;no -- handle the open error
                mov     ah,DELETE       ;DELETE function code
                mov     al,5            ;the open channel
                int     1Ah             ;delete the file
                or      al,al           ;successful delete?
                jne     del_err         ;no -- handle the delete error
                .
                .
                .
```

DISPLAY_ERROR

Display the specified numeric error code. The displayed message will be of the form `Error nnn`, where `nnn` is the decimal value of the error code.

Call with:

AH=18h	DISPLAY_ERROR function code.
AL	Error code.

Returns:

Nothing.

Notes:

- The error number in the displayed message will always be three decimal digits. For error codes less than 64h (100), leading zeroes will be added.
- Before displaying an error message the cursor is moved to the first column of the next line in the display. After displaying the message, the cursor will be placed in the first column to the right of the error message.
- A beep will occur when the error message is displayed.

Related functions:

None.

Example:

The following example will display error 101 (illegal parameter).

```
DISPLAY_ERROR       equ     18h             ;DISPLAY_ERROR function code
                    .
                    .
                    .
                    mov     ah,DISPLAY_ERROR    ;DISPLAY_ERROR function code
                    mov     al,101          ;illegal parameter error code
                    int     1Ah             ;and display the error message
                    .
                    .
                    .
```

END_PROGRAM

Terminate an application program. The application program can turn off the HP-94, specifying either cold or warm start at the next power on, or return to command mode.

Call with:

AH=00h	END_PROGRAM function code.
AL=00h	Cold start.
=01h	Warm start.
>01h	End application program and enter command mode.

Returns:

Nothing.

Notes:

- When cold start is selected, the HP-94 will be turned off. When power is turned back on, the program MAIN will be run if it exists, or command mode will be entered.
- When warm start is selected, all registers and flags will be stored by the HP-94 before turning off power. When power is turned back on, the registers and flags will be restored and program execution resumed.

Cautions:

- Warm start may not be used by an application program which uses the HP-94 operating system's stack. An application which is initialized from command mode must allocate its own stack area in order to do a warm start END_PROGRAM. If a warm start is attempted using the operating system's stack, error 219 will be generated and control will return to command mode.
- This function may not be called from the POWERON routine of a handler.
- A FAR RET can also be used to end a program, but there are some subtle side effects. Refer to "Program Execution" for details.

Related functions:

None.

Example:

The following example will end the program and return to command mode.

```
END_PROGRAM     equ     00h             ;END_PROGRAM function code
COLDSTART       equ     00h
WARMSTART       equ     01h
CMDMODE         equ     02h
                .
                .
                .
                mov     ah,END_PROGRAM  ;END_PROGRAM function code
                mov     al,CMDMODE      ;to enter command mode
                int     1Ah
                .
                .
                .
```

FIND_FILE

Find the first file to match a file name pattern. Wild card characters may be included in the file name.

Call with:

AH=16h	FIND_FILE function code.
ES	Segment address of the search file name.
BX	Offset address of the search file name.
DS	Segment address of the file information buffer.
DX	Offset address of the file information buffer.

Returns:

AL=00h	Successful FIND_FILE.
=65h (101)	Illegal parameter.
=66h (102)	Directory does not exist.
=67h (103)	File not found.
CX	Segment address of the directory table entry for the matched file.
DX	Offset address of the directory table entry for the matched file.

Notes:

- The search file name must be uppercase.
- There is one wild card character, "*", which matches any character at that position and all subsequent positions in the file name.
- If the search file name contains a directory specifier, only that directory will be searched. If it does not contain a directory specifier, directories 0 - 4 will be searched in ascending order. The system directory (directory 5) will not be searched by FIND_FILE.
- If the file name is longer than 4 characters, only the first 4 characters will be used.
- The file information buffer consists of 14 bytes formatted as follows:

Bytes	Data
00h - 06h	Name of first file found which matches the search file name.
07h	File type
08h - 09h	Start segment address of the file
0Ah - 0Bh	Low word of the end-of-data address
0Ch	High byte of the end-of-data address
0Dh	NUL (00h)

File names are of the form "d:name" where "d" is the directory number and "name" is the 1 - 4 byte file name terminated by a null (00h).

Related functions:

FIND_NEXT

Example:

The following example will search for the first file which matches a specific file name. It is part of the example included with the FIND_NEXT function. Since the code contains a scratch buffer and the file information buffer, it will not work in ROM.

```
FIND_FILE       equ     16h             ;FIND_FILE function code
                .
                .
                .
                mov     ah,FIND_FILE    ;FIND_FILE function code
                push    cs              ;ES:BX address of scratch buffer
                pop     es
                mov     bx,offset buffer
                push    cs              ;DS:DX address of file info buffer
                pop     ds
                mov     dx,offset fbuffer
                int     1Ah             ;find the first file
                call    errchk          ;check for error
                .
                .
                .
buffer          db      BUFSIZ+1 dup (?)    ;read buffer
                                            ;must be in RAM
fbuffer         db      14 dup (?)          ;find filename dest buffer
                                            ;must be in RAM
                .
                .
                .
```

FIND_NEXT

Find the next file to match a file name pattern set up by a FIND_FILE function call.

Call with:

AH=17h	FIND_NEXT function code.

Returns:

AL=00h	Successful FIND_NEXT.
=67h (103)	File not found.
CX	Segment address of the directory table entry for the matched file.
DX	Offset address of the directory table entry for the matched file.

Notes:

- The format of the file information buffer is the same as that of the FIND_FILE function.
- The FIND_FILE function must be executed before FIND_NEXT.
- FIND_NEXT will return data in the area specified by the last FIND_FILE function call.

Cautions:

- FIND_NEXT will search only in the directory in which FIND_FILE found the first matching file name.

Related functions:

FIND_FILE

Example:

The following example will prompt for a file name and use FIND_FILE and FIND_NEXT to find and display all the files which match that file name. Since the code contains a scratch buffer and the file information buffer, it will not work in ROM.

```
GET_CHAR        equ     01h             ;GET_CHAR function code
ECHO            equ     00h
NOECHO          equ     01h
GET_LINE        equ     02h             ;GET_LINE function code
PUT_LINE        equ     04h             ;PUT_LINE function code
FIND_FILE       equ     16h             ;FIND_FILE function code
FIND_NEXT       equ     17h             ;FIND_NEXT function code
DISPLAY_ERROR   equ     18h             ;DISPLAY_ERROR function code
BUFSIZ          equ     80

code            segment
                assume cs:code,ds:code
prog            proc    far
start:

                dw      prgmend-start
                dw      0006h           ;offset of internal entry point
                db      0100h           ;version 1.00

                push    cs              ;set DS to CS
                pop     ds
```

```
                        ; display "DIR> " prompt
                        mov         bx,offset DIR
                        call        puts
                        ; get name of file
                        mov         bx,offset buffer
                        mov         al,BUFSIZ
                        call        gets
                        call        errchk
                        ; null out the terminating CR
                        sub         dh,dh               ;clear out dh
                        mov         di,dx               ;and put it into di
                        mov         byte ptr buffer[di],00h;null out last byte
                        ; go through buffer and substitute ":" for "."
                        sub         dx,dx               ;clear out dx
                        mov         di,dx               ;and put it into di
dots01:
                        mov         al,byte ptr buffer[di];get the next character
                        cmp         al,"."              ;is it a dot?
                        jne         nodot               ;no, don't swap it
                        mov         al,":"              ;get a ":"
                        mov         byte ptr buffer[di],al;and save it
nodot:
                        inc         di                  ;increment di
                        or          al,al               ;is al null?
                        jne         dots01              ;no, check next byte

                        mov         ah,FIND_FILE        ;FIND_FILE function code
                        push        cs                  ;ES:BX address of buffer
                        pop         es
                        mov         bx,offset buffer
                        mov         dx,offset fbuffer   ;DS:DX address of file info buffer
                        int         1Ah                 ;find the first file
                        call        errchk              ;check for error
                        mov         byte ptr lcnt,5     ;reset display line counter
loop01:
                        dec         byte ptr lcnt       ; decrement line counter
                        jne         pause01             ;not 0 -- don't pause
                        mov         ah,GET_CHAR         ;GET_CHAR function code
                        mov         al,NOECHO           ;don't echo
                        int         1Ah                 ;get a character
                        mov         byte ptr lcnt,4     ;reset the counter (less this line)
                        cmp         dl,"."              ;was it a dot?
                        jne         pause01             ;no -- leave the counter as is
                        mov         byte ptr lcnt,1     ;only 1 line
pause01:
                        ; display file name from fbuffer
                        mov         bx,offset CRLF      ;display a cr/lf
                        call        puts
                        mov         bx,offset fbuffer
                        call        puts
                        mov         ah,FIND_NEXT        ;FIND_NEXT function code
                        int         1Ah                 ;find the next file
                        or          al,al               ;returned a 0?
                        je          loop01              ;yes -- display the file (if found)
                        mov         al,GET_CHAR         ;GET_CHAR function code
                        mov         ah,NOECHO           ;don't echo
                        int         1Ah                 ;wait for a key
exit:
                        ret                             ;if no more files, done.
prog                    endp
```

```
puts            proc    near
                mov     ah,PUT_LINE         ;PUT_LINE function code
                push    cs                  ;segment address of buffer to ES
                pop     es
                int     1Ah                 ;display it
                ret
puts            endp

gets            proc    near
                mov     ah,GET_LINE         ;GET_LINE function code
                push    cs                  ;CS to ES
                pop     es
                int     1Ah                 ;get a line
                ret
gets            endp

errchk          proc    near
                or      al,al               ;return code 0?
                je      errret              ;yes -- just return
                mov     ah,DISPLAY_ERROR    ;DISPLAY_ERROR function code
                int     1Ah                 ;display the error message
                add     sp,2                ;pull off the near return address
                jmp     exit                ;terminate program
errret:
                ret
errchk          endp

lcnt            db      ?
DIR             db      "DIR> ",0Fh,00h     ;prompt, alpha mode
CRLF            db      0Dh,0Ah,00h         ;cr/lf
buffer          db      BUFSIZ+1 dup (?)    ;read buffer
                                            ;must be in RAM
fbuffer         db      14 dup (?)          ;find filename dest buffer
                                            ;must be in RAM

prgmend:
code            ends
                end
```

GET_CHAR

Get one character from the key buffer and optionally echo it to the display.

Call with:

AH=01h	GET_CHAR function code.
AL=00h	Echo the character being read.
>00h	Do not echo the character being read.

Returns:

AL=00h	Successful read.
=76h (118)	Timeout. A timeout occurred before a key was pressed.
=77h (119)	Power switch pressed.
=C8h (200)	Low battery.
DL	Character read from the key buffer and optionally echoed to the display.

Notes:

- When the key buffer is empty, GET_CHAR will wait for a key.
- The [SHIFT] key cannot be read.
- The keys [CLEAR], [←], and [ENTER], return the codes 18h, 7Fh, and 0Dh respectively. They are never echoed to the display, nor are any control codes (00h-1Fh) or user-defined characters (80h-8Fh), the first 16 of which correspond to user-defined keys.
- The following behavior only applies when echoing to the display: When a character is echoed to the last column of a line in the display, the cursor will remain over that character. The display will be scrolled, if necessary, and the cursor moved to the first column of the next line before echoing the next character.

Related functions:

BUFFER_STATUS, GET_LINE

...GET_CHAR

Example:

The following example will wait for the ENTER key to be pressed.

```
GET_CHAR        equ     01h             ;GET_CHAR function code
ECHO            equ     00h
NOECHO          equ     01h
                .
                .
                .
entwait:
                mov     ah,GET_CHAR     ;GET_CHAR function code
                mov     al,NOECHO       ;don't echo the character read
                int     1Ah             ;read a character
                or      al,al           ;read error (al <> 0)?
                jne     rd_err          ;yes -- process read error
                cmp     dl,0Dh          ;[ENTER] key?
                jne     entwait         ;no -- wait for another key
                .
                .
                .
```

GET_LINE

Get a character string from the keyboard buffer and echo it to the display.

Call with:

AH=02h	GET_LINE function code.
AL	Maximum number of bytes to read (1 - 255 bytes).
ES	Segment address of the read buffer.
BX	Offset address of the read buffer.

Returns:

AL=00h	Successful read.
=76h (118)	Timeout. A timeout occurred before a key was pressed.
=77h (119)	Power switch pressed.
=C8h (200)	Low battery.
DL	Number of characters read from keyboard buffer.

Notes:

- If the key buffer does not contain an [ENTER], GET_LINE will wait until [ENTER] is pressed.
- The terminating [ENTER] will not be echoed to the display. The cursor will be left at the column to the right of the character before the [ENTER].
- The buffer must contain enough space for the maximum number of bytes to read (register AL), as well as one byte for the terminating [ENTER].
- The character count returned in DL does not include the terminating [ENTER].
- The buffer returned by GET_LINE will contain the terminating [ENTER] (0Dh).
- When AL characters have been read and [ENTER] has not been pressed, subsequent characters will be discarded, and a low beep issued, until [ENTER] is pressed.
- GET_LINE processes [←] and [CLEAR] separately. If there are any characters in the buffer, [←] will remove the last character from the input buffer, erase the character from the display and move the display cursor back one character position. [CLEAR] will clear the entire input buffer and erase the entire input line from the display.
- When a character is displayed in the last column of a line in the display, the cursor will remain over that character. The display will be scrolled, if necessary, and the cursor move to the first column of the next line before displaying the next character.
- If timeout, power switch, or low battery interrupts occur, the buffer will contain any characters already entered. The DL register will contain the number of characters actually read.

Cautions:

- GET_LINE will not check for wraparound of the read buffer's offset address.

...GET_LINE

Related functions:

BUFFER_STATUS, GET_CHAR

Example:

The following example will read in a 20-character string. Since the code contains the read buffer for GET_LINE, it will not work in ROM.

```
GET_LINE        equ     02h             ;GET_LINE function code
BUFSIZ          equ     20
buffer          db      BUFSIZ+1 dup (?)    ;must be in RAM
                .
                .
                .
                mov     ah,GET_LINE     ;GET_LINE function code
                mov     al,BUFSIZ       ;size of buffer
                push    ds              ;set ES to DS
                pop     es
                mov     bx,offset buffer    ;buffer offset to BX
                int     1Ah             ;read string
                or      al,al           ;read error (al <> 0)?
                jne     rd_err          ;yes -- process read error
                .
                .
                .
```

GET_MEM

Allocate a scratch area of memory.

Call with:

AH=0Bh	GET_MEM function code.
AL	Channel number if the request is being made by a handler.
=00h	If the request is not being made by a handler.
BX	Size of the requested area in paragraphs.

Returns:

AL=00h	Successful allocation.
=65h (101)	Illegal parameter. Invalid channel number.
=6Eh (110)	Access restricted. No scratch areas available or main memory not initialized.
=71h (113)	No room for scratch area.
CX	Segment address of scratch area.
DX	Length in paragraphs of scratch area.

Notes:

- A maximum of 34 scratch areas may be allocated.
- Scratch memory is automatically initialized to nulls (00h).
- Handlers should set AL to the channel number to which they are open, or to zero, depending on whether or not they want the operating system to pass this scratch area address to all their routines. See the "User-Defined Handlers" chapter for details.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

REL_MEM

Example:

The following example will allocate a 10-paragraph (160-byte) scratch area.

```
GET_MEM         equ     0Bh             ;GET_MEM function code
SCRSIZ          equ     0Ah             ;scratch area size (10 paragraphs)
                .
                .
                .
                mov     ah,GET_MEM      ;GET_MEM function code
                mov     al,00h          ;called by an application (not a handler)
                mov     bx,SCRSIZ       ;size of scratch area
                int     1Ah
                or      al,al           ;error?
                jne     get_mem_err     ;yes -- handle it
                ;
                ;       CX=segment address of scratch area initialized to nulls
                ;       DX=length of allocated scratch area (0Ah)
                ;
                .
                .
                .
```

MEM_CONFIG

Get the current memory configuration of the HP-94.

MEM_CONFIG returns 5 bytes of configuration information. Bytes 0 - 4 describe the contents of directories 0 - 4 respectively as follows.

Value		Meaning
Hex	ASCII	
00h	NUL	No memory installed
4Dh	M	Main memory
41h	A	40K RAM card
4Fh	O	ROM/EPROM card

Call with:

AH=0Dh	MEM_CONFIG function code.
ES	Segment address of the 5-byte configuration buffer.
BX	Offset address of the 5-byte configuration buffer.

Returns:

AL	Number of directories with memory installed. This value is the same as the number of bytes in the configuration buffer which contain a non-zero value.

Related functions:

ROOM

...MEM_CONFIG

Example:

The following program will display the number of installed directories followed by the type of each directory. Since the code contains the configuration buffer, it will not work in ROM.

```
MEM_CONFIG      equ     0Dh             ;MEM_CONFIG function code
PUT_CHAR        equ     03h             ;PUT_CHAR function code
END_PROGRAM     equ     00h             ;END_PROGRAM function code
CMDMODE         equ     02h
code            segment
                assume  cs:code,ds:code
mem             proc    far
start:
                dw      prgmend-start
                dw      0006h           ;offset of internal entry point
                dw      0100h           ;version 1.00

                mov     ax,cs           ;set DS to CS
                mov     ds,ax
                mov     es,ax           ;set ES to segment addr of membuf
                mov     bx,offset membuf ;set BX to offset addr of membuf
                mov     ah,MEM_CONFIG   ;MEM_CONFIG function code
                int     1Ah             ;get it
                add     al,"0"          ;turn al into a number
                mov     ah,PUT_CHAR     ;PUT_CHAR function code
                int     1Ah             ;display it
                mov     al,":"          ;display a ":"
                mov     ah,PUT_CHAR
                int     1Ah
                mov     ax,offset membuf ;set DI to offset addr of membuf
                mov     di,ax
                mov     cx,5            ;number of bytes to check
mem01:
                mov     al,byte ptr [di] ;get it
                cmp     al,00h          ;is it 00h?
                jne     notzero         ;no, leave it alone
                mov     al,"-"          ;change it to a "-"
notzero:
                mov     ah,03h          ;PUT_CHAR function code
                int     1Ah             ;display it
                inc     di              ;increment offset (DI)
                loop    mem01           ;and do the next character
                mov     ah,END_PROGRAM  ;enter command mode
                mov     al,CMDMODE
                int     1Ah

membuf          db      5 dup (?)       ;must be in RAM

prgmend:
mem             endp
code            ends
                end
```

OPEN

Open a data file or handler and assign it to a specific channel.

Call with:

AH=0Fh	OPEN function code.
AL	Channel number to open.
ES	Segment address of file or handler name string to open.
BX	Offset address of file or handler name string to open.
DS	Segment address of parameter area (built-in serial port handler only).
DX	Offset address of parameter area (built-in serial port handler only).

Returns:

AL=00h	Successful open.
=65h (101)	Illegal parameter.
=66h (102)	Directory does not exist.
=67h (103)	File or handler not found.
=6Ah (106)	Channel already open.
=6Bh (107)	File or handler already open.
=6Eh (110)	Access restricted. The specified file is not a data file or handler.
CX	Segment address of the data file or handler.

Notes:

- The OPEN function will search for the file (type D) or handler (type H) with the specified name in directories 0 - 4 in ascending order, or only in a specified directory (e.g., "2:ABCD").
- Channel 0 (keyboard for read operations, display for write operations) is always open. If channel 0 is opened, AL will always return zero. The handler name string is ignored when opening channel 0.
- When opening channels 1 - 4, if the handler is not found, or if a null string was specified as the handler name, the default handler will be used. For channel 1, the default handler is the built-in serial port handler. For channels 2 - 4, there is no built-in handler, and the OPEN function will report error 65h.

  Once the handler is found (either user-defined or built-in), the OPEN function will transfer control to the OPEN routine of the handler. The handler will then become associated with the device, and the CLOSE, READ, and WRITE functions will transfer control to the CLOSE, READ, and WRITE routines of the handler. The same registers passed to the OPEN function will be passed to the user-defined handler OPEN routine with the following exceptions:

BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

  Refer to the "User-Defined Handlers" chapter for details.

- If the name string is longer than 4 characters, only the first 4 characters (plus a leading directory number and colon, if any) will be used by the OPEN function. The entire handler name string (pointed to by ES : BX) will be passed to the OPEN routine of user-defined handlers. This name string can include a high- and low-level handler pair (such as "HNWN;HNBC"), or in-line parameters if the handler allows them (e.g., "RSHN 9600,7ES").
- Alphabetic characters in the name string must be uppercase.
- The file or handler name part of the name string must be terminated by either a null (00h) or a space (20h).
- The wild card character "*" is not allowed in the file or handler name part of the name string.
- The parameter area address (DS : DX) is only used when the built-in serial port handler is opened. The meanings of the parameters are defined in the "Serial Port" chapter. Refer to the "User-Defined Handlers" chapter for a discussion of passing configuration parameters to user-defined handlers using the handler information table.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CLOSE, CREATE, DELETE, READ, SEEK, WRITE

Example:

The following example will open the serial port (channel 1) with the built-in handler.

```
OPEN        equ     0Eh             ;OPEN function code
            .
            .
            .
spmode      db      1               ;9600 baud
            db      00001101b       ;XON/XOFF,7 bits, even parity,
                                    ;1 stop, null strip disabled
            db      0Dh             ;terminate on CR
null        db      00h             ;the null string

            mov     ah,OPEN         ;OPEN function code
            mov     al,1            ;serial port channel
            push    cs              ;use default handler (ES:BX = null string)
            pop     es
            mov     bx,offset null
            push    cs              ;DS:DX = port config buffer
            pop     ds
            mov     dx,offset spmode
            int     1Ah             ;open the port
            or      al,al           ;error?
            jne     open_err        ;yes -- process the error
            .
            .
            .
```

PUT_CHAR

Display one character on the display and move the cursor one column to the right.

Call with:

AH=03h	PUT_CHAR function code.
AL	Character to display.

Returns:

Nothing.

Notes:

- When a character is written to the last column of a line in the display, the cursor will remain over that character. The display will be scrolled, if necessary, and the cursor moved to the first column of the next line before writing the next character.

Cautions:

- While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This time may be important to serial and bar code port handlers.

Related functions:

CURSOR, PUT_LINE

Example:

The following example will turn on the backlight, change the keyboard into alpha mode and display a prompt character.

```
PUT_CHAR        equ     03h             ;PUT_CHAR function code
ELON            equ     1Eh
ELOFF           equ     1Fh
ALPHMODE        equ     0Fh
NUMMODE         equ     0Eh
                .
                .
                .
                mov     ah,PUT_CHAR     ;PUT_CHAR function code
                mov     al,ELON         ;turn on backlight
                int     1Ah
                mov     al,ALPHMODE     ;alpha mode keyboard
                int     1Ah
                mov     al,">"          ;prompt character
                int     1Ah
                .
                .
                .
```

PUT_LINE

Display a character string on the display.

Call with:

AH=04h	PUT_LINE function code.
ES	Segment address of the write string.
BX	Offset address of the write string.

Returns:

Nothing.

Notes:

- The write string must be terminated with a null character (00h); the null will not be displayed. Any other ASCII character, including display control characters, may be embedded in the string.
- When a character is written to the last column of a line in the display, the cursor will remain over that character. The display will be scrolled, if necessary, and the cursor moved to the first column of the next line before writing the next character.

Cautions:

- While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This time may be important to serial and bar code port handlers.
- PUT_LINE will not check for wraparound of the write buffer's offset address.

Related functions:

CURSOR, PUT_CHAR

Example:

The following program will display the message "Hello world".

```
PUT_LINE        equ     04h             ;PUT_LINE function code
END_PROGRAM     equ     00h             ;END_PROGRAM function code
CMDMODE         equ     02h
code            segment
                assume  cs:code,ds:code
hiworld         proc    far
start:
                dw      prgmend-start
                dw      0006h           ;offset of internal entry point
                dw      0100h           ;version 1.00

                push    cs              ;set DS to CS
                pop     ds
                .
                .
                .
                mov     ah,PUT_LINE     ;PUT_LINE function code
                push    ds              ;set ES to DS
                pop     es
                mov     bx,offset msg   ;buffer offset to BX
                int     1Ah             ;write string to LCD
                .
                .
                .
                mov     ah,END_PROGRAM  ;enter command mode
                mov     al,CMDMODE
                int     1Ah
prgmend:
hiworld         endp
msg             db      "Hello world",0Dh,0Ah,00h
code            ends
                end
```

READ

Read data from an open channel.

Call with:

AH=12h	READ function code.
AL	Channel number to read.
CX	Number of bytes to read.
ES	Segment address of read buffer.
BX	Offset address of read buffer.

Returns:

AL=00h	Successful read.
=65h (101)	Illegal parameter.
=69h (105)	Channel not open.
=73h (115)	Short record detected.
=74h (116) *	Terminate character detected.
=75h (117)	End of data.
=76h (118)	Timeout. A timeout occurred before the read was completed.
=77h (119) †	Power switch pressed.
=C8h (200) †	Low battery.
=C9h (201) †	Receive buffer overflow.
=CAh (202) *	Parity error.
=CBh (203) *	Overrun error.
=CCh (204) *	Parity and overrun error.
=CDh (205) *	Framing error.
=CEh (206) *	Framing and parity error.
=CFh (207) *	Framing and overrun error.
=D0h (208) *	Framing, overrun and parity error.
CX	The number of bytes actually read.

* Can only occur when reading from channels 1 - 4. Whether these errors occur for a user-defined handler depends on the handler.

† Can only occur when reading from channels 0 - 4. Whether these errors occur for a user-defined handler depends on the handler.

Notes:

- When reading data from channels 1 - 4, READ will transfer control to the READ routine of the user-defined handler specified when the channel was opened. The same registers passed to the READ function will be passed to the user-defined handler READ routine with the following exceptions:

  DS Set to the segment address of the scratch area allocated by the handler.

  BP Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.

  DI Destroyed.

  Refer to the "User-Defined Handlers" chapter for details.

- Timeout, power switch and low battery will cause reads from channels 0 - 4 to be aborted, but will not interrupt reads from channels 5 - 15. Device I/O will be halted by these conditions, but file I/O will always be completed (unless the reset switch is pressed or the machine turns off automatically because of very low battery).
- When reading data from the keyboard (channel 0), no echoing to the display will occur. All keys pressed (except [SHIFT]) will be returned, unlike the GET_CHAR and GET_LINE functions. The number of bytes to read determines when READ will end, whether or not the [ENTER] key was pressed.
- When reading data from the built-in serial port handler (channel 1), if a terminate character was specified when the channel was opened, READ will stop if the terminate character is received even though the full read count has not been reached. The terminate character will be placed in the read buffer, but it will not be included in the returned read length, and error 74h will be reported.
- When reading data from a file (channels 5 - 15), data is read from the current file access pointer position. After the read is complete, the file access pointer is advanced by the size of the data read.
- Error 65h will occur if the number of bytes to read would cause the read buffer's offset address to wraparound.

Related functions:

CLOSE, CREATE, DELETE, OPEN, SEEK, WRITE

Cautions:

- This function may not be called from the POWERON routine of a handler.
- The number of bytes to read must not be greater than the actual read buffer length (although it can be less).

Example:

The following example will read from a channel.

```
READ                    equ         12h                 ;READ function code
                        .
                        .
                        .
;                       fread -- read a channel into a buffer
;
;                       call with:
;                                   al = channel #
;                                   cx = number of bytes to read
;                                   es = segment address of read buffer
;                                   bx = offset address of read buffer
;
fread                   proc        near
                        mov         ah,READ             ;READ function code
                        int         1Ah                 ;read the channel
                        or          al,al               ;set status for caller
                        ret
fread                   endp
                        .
                        .
                        .
```

REL_MEM

Release a scratch area obtained via GET_MEM.

Call with:

AH=0Ch	REL_MEM function code.
CX	Segment address of scratch area to release.

Returns:

AL=00h	Successful release.
=6Eh (110)	Access restricted. No free blocks available.
=72h (114)	Scratch area does not exist. Scratch area address does not correspond to a currently allocated scratch area.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

GET_MEM

Example:

The following example will free a the scratch area addressed by the current extra data segment (ES).

```
REL_MEM         equ     0Bh             ;REL_MEM function code
                .
                .
                .
                mov     cx,es           ;segment address of scratch area into cx
                mov     ah,REL_MEM      ;REL_MEM function code
                int     1Ah
                or      al,al           ;error?
                jne     rel_mem_err     ;yes -- handle it
                .
                .
                .
```

ROOM

Identify available room in a directory.

Call with:

AH=0Eh	ROOM function code.
AL	Directory number (0 - 4).

Returns:

AL=00h	Successful request.
=65h (101)	Illegal parameter. Invalid directory number.
=66h (102)	Directory does not exist.
BX	The available directory free space in paragraphs.
CX	Segment address of directory table.
DX	Total memory in directory in paragraphs, not including directory table.

Notes:

Related functions:

MEM_CONFIG

Example:

The following example will get the remaining space in main memory.

```
ROOM            equ     0Dh             ;ROOM function code
                .
                .
                .
                mov     al,0            ;directory 0
                mov     ah,ROOM         ;ROOM function code
                int     1Ah
                ;
                ;       BX=available free space in paragraphs
                ;       CX=segment address of directory table
                ;       DX=total memory in directory 0 in paragraphs
                ;
                .
                .
                .
```

SEEK

Move the file access pointer of an open file, or get the current pointer position.

Call with:

AH=15h	SEEK function code.
AL	Channel number.
BL=00h	Read the current file access pointer position.
=01h	Seek relative to the start of the file.
=02h	Set the file access pointer to EOD.
CX	High byte of 24-bit seek offset — CH ignored (for BL=00h or 01h).
DX	Low word of 24-bit seek offset (for BL=00h or 01h).

Returns:

AL=00h	Successful seek.
=65h (101)	Illegal parameter. This error will also occur for seeks on channels 0 - 4.
=69h (105)	Channel not open.
CX	High byte of the current 24-bit file access pointer (CH always set to zero).
DX	Low word of the current 24-bit file access pointer.

Notes:

- Seeks past EOD will generate error 65h.
- The 24-bit seek offset and file access pointer are relative to the start of the file. The first byte of the file has a seek offset and file access pointer position of 0.
- The file access pointer is set to 0 when the file is opened.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CLOSE, CREATE, DELETE, OPEN, READ, WRITE

...SEEK

Example:

The following example will seek to EOD to get the current true file size.

```
SEEK            equ     15h             ;SEEK function code
channel         db      5               ;channel to seek on
                .
                .
                .
                mov     ah,SEEK
                mov     al,channel      ;channel to seek on
                mov     bl,02h          ;SEEK to EOD
                int     1Ah             ;SEEK...
                or      al,al           ;SEEK error?
                jne     seek_err        ;yes, process it.
;
;               CX,DX now contain the exact number of bytes in the file,
;               regardless of padding to the nearest paragraph boundary.
;
                .
                .
                .
```

SET_INTR

Two system interrupts may put under program control — the power switch/system timeout and low battery interrupt. In addition, the power switch interrupt may be disabled or enabled.

Call with:

AH=0Ah	SET_INTR function code.
AL=00h	Define a power switch/system timeout interrupt routine.
=01h	Define a low battery interrupt routine.
=02h	Disable the power switch interrupt.
>02h	Enable the power switch interrupt.
BX	The data segment to be used for the interrupt routine. This value will be loaded into DS before the interrupt routine is activated (for AL=00h or 01h).
CX	Segment address of interrupt routine (for AL=00h or 01h).
DX	Offset address of interrupt routine (for AL=00h or 01h).

Returns:

Nothing.

Notes:

- An interrupt can be restored to its default behavior by calling SET_INTR with both CX=00h and DX=00h.
- When the power switch/timeout interrupt routine is called, the AL register will be set to 76h (118) if a timeout occurred, or 77h (119) if the power switch was pressed.

Cautions:

- The offset address specified in DX *must be non-zero* for the operating system to properly interpret the existence of user-defined interrupt routines.

Related functions:

None.

...SET_INTR

Example:

The following example will set up a power switch interrupt routine.

```
SET_INTR        equ     0Ah             ;SET_INTR function code
                .
                .
                .
                mov     ax,ds           ;put DS in BX
                mov     bx,ax
                mov     ax,cs           ;put CS in CX
                mov     cx,ax
                mov     dx,offset psint ;put routine offset into DX
                mov     ah,SET_INTR     ;SET_INTR function code
                mov     al,00h          ;set power switch interrupt routine
                int     1Ah             ;set it
                .
                .
                .
psint           proc    far             ;power switch interrupt routine
                .                       ;  interrupt routine for
                .                       ;  power switch goes here
                .
                ret                     ;return from interrupt
psint           endp
                .
                .
                .
```

TIMEOUT

Set the display backlight timeout and system timeout intervals. The timeouts may be as short as 1 second, as long as 1800 seconds, or disabled.

Call with:

AH=09h	TIMEOUT function code.
AL=00h	Set the display backlight timeout interval.
=01h	Set the system timeout interval.
BX	Number of seconds to set timeout to (1 - 1800). A value of 0 may be used to disable the timeout completely, in which case the backlight will never turn off or the system will never timeout (turn itself off).

Returns:

Nothing.

Notes:

- The initial value at cold start for both timeout intervals is 120 seconds.
- If AL is greater then 01h, no action is performed.
- If BX is greater than 1800 (0708h), no action is performed.
- Setting the display backlight timeout only sets the interval — it does not turn on the backlight. The backlight is turned on programmatically by writing the display control character 1Eh to the display.

Cautions:

- Leaving the backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.
- If the backlight is on and a new timeout interval is set, the backlight must be turned off (either programmatically or by timeout) before the new timeout interval will be in effect.

Related functions:

None.

...TIMEOUT

Example:

The following example will set the display backlight timeout interval to 3 minutes, and disable the system timeout.

```
TIMEOUT         equ     09h             ;TIMEOUT function code
                .
                .
                .
                mov     ah,TIMEOUT      ;TIMEOUT function code
                mov     al,00h          ;display backlight timeout
                mov     dx,180          ;3 minutes (180 seconds)
                int     1Ah             ;set it
                mov     al,01h          ;system timeout interval
                mov     dx,00h          ;disable timeout
                int     1Ah             ;set it
                .
                .
                .
```

TIME_DATE

Read or set the time and date of the real-time clock. The time and date is read into or set from a 17-byte fixed-length buffer. The format of the buffer is:

```
MM/DD/YY,hh:mm:ss
```

Symbol	Value	Range	Symbol	Value	Range
MM	Month	01-12	hh	Hour	00-23
DD	Date	01-31	mm	Minute	00-59
YY	Year	00-99	ss	Second	00-59

Call with:

AH=08h	TIME_DATE function code.
AL=00h	Set time and date.
=01h	Read time and date.
ES	Segment address of the time and date buffer.
BX	Offset address of the time and date buffer.

Returns:

Nothing.

Notes:

- When AL is greater than 01h, no action is performed.

Cautions:

- The validity of the time and date is not checked. If times and dates are set outside the above ranges, the clock wil be set to unpredictable values.

Related functions:

None.

Example:

The following program will read the current time and date and write it to the display. Since the code contains the read buffer for TIME_DATE, it will not work in ROM.

```
TIME_DATE       equ     08h             ;TIME_DATE function code
PUT_LINE        equ     04h             ;PUT_LINE function code
END_PROGRAM     equ     00h             ;END_PROGRAM function code
CMDMODE         equ     02h
TDBUFLEN        equ     17
code            segment
                assume  cs:code,ds:code
mem             proc    far
start:
                dw      prgmend-start
                dw      0006h           ;offset of internal entry point
                dw      0100h           ;version 1.00

                mov     ax,cs           ;set DS to CS
                mov     ds,ax
                mov     ah,TIME_DATE    ;TIME_DATE function code
                mov     al,01h          ;get date
                push    ds              ;set ES to DS
                pop     es
                mov     bx,offset buffer
                int     1Ah             ;get the time and date
                mov     ah,PUT_LINE     ;PUT_LINE function code
                int     1Ah             ;display it
                .
                .
                .
                mov     ah,END_PROGRAM  ;enter command mode
                mov     al,CMDMODE
                int     1Ah

buffer          db      TDBUFLEN dup (?)    ;must be in RAM
                db      0Dh,0Ah,00h
prgmend:
mem             endp
code            ends
                end
```

Write data to an open channel.

Call with:

AH=13h	WRITE function code.
AL	Channel number to write.
CX	Number of bytes to write.
ES	Segment address of write buffer.
BX	Offset address of write buffer.

Returns:

AL=00h	Successful write.
=65h (101)	Illegal parameter.
=69h (105)	Channel not open.
=6Dh (109)	Read-only access.
=70h (112)	No room to expand file.
=76h (118) *	Timeout. A timeout occurred before the write was completed.
=77h (119) *	Power switch pressed.
=C8h (200) *	Low battery.
=DAh (218) *	Lost connection while transmitting. The Clear to Send (CTS) control line was lowered.
CX	The number of bytes actually written.

Notes:

- When writing data to channels 1 - 4, WRITE will transfer control to the WRITE routine of the user-defined handler specified when the channel was opened. The same registers passed to the WRITE function will be passed to the user-defined handler WRITE routine with the following exceptions:

  DS — Set to the segment address of the scratch area allocated by the handler.

  BP — Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.

  DI — Destroyed.

  Refer to the "User-Defined Handlers" chapter for details.

* Can only occur when writing to channels 1 - 4. Whether these errors occur for a user-defined handler depends on the handler.

- Timeout, power switch and low battery will cause writes to channels 1 - 4 to be aborted, but will not interrupt writes to channels 5 - 15. Device I/O will be halted by these conditions, but file I/O will always be completed (unless the reset switch is pressed or the machine turns off automatically because of very low battery).
- When writing data to the built-in serial port handler (channel 1), if a terminate character was specified, the terminate character will be written after writing the data in the write buffer.
- When writing data to a file (channels 5 - 15), data is written from the current file access pointer position. After the write is complete, the file access pointer is advanced by the size of the buffer written.
- A write of 0 bytes to a data file will cause the EOD to be set equal to the current file access pointer. This has the effect of truncating the data in the file to the current pointer position, even though the file size will remain unchanged.
- Error 65h will occur if the number of bytes to write would cause the write buffer's offset address to wraparound.

Cautions:

- This function may not be called from the POWERON routine of a handler.
- The number of bytes to write must not be greater than the actual write buffer length (although it can be less).

Related functions:

CLOSE, CREATE, DELETE, OPEN, READ, SEEK

Example:

The following program will append one data file to another. If the destination file does not exist, it will be created. The program illustrates the use of the OPEN, CLOSE, CREATE, READ, WRITE and SEEK functions. Since the code contains the buffer address and length, it will not work in ROM.

```
GET_LINE            equ     02h             ;GET_LINE function code
PUT_CHAR            equ     03h             ;PUT_CHAR function code
PUT_LINE            equ     04h             ;PUT_LINE function code
GET_MEM             equ     0Bh             ;GET_MEM function code
REL_MEM             equ     0Ch             ;REL_MEM function code
OPEN                equ     0Fh             ;OPEN function code
CLOSE               equ     10h             ;CLOSE function code
CREATE              equ     11h             ;CREATE function code
READ                equ     12h             ;READ function code
WRITE               equ     13h             ;WRITE function code
SEEK                equ     15h             ;SEEK function code
DISPLAY_ERROR       equ     18h             ;DISPLAY_ERROR function code

BUFSIZ              equ     4               ;4 paragraphs

code                segment
                    assume  cs:code,ds:code
prog                proc    far
start:
                    dw      prgmend-start
```

```
                dw      0006h           ;offset of internal entry point
                dw      0100h           ;version 1.00

                push    cs              ;set DS to CS
                pop     ds

                ; initialize storage
                sub     ax,ax           ;clear AX
                mov     word ptr bufaddr,ax ;and clear buffer address

                ; allocate buffer memory
                mov     bx,BUFSIZ       ;size of buffer to allocate
                mov     al,0            ;not associated with a handler
                call    alloc           ;allocate memory
                call    errchk          ;call error check
                mov     word ptr bufaddr,cx ;store away buffer address
                mov     cl,4            ;turn buffer size from paragraphs to bytes
                shl     dx,cl
                mov     word ptr buflen,dx  ;store away buffer length

                push    cs              ;ES:BX address of "From" message
                pop     es
                mov     bx,offset FROM
                call    puts
                call    gets
                call    errchk
                mov     al,15           ;channel number for input file
                call    fopen           ;open the file
                call    errchk

                push    cs              ;ES:BX address of "To" message
                pop     es
                mov     bx,offset TO
                call    puts
                call    gets
                call    errchk

                mov     ah,SEEK         ;SEEK function code
                mov     al,15           ;infile channel
                mov     bl,02h          ;seek to EOD
                int     1Ah             ;seek to EOD to find file size
                                        ;(could also use FIND_FILE)
                or      cx,cx           ;is the file big (> 64k) (cx > 0)?
                jnz     crbig           ;yes -- use a default big size
                add     dx,15           ;to round up # of paragraphs
                jc      crbig           ;if we had a carry it is big file
                mov     cl,4            ;shift dx 4 bits right
                shr     dx,cl           ;to turn from bytes to paragraphs
                mov     cx,dx           ;initial size allocation to dx
                mov     dx,1            ;size increment
                jmp     crfile          ;and create the file
crbig:
                mov     cx,1000h        ;file is >= 10000h bytes (64K) long
                mov     dx,10h          ;use a large increment
crfile:
                mov     ax,word ptr bufaddr ;segment address of buffer
                mov     es,ax           ;to ES
                sub     bx,bx           ;offset address of buffer
                call    fcreate         ;create the file if it does not exist
                cmp     al,6Ch          ;file already exist?
```

```
                        je      nocrerr         ;ignore the error
                        or      al,al           ;set status for caller
                        call    errchk          ;check for any other create errors
nocrerr:
                        sub     bx,bx           ;offset address of buffer
                                                ;(clobbered by CREATE)
                        mov     al,14           ;channel number for outfile
                        call    fopen           ;open the file
                        jnz     errexit1        ;error -- close infile
                        mov     ah,SEEK         ;SEEK function code
                        mov     al,15           ;channel number for infile
                        mov     bl,01h          ;seek absolute
                        sub     cx,cx           ;to start of file (000000h)
                        sub     dx,dx
                        int     1Ah             ;seek to start of infile
                        mov     al,14           ;channel number for outfile
                        mov     bl,02h          ;seek to EOD
                        int     1Ah             ;seek to EOD to append to outfile
                        push    cs              ;ES:BX address of CRLF
                        pop     es
                        mov     bx,offset CRLF
                        call    puts

loop:
                        mov     ax,word ptr bufaddr ;ES:BX buffer address
                        mov     es,ax
                        sub     bx,bx           ;offset of buffer is 0
                        mov     cx,word ptr buflen  ;buffer size
                        mov     al,15           ;infile channel #
                        call    fread           ;read infile
                        cmp     al,73h          ;short record error?
                        je      norderr         ;not an error
                        cmp     al,75h          ;EOD?
                        je      closefiles      ;yes -- finish up
                        or      al,al           ;any other read error?
                        jnz     errexit         ;yes -- exit
norderr:
                        mov     al,14           ;outfile channel #
                        call    fwrite          ;write outfile (rest already set up)
                        jnz     errexit         ;error? -- exit
                        mov     ah,PUT_CHAR     ;PUT_CHAR function code
                        mov     al,"."
                        int     1Ah             ;display a "."
                        jmp     loop
closefiles:
                        sub     al,al           ;no error (al=0)
errexit:                                        ;close both infile and outfile
                        push    ax              ;save error code
                        mov     al,14           ;outfile channel #
                        call    fclose          ;close the file (ignore error code)
                        pop     ax              ;restore error code
errexit1:                                       ;only close infile
                        push    ax              ;save error code
                        mov     al,15           ;infile channel #
                        call    fclose          ;close the file (ignore error code)
                        pop     ax              ;restore error code
dsperr:
                        or      al,al           ;error?
                        jz      noerr           ;no -- don't display an error message
                        mov     ah,DISPLAY_ERROR    ;DISPLAY_ERROR function code
```

```
                int     1Ah             ;display it
noerr:
                mov     cx,word ptr bufaddr ;get segment address of buffer
                or      cx,cx           ;=0?
                jz      nofree          ;yes -- no buffer to free
                call    free            ;free the buffer
nofree:
                ret                     ;exit program
prog            endp

;               puts -- write a line to the LCD
;
;               call with:
;                       es = segment address of string
;                       bx = offset address of string
;
puts            proc    near
                mov     ah,PUT_LINE     ;PUT_LINE function code
                int     1Ah             ;display it
                ret
puts            endp

;               gets -- Read a line from the keyboard into buffer.
;                       Turn trailing CR into a NUL.
;                       Turn '.' into ':'
;
;               call with:
;                       nothing.
;
gets            proc    near
                mov     ax,word ptr bufaddr ;get segment address of buffer
                mov     es,ax           ;set ES to it
                sub     bx,bx           ;offset address to buffer
                mov     ax,word ptr buflen  ;size in bytes
                or      ah,ah           ;bigger than 256?
                jnz     okbuffer        ;no -- we will use the actual size
                mov     al,255
okbuffer:
                dec     al              ;leave room for the CR
                mov     ah,GET_LINE     ;GET_LINE function code
                int     1Ah             ;get string

                or      al,al           ;set status for caller
                jnz     getsret         ;error? -- return now
                ; save away registers
                push    ax
                push    cx
                push    dx
                push    di
                sub     dh,dh           ;clear high byte of dh
                add     dx,bx           ;address of last byte
                mov     di,dx           ;length to index register
                mov     byte ptr es:[di],00h;null out CR

                ; change '.' to ':'
                mov     di,bx           ;offset of string into offset register
dotloop:
                mov     ah,es:[di]      ;get the next character
                cmp     ah,"."          ;is it a dot?
                jne     nodot           ;no -- don't change it.
```

```
                        mov     ah,":"          ;replace it with a ":"
                        mov     es:[di],ah
nodot:
                        inc     di
                        or      ah,ah           ;is it a NUL (end of string)
                        jnz     dotloop
                        ; restore registers
                        pop     di
                        pop     dx
                        pop     cx
                        pop     ax
getsret:
                        or      al,al           ;set status for caller
                        ret
gets                    endp

errchk                  proc    near
                        or      al,al           ;return code 0?
                        jnz     err01           ;yes -- display an error
                        ret                     ;no -- just return
err01:
                        add     sp,2            ;pull off the near return address
                        jmp     dsperr          ;display the error & exit program
errchk                  endp

;                       alloc -- allocate a scratch area
;
;                       call with:
;                                 al = channel number for handler, 0 for others
;                                 bx = size of area in paragraphs
;
alloc                   proc    near
                        mov     ah,GET_MEM      ;GET_MEM function code
                        int     1Ah             ;allocate scratch area
                        or      al,al           ;set status for caller
                        ret
alloc                   endp

;                       free -- free a scratch area
;
;                       call with:
;                                 cx = segment address of scratch area
;
free                    proc    near
                        mov     ah,REL_MEM      ;REL_MEM function code
                        int     1Ah             ;release scratch area
                        or      al,al           ;set status for caller
                        ret
free                    endp

;                       fopen -- open a file
;
;                       call with:
;                                 al = channel #
;                                 es = segment address of file name buffer
;                                 bx = offset address of file name buffer
;                                 dx = offset address of parameter area
;                                      (built-in serial port only)
;
fopen                   proc    near
```

```
                        mov     ah,OPEN         ;OPEN function code
                        int     1Ah             ;open the file
                        or      al,al           ;set status for caller
                        ret
fopen                   endp

;                       fclose -- close an open file
;
;                       call with:
;                               al = channel #
;
fclose                  proc    near
                        mov     ah,CLOSE        ;CLOSE function code
                        int     1Ah             ;close the file
                        or      al,al           ;set status for caller
                        ret
fclose                  endp

;                       fcreate -- create a file
;
;                       call with:
;                               es = segment address of file name buffer
;                               bx = offset address of file name buffer
;                               cx = initial size in paragraphs
;                               dx = size increment in paragraphs
;
fcreate                 proc    near
                        mov     ah,CREATE       ;CREATE function code
                        int     1Ah             ;create the file
                        or      al,al           ;set status for caller
                        ret
fcreate                 endp

;                       fread -- read a channel into a buffer
;
;                       call with:
;                               al = channel #
;                               cx = number of bytes to read
;                               es = segment address of read buffer
;                               bx = offset address of read buffer
;
fread                   proc    near
                        mov     ah,READ         ;READ function code
                        int     1Ah             ;read the channel
                        or      al,al           ;set status for caller
                        ret
fread                   endp

;                       fwrite -- write a buffer into a channel
;
;                       call with:
;                               al = channel #
;                               cx = number of bytes to write
;                               es = segment address of write buffer
;                               bx = offset address of write buffer
;
fwrite                  proc    near
                        mov     ah,WRITE        ;WRITE function code
                        int     1Ah             ;write the buffer
                        or      al,al           ;set status for caller
```

```
                        ret
fwrite                  endp

FROM                    db              "From: ",0Fh,00h
TO                      db          0Dh,0Ah,"To:   ",0Fh,00h
CRLF                    db          0Dh,0Ah,00h
bufaddr                 dw          ?                   ;must be in RAM
buflen                  dw          ?                   ;must be in RAM

prgmend:
code                    ends
                        end
```

5

Hardware Control and Status Registers

Contents

5

Hardware Control and Status Registers

The HP-94 has control and status registers that allow a program to control the various hardware devices and determine their status. The control and status registers are in the CPU I/O space, so programs interact with them using the IN and OUT instructions. The details of these registers are discussed in the appropriate device chapters. The table below summarizes the I/O addresses for all the control and status registers.

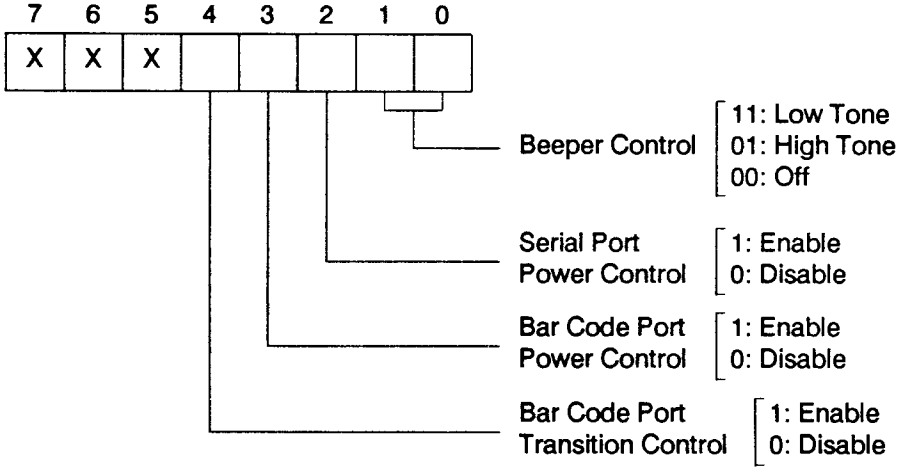
Table 5-1. I/O Addresses for Control and Status Registers

I/O Address	Register Name	Read/ Write
00h	Interrupt Control	W
00h	Interrupt Status	R
01h	Interrupt Clear	W
01h	End of Interrupt	R
02h	System Timer Data	R/W
03h	System Timer Control	W
04h	Bar Code Timer Data (lower 8 bits)	R/W
05h	Bar Code Timer Data (upper 4 bits)	R/W
06h	Bar Code Timer Control	W
07h	Bar Code Timer Value Capture	W
08h	Bar Code Timer Clear	W
0Ah	Baud Rate Clock Value	W
0Bh	Main Control	W
0Bh	Main Status	R
0Ch	Real-Time Clock Control	W
0Ch	Real-Time Clock Status/Data	R
0Eh	Keyboard Control	W
0Eh	Keyboard Status	R
10h	Serial Port Data	R/W
11h	Serial Port Control	W
11h	Serial Port Status	R
12h	Right LCD Driver Control	W
12h	Right LCD Driver Status	R
13h	Right LCD Driver Data	R/W
14h	Left LCD Driver Control	W
14h	Left LCD Driver Status	R
15h	Left LCD Driver Data	R/W
1Bh	Power Control	W

Two primary control registers are particularly important to programs: the main control register (0Bh) and the interrupt control register (00h),

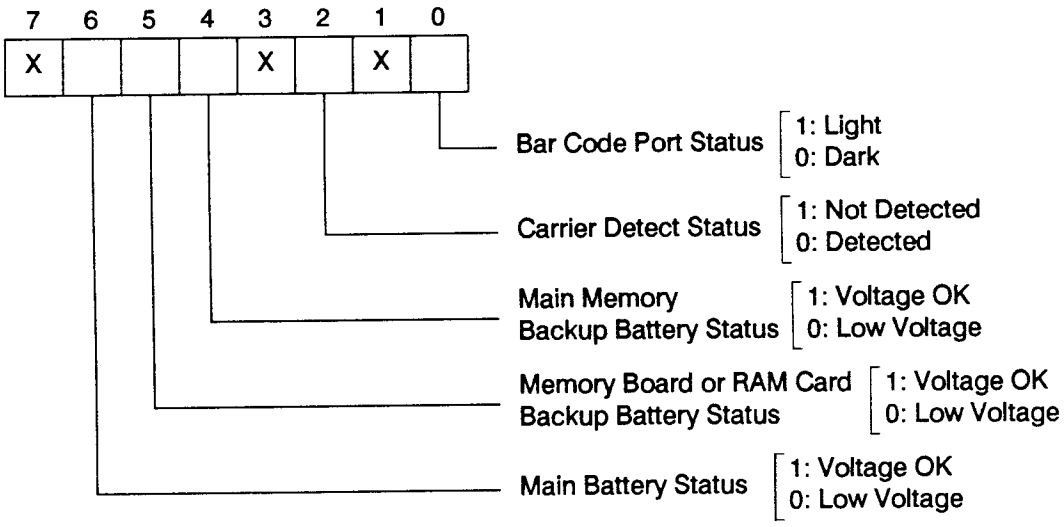
Main Control and Status Registers

The main control and status registers are at I/O address 0Bh. The uses of these registers to control specific hardware devices and determine their status are discussed in the appropriate device chapters. All the uses of these registers are summarized below.



X = don't care

Figure 5-1. Main Control Register (I/O Address 0Bh, Write)



X = ignore

Figure 5-2. Main Status Register (I/O Address 0Bh, Read)

Interrupt Control and Status Registers

The interrupt control and status registers are at I/O address 00h. The uses of these registers to enable specific hardware interrupts and determine which interrupts occurred are discussed in the "Interrupt Controller" chapter. All the uses of these registers are summarized below.

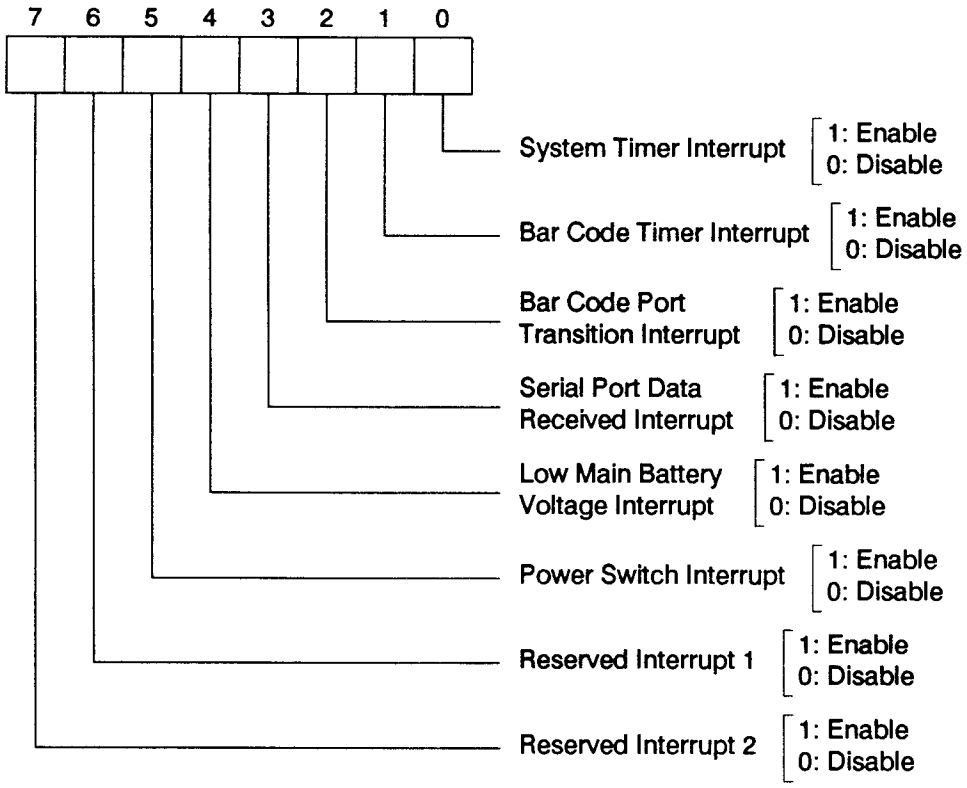


Figure 5-3. Interrupt Control Register (I/O Address 00h, Write)

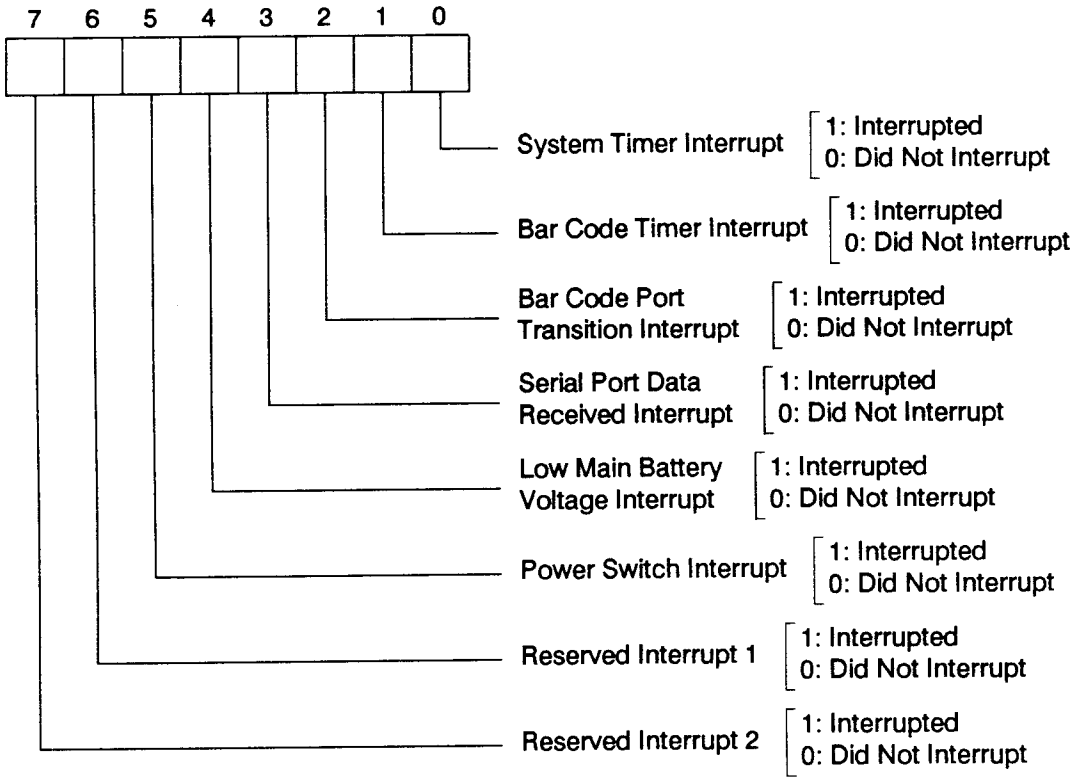


Figure 5-4. Interrupt Status Register (I/O Address 00h, Read)

Copies of Write-Only Control Registers

Control of the HP-94 I/O devices and interrupts is accomplished by using two primary control registers: the main control register and the interrupt control register. These are both write-only as far as controlling the devices and interrupts is concerned, and reading them back yields different results. Reading the main control register obtains other hardware status, and reading the interrupt control register indicates which interrupt occurred.

To allow the operating system and assembly language programs to know what status was set using these two registers, the operating system writes a copy of the register values to two locations in the operating system scratch space. When hardware or interrupt status is changed, the operating system uses the following procedure to ensure that hardware devices or interrupts unaffected by the change remain in their current state:

- Read the copy of the register being changed.
- Change the bits needed to cause the status to change.
- Write the updated value back to its original location.
- Output the updated value to the control register.

When a program uses the operating system functions and utility routines, these copies will be updated automatically. If a program changes the device or interrupt status independent of the operating system,

it is the program's responsibility to mimic the operating system action. That is, the program must make the change correctly while preserving the state of unaffected devices, and must update the copies of the control registers for use by the operating system and other programs.

The status of these registers at cold and warm start is shown below. Refer to appendix L for information about the utility subroutines for reading and saving copies of the control registers.

Table 5-2. Copies of Primary Control Registers

Control Register Name	I/O Address	Initial Value	Meaning of Initial Value	Utility Subroutines
Main Control	0Bh	00h	Beeper off, serial port power off, bar code port power off, bar code port transitions disabled	READCTRL.ASM SETCTRL.ASM
Interrupt Control	00h	31h	System timer, low battery, and power switch interrupts enabled	READINTR.ASM SETINTR.ASM

6

CPU

6

CPU

The HP-94 CPU is the NEC μPD70108 (V20) microprocessor. This is a CMOS microprocessor that is compatible with the Intel 8088 and provides a standby mode for reduced power consumption. Programs written for the 8088 can be run on the V20 with no modifications.

The V20 provides a superset of the 8088 instruction set. Some 8088 instructions have been enhanced, and new instructions have been added. All the changes are described in the CPU data sheet in the "Hardware Specifications". The enhancements and additions are only available if NEC assembly language development tools are used. Contact NEC for information on these if using the V20 features is important to your applications.

The HP-94 CPU runs at an operating frequency of 3.6864 MHz, which is 0.27 μs/clock cycle. Note, however, that the V20 instruction timing is different than the 8088 instruction timing. The V20 timing should be used whenever determining the number of clock cycles for specific operations. The instruction timing is shown in the CPU data sheet using NEC mnemonics. These are similar but not identical to 8088 mnemonics, as shown in the next table.

Table 6-1. Intel 8088 and NEC V20 Instruction Mnemonics

Intel 8088	NEC V20	Intel 8088	NEC V20	Intel 8088	NEC V20	Intel 8088	NEC V20
AAA	ADJBA	JA	BH	JZ	BE,BZ	REPE	REPE
AAD	CVTDB	JAE	BNC,BNL	LAHF	MOV	REPZ	REPZ
AAM	CVTBD	JB	BC,BL	LDS	MOV	REPNE	REPNE
AAS	ADJBS	JBE	BNH	LEA	LDEA	REPNZ	REPNZ
ADC	ADDC	JC	BC,BL	LES	MOV	RET	RET
ADD	ADD	JCXZ	BCWZ	LOCK	BUSLOCK	ROL	ROL
AND	AND	JE	BE,BZ	LODS	LDM	ROR	ROR
CALL	CALL	JG	BGT	LODSB	LDM	SAHF	MOV
CBW	CVTBW	JGE	BGE	LODSW	LDM	SAL	SHL
CLC	CLR1	JL	BLT	LOOP	DBNZ	SAR	SHRA
CLD	CLR1	JLE	BLE	LOOPE	DBNZE	SBB	SUBC
CLI	DI	JMP	BR	LOOPNE	DBNZNE	SCAS	CMPM
CMC	NOT1	JNA	BNH	LOOPNZ	DBNZNE	SCASB	CMPM
CMP	CMP	JNAE	BC,BL	LOOPZ	DBNZE	SCASW	CMPM
CMPS	CMPBK	JNB	BNC,BNL	MOV	MOV	SHL	SHL
CMPSB	CMPBK	JNBE	BH	MOVS	MOVBK	SHR	SHR
CMPSW	CMPBK	JNC	BNC,BNL	MOVSB	MOVBK	STC	SET1
CWD	CVTWL	JNE	BNE,BNZ	MOVSW	MOVBK	STD	SET1
DAA	ADJ4A	JNG	BLE	MUL	MULU	STI	EI
DAS	ADJ4S	JNGE	BLT	NEG	NEG	STOS	STM
DEC	DEC	JNL	BGE	NOP	NOP	STOSB	STM
DIV	DIVU	JNLE	BGT	NOT	NOT	STOSW	STM
ESC	FPO1	JNO	BNV	OR	OR	SUB	SUB
HLT	HALT	JNP	BPO	OUT	OUT	TEST	TEST
IDIV	DIV	JNS	BP	POP	POP	WAIT	POLL
IMUL	MUL	JNZ	BNE,BNZ	POPF	POP	XCHG	XCH
IN	IN	JO	BV	PUSH	PUSH	XLAT	TRANS
INC	INC	JP	BPE	PUSHF	PUSH	XOR	XOR
INT	BRK	JPE	BPE	RCL	ROLC		
INTO	BRKV	JPO	BPO	RCR	RORC		
IRET	RETI	JS	BN	REP	REP		

7

Interrupt Controller

Contents

7

Interrupt Controller

The HP-94 interrupt controller receives interrupt requests from eight different HP-94 hardware devices. It prioritizes these interrupts, and informs the CPU of the highest priority interrupt. The CPU then locates the interrupt vector for that interrupt and transfers control to the interrupt service routine. The hardware interrupts and their priority are shown below:

Table 7-1. HP-94 Hardware Interrupts

Interrupt Type	Interrupt Name
50h	System Timer
51h	Bar Code Timer
52h	Bar Code Port Transition
53h	Serial Port Data Received
54h	Low Main Battery Voltage
55h	Power Switch
56h	Reserved Interrupt 1
57h	Reserved Interrupt 2

Highest

↓

Interrupt Priority

↓

Lowest

Information about the behavior of interrupt service routines for the different hardware devices are in the appropriate device chapters.

At both cold and warm start, the system timer, serial port data received, low main battery voltage, and power switch interrupt vectors all point to their operating system interrupt service routines. They are all enabled except for the serial port data received interrupt. The other hardware interrupt vectors point to a dummy interrupt service routine which clears the interrupt, reads the end of interrupt register, and returns (with an IRET). Reserved interrupts 1 and 2 are for future use.

Procedure for Using a Hardware Interrupt

There are four control registers available for controlling interrupt behavior:

- Interrupt Control Register
  This is used to enable or disable any of the hardware interrupts.
- Interrupt Status Register
  This indicates which hardware devices have issued interrupt requests. The interrupt status register will indicate that an interrupt request occurred *even if the interrupt was disabled.* This is useful for polling device status.

- Interrupt Clear Register
  Once a hardware interrupt has occurred, another interrupt of the same type will not be processed by the interrupt controller until that interrupt has been cleared.
- End of Interrupt Register
  This is read at the end of an interrupt service routine to allow the interrupt controller to generate new interrupts of any type.

There are several things that must be done to use a hardware interrupt. Some must be done when the interrupt is initialized, and others during an interrupt service routine. These are summarized below:

Table 7-2. Using Hardware Interrupts

Action	Control or Status Register Used	Required During Initialization	Required In Service Routine
Disable Interrupt	Interrupt Control	No *	No
Take Over Interrupt Vector	—	Yes	No
Enable Interrupt	Interrupt Control	Yes	No
Set CPU Interrupt Flag (STI)	—	Yes	No †
Verify Interrupt Source	Interrupt Status	No	No *
Clear Interrupt	Interrupt Clear	No *	Yes
Read End of Interrupt Register	End of Interrupt	No *	Yes
Return from Interrupt (IRET)	—	No	Yes

* Not required, but can be done as defensive programming. For example, it is unlikely when enabling an interrupt that a previous interrupt request of the same type is present, requiring that the interrupt be cleared before it can occur again. The same reasoning can be applied to the other items that reference this footnote.

† Set automatically by IRET.

When taking over an interrupt, the interrupt vector location is the two words starting at address *T* * 4, where *T* is the interrupt type. This is at addresses 00140h-0015Ch for the hardware interrupts. The instruction pointer (IP) offset of the interrupt service routine should be stored at the first word, and the code segment (CS) address of the routine should be stored at the second word.

The existing interrupt vector should be saved when the interrupt is taken over, then restored when the program gives up the interrupt.

If the interrupt service routine is in a user-defined handler, the program should save the segment address of the handler scratch area in the handler information table. See the "User-Defined Handlers" chapter for details.

Software interrupt 1Ah for calling operating system functions is discussed in the "Operating System Functions" chapter, and software interrupt 1Ch for the background timer is discussed in the "Timers" chapter.

Interrupt Control and Status Registers

The interrupt control and status registers are shown below. A copy of the main interrupt control register is maintained in the operating system scratch space for reference. Refer to the "Hardware Control and Status Registers" chapter for further information.

Table 7-3. Interrupt Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	0-7	W
Interrupt Status	00h	0-7	R
Interrupt Clear	01h	0-7	W
End of Interrupt	01h	None	R

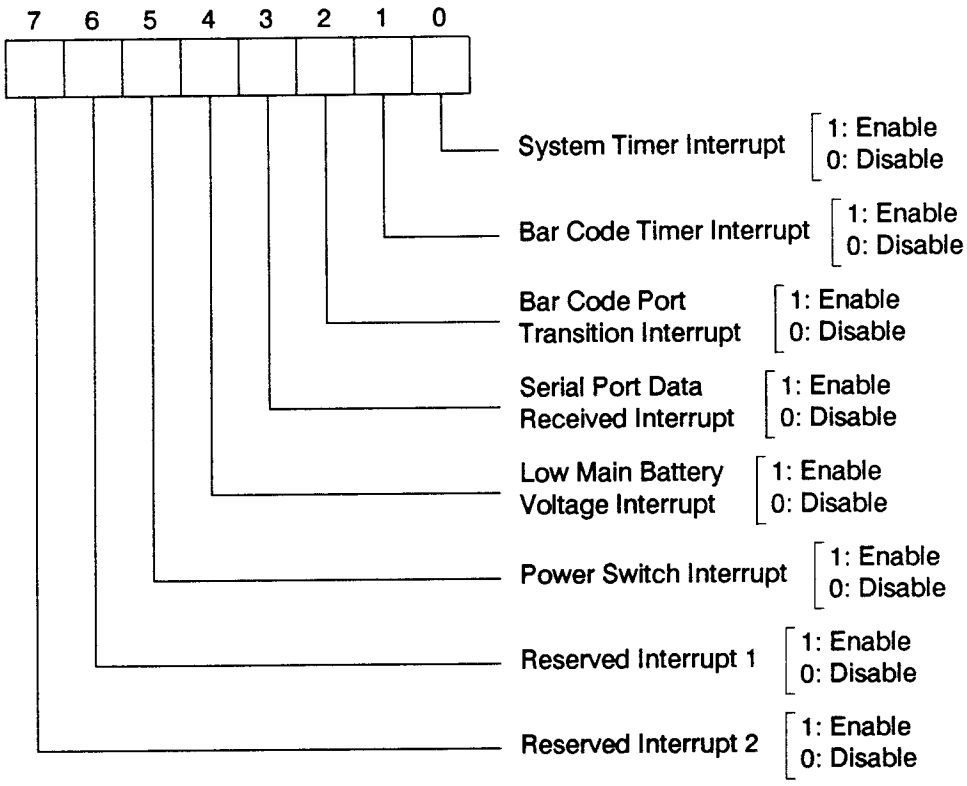


Figure 7-1. Interrupt Control Register (I/O Address 00h, Write)

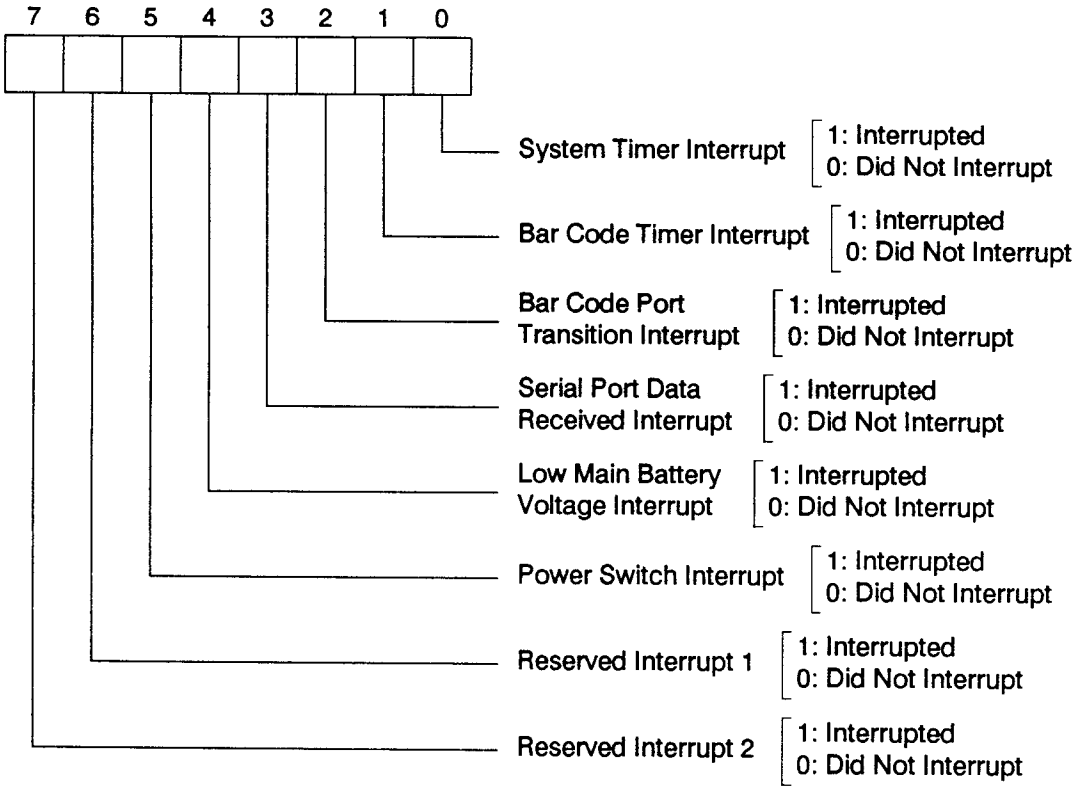


Figure 7-2. Interrupt Status Register (I/O Address 00h, Read)

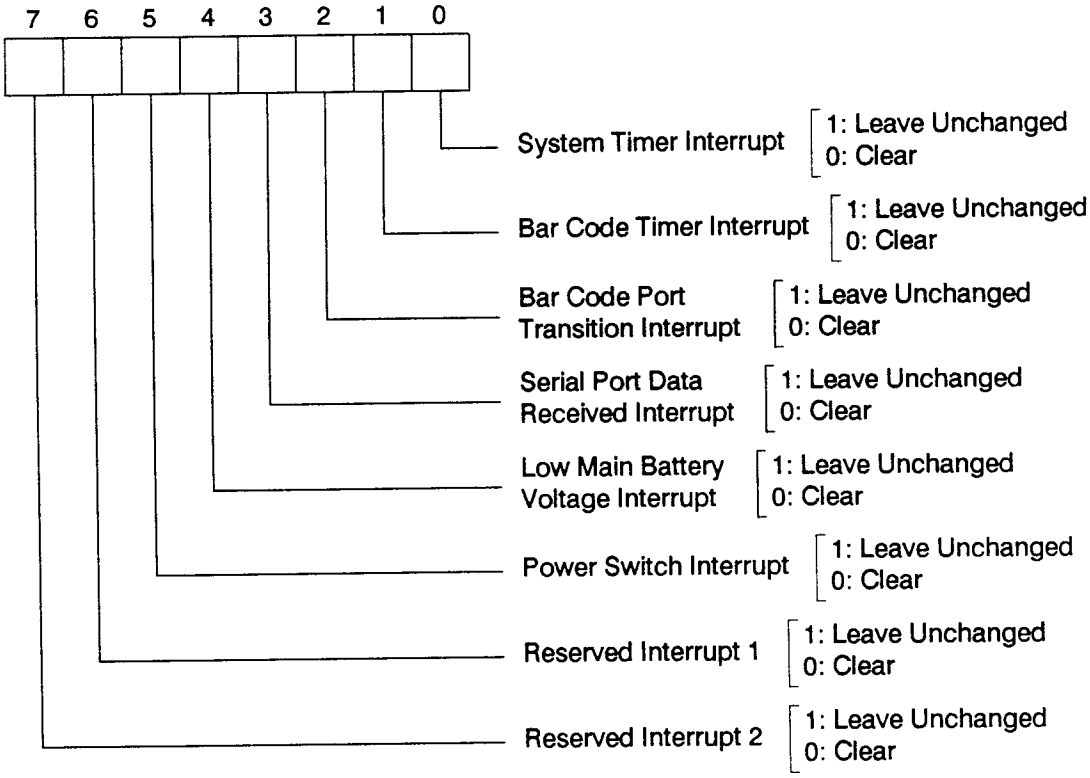


Figure 7-3. Interrupt Clear Register (I/O Address 01h, Write)

7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X

X = ignore

Figure 7-4. End of Interrupt Register (I/O Address 01h, Read)

When the Operating System Disables Interrupts

The operating system disables interrupts by clearing the CPU interrupt flag (CLI) at two times that may be important to time-critical interrupt service routines:

- While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This may be important for serial and bar code port handlers.
- While checking to see if the beeper needs to be turned off, interrupts are disabled for ~50 μs. This may be important for bar code port handlers.

Operating System Functions

The interrupt software implements the following operating system functions:

Table 7-4. Interrupt-Related Operating System Functions

Function Name	Function Code
TIMEOUT	09h
SET_INTR	0Ah

8

Keyboard

Contents

Chapter 8 Keyboard

8

Keyboard

The HP-94 keyboard has 34 keys, arranged as shown below.

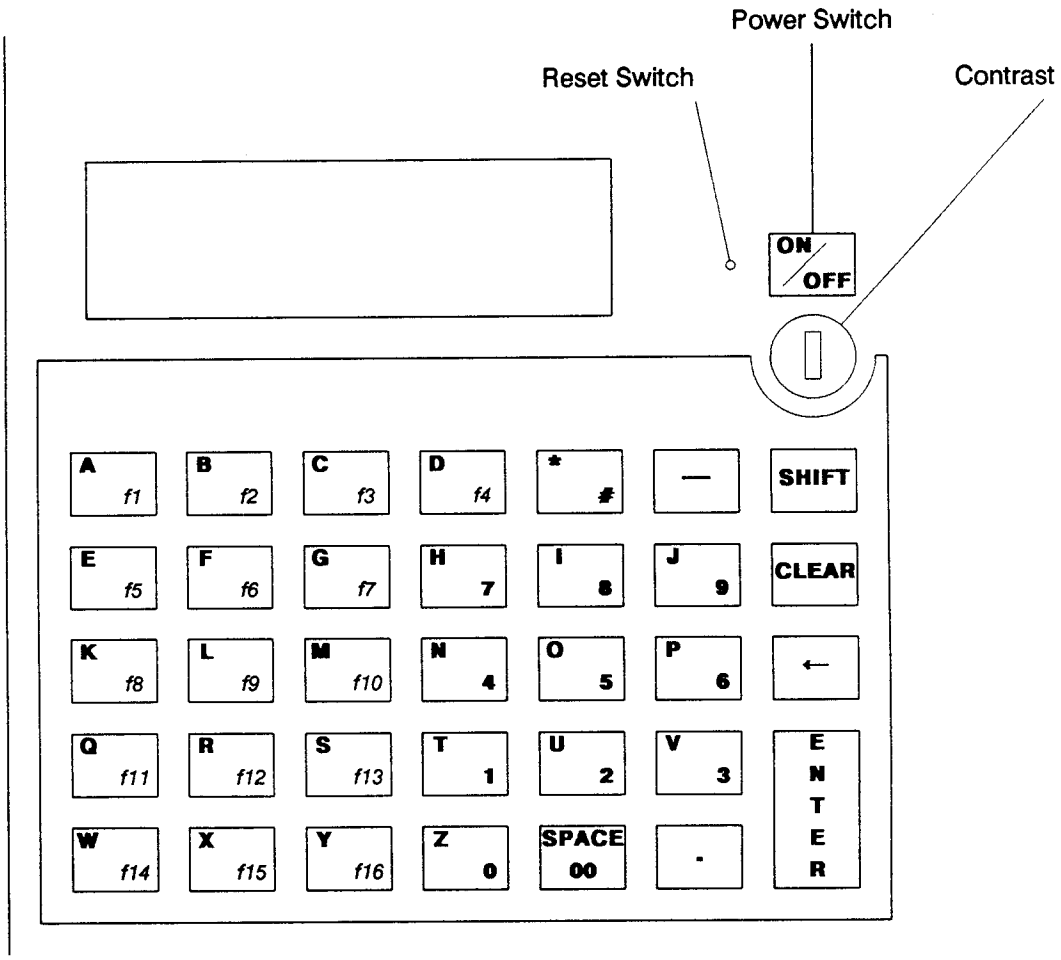


Figure 8-1. HP-94 Keyboard

Keyboard Shift Status

The symbols on the upper left corner of each key are in orange and can be entered when the keyboard is shifted. The symbols on the lower right corner of each key are in white and can be entered when the keyboard is unshifted. Keys with only one centered symbol are in white and can be entered whether the keyboard is shifted or not.

The keys labelled *f1* through *f16* are the user-defined keys, and have no predefined action associated with them. When the keyboard is unshifted, they return ASCII 80h-8Fh which corresponds to the first 16 user-defined characters (see the "Display" chapter for details).

The [SHIFT] key toggles between unshifted and shifted keys. The keyboard shift status is indicated by the shape of the cursor. An underscore cursor indicates unshifted (white keys), and a block cursor indicates shifted (orange keys).

Display Backlight Control

The [SHIFT] key controls the display backlight. If the [SHIFT] key is held down for one second, the display backlight will be turned on (or off if it was already on). When the backlight is toggled by holding down [SHIFT] for one second, the keyboard status and cursor type will be unchanged.

The backlight will turn off automatically after two minutes (120 seconds). This timeout can be set under program control between 0 (never turn off) and 1800 seconds. The display backlight can be turned on or off from a program by writing the appropriate display control character to the display: 1Eh turns on the backlight, and 1Fh turns off the backlight. The keyboard control register has a bit to turn on and off the backlight.

CAUTION Leaving the display backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.

Key Buffer

There is an eight-character key buffer where the ASCII equivalents of each scanned key (not the keycodes) are placed. A short, low tone beep will be issued when a key is placed in the key buffer (note that this beep cannot be disabled). A long, high tone beep will be issued when a key is pressed after the buffer is full — the key will be discarded. When the [SHIFT] key is pressed, it is processed for changing keyboard shift status and the backlight control, but is not placed in the key buffer.

Waiting for a Key

While waiting for a key to be pressed, the keyboard software puts the CPU into its standby mode to save power, and monitors the system timeout. The timeout is restarted every time a key is pressed. When the timeout expires, the default behavior is to turn the machine off. If a program has defined a power switch/timeout interrupt routine using the SET_INTR function (0Ah), that routine will be executed with a FAR CALL when the timeout expires. This will only occur in a running program, not in command mode.

Keyboard Scanning

The keyboard is scanned by the operating system software every 5 ms. Keys are debounced for 25 ms. When a key has been held down for 675 ms, it begins to repeat every 115 ms until it is released.

The keyboard control register has a bit for each column to be scanned. The keyboard is scanned by clearing the bit corresponding to the column to be scanned, and reading the keyboard status register to see which row(s) have a key down. If a key is down, the bit corresponding to that row will be set. The correspondence between the keyboard and the bits in the keyboard control and status registers are shown below.

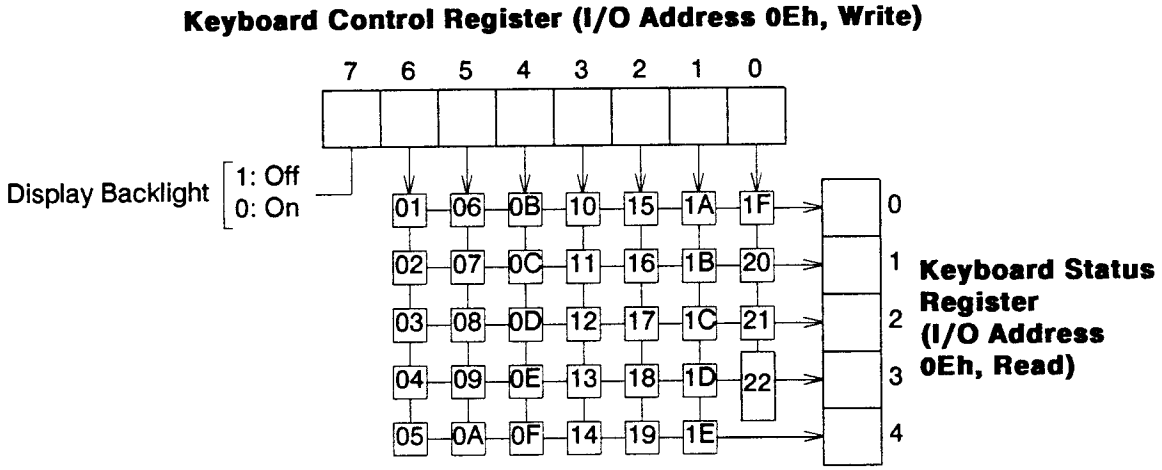


Figure 8-2. HP-94 Keycodes

If multiple columns are selected for scanning, the program will not be able to distinguish which key was pressed. It will only be able to identify that a key in a particular row was held down.

The operating system scans the keyboard columns from right to left, and checks the rows from top to bottom. The first key found down in that scanning sequence will be reported as a keycode (shown above in hex). Other keys to the left or below the first key found will be ignored (the [CLEAR] and [ENTER] sequence to enter command mode is scanned as a special case). The keycode will be translated into an ASCII character according to the keyboard shift status and the following keyboard map.

Table 8-1. ASCII Characters and Keycodes for Each Key

Shifted Key (orange)	Shifted Character	Unshifted Key (white)	Unshifted Character	Keycode
A	A (41h)	(unmarked)	user-defined (80h)	01h
B	B (42h)	(unmarked)	user-defined (81h)	06h
C	C (43h)	(unmarked)	user-defined (82h)	0Bh
D	D (44h)	(unmarked)	user-defined (83h)	10h
E	E (45h)	(unmarked)	user-defined (84h)	02h
F	F (46h)	(unmarked)	user-defined (85h)	07h
G	G (47h)	(unmarked)	user-defined (86h)	0Ch
H	H (48h)	7	7 (37h)	11h
I	I (49h)	8	8 (38h)	16h
J	J (4Ah)	9	9 (39h)	1Bh
K	K (4Bh)	(unmarked)	user-defined (87h)	03h
L	L (4Ch)	(unmarked)	user-defined (88h)	08h
M	M (4Dh)	(unmarked)	user-defined (89h)	0Dh
N	N (4Eh)	4	4 (34h)	12h
O	O (4Fh)	5	5 (35h)	17h
P	P (50h)	6	6 (36h)	1Ch
Q	Q (51h)	(unmarked)	user-defined (8Ah)	04h
R	R (52h)	(unmarked)	user-defined (8Bh)	09h
S	S (53h)	(unmarked)	user-defined (8Ch)	0Eh
T	T (54h)	1	1 (31h)	13h
U	U (55h)	2	2 (32h)	18h
V	V (56h)	3	3 (33h)	1Dh
W	W (57h)	(unmarked)	user-defined (8Dh)	05h
X	X (58h)	(unmarked)	user-defined (8Eh)	0Ah
Y	Y (59h)	(unmarked)	user-defined (8Fh)	0Fh
Z	Z (5Ah)	0	0 (30h)	14h
*	* (2Ah)	#	# (23h)	15h
SPACE	(space) (20h)	00	00 (30h 30h)	19h
—	— (2Dh)	—	— (2Dh)	1Ah
.	. (2Eh)	.	. (2Eh)	1Eh
SHIFT	(none)	SHIFT	(none)	1Fh
CLEAR	(CAN) (18h)	CLEAR	(CAN) (18h)	20h
←	(DEL) (7Fh)	←	(DEL) (7Fh)	21h
ENTER	(CR) (0Dh)	ENTER	(CR) (0Dh)	22h

Refer to the appendixes for a utility routine that scans the keyboard and returns the keycode of the first key found down.

Keyboard Scanning at Turn On

When the machine turns on, the operating system checks the keyboard after performing the first three memory integrity checks (system ROM checksum, reserved scratch space read/write, and valid RAM configuration). If any keys are down other than [CLEAR] and [ENTER], the machine will turn back off immediately. This is to prevent accidental turn on (while in a full briefcase, for example).

Keyboard Control and Status Registers

The keyboard control and status registers are summarized below.

Table 8-2. Keyboard Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Keyboard Control	0Eh	0-7	W
Keyboard Status	0Eh	0-4	R

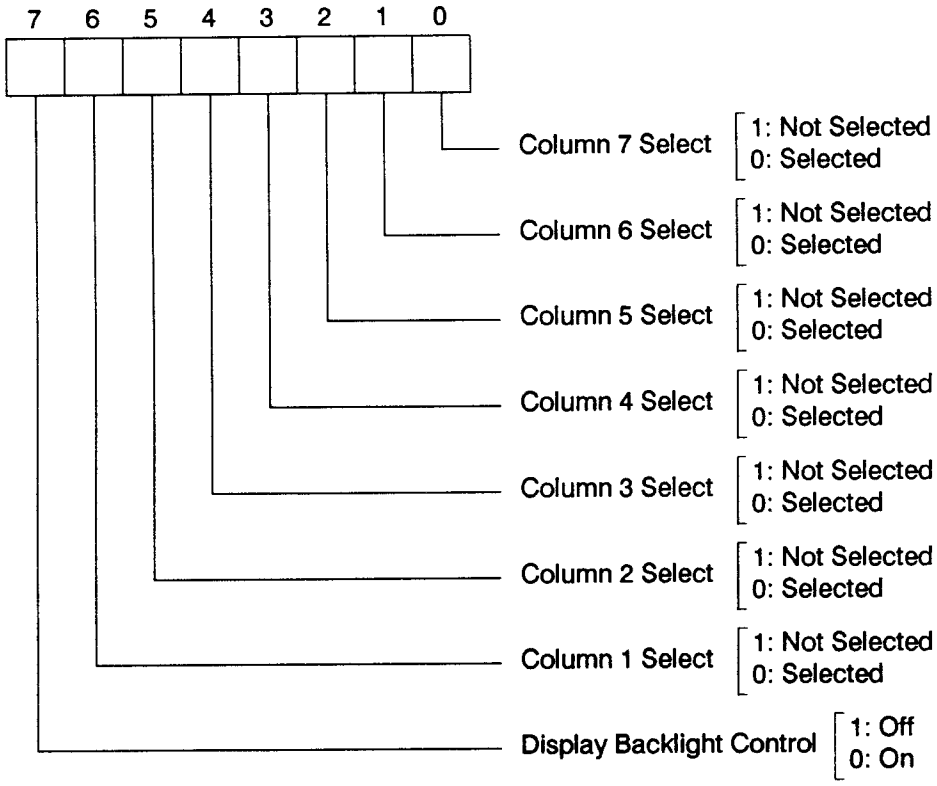
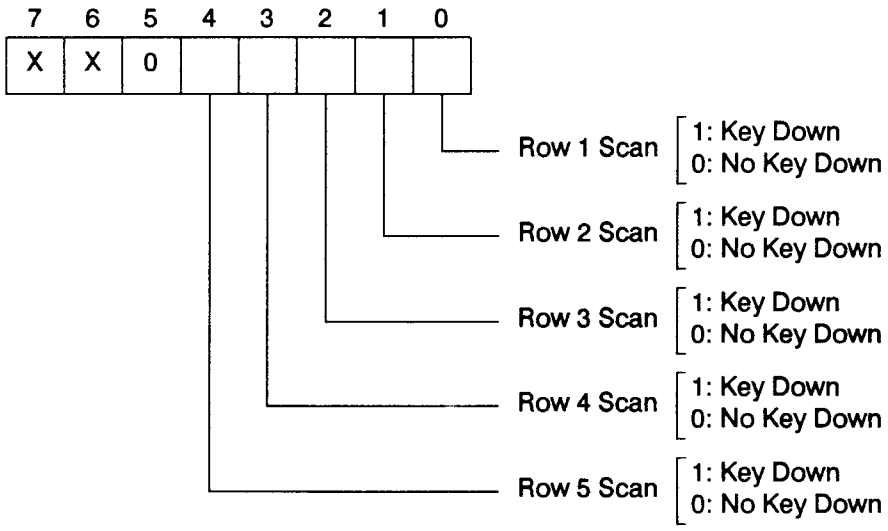


Figure 8-3. Keyboard Control Register (I/O Address 0Eh, Write)



X = ignore

Figure 8-4. Keyboard Status Register (I/O Address 0Eh, Read)

Operating System Functions

The keyboard software implements the following operating system functions:

Table 8-3. Keyboard-Related Operating System Functions

Function Name	Function Code
GET_CHAR	01h
GET_LINE	02h
PUT_CHAR	03h
PUT_LINE	04h
BUFFER_STATUS	06h
READ	12h

9

Display

Contents

Chapter 9 Display

9

Display

The HP-94 has a liquid crystal display (LCD) with an electroluminescent backlight. The display is a continuous dot-matrix of 120 columns and 32 rows, yielding 4 lines of 20 characters each, where each character is in a 6 × 8 character cell. The built-in Roman-8 character set places characters in a 5 × 8 cell, leaving the right column of the 6 × 8 cell blank. It uses the eighth dot for descenders only. The orientation of a character cell is shown below. The filled-in boxes are the dot positions used by the built-in character set.

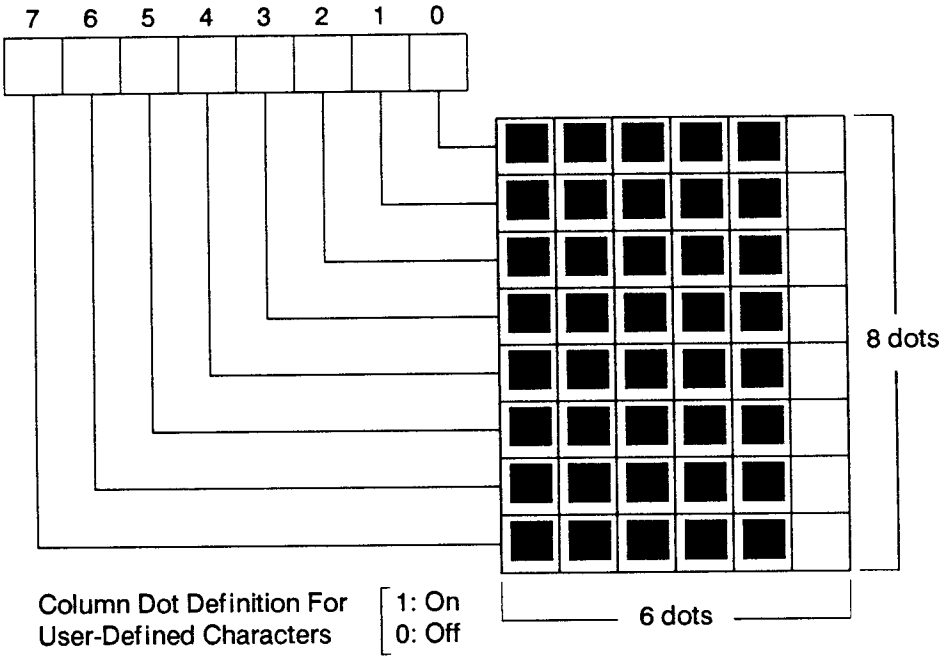


Figure 9-1. 6 × 8 Character Cell

All characters are mapped upside-down. The upper dot of a column of a character is bit 0 of the byte containing the bit pattern for that column. There are 6 bytes per character, one per column from left to right.

Display Backlight Control

The [SHIFT] key controls the display backlight. If the [SHIFT] key is held down for one second, the display backlight will be turned on (or off if it was already on). When the backlight is toggled by holding down [SHIFT] for one second, the keyboard status and cursor type will be unchanged.

The backlight will turn off automatically after two minutes (120 seconds). This timeout can be set under program control between 0 (never turn off) and 1800 seconds. The display backlight can be turned on or off from a program by writing the appropriate display control character to the display: 1Eh turns on the backlight, and 1Fh turns off the backlight. The keyboard control register has a bit to turn on and off the backlight.

CAUTION Leaving the display backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.

LCD Controllers

There are three LCD controllers. The row driver is a Hitachi HD61103A. It is not accessible to software — the rows are driven automatically by the hardware.

The column driver is a Hitachi HD61102A. Since the column driver can only support 64 columns, two are used. The left half driver controls columns 0-63 (counting from the left), and the right half driver controls column 64-119. Columns 120-127 are ignored. The details of the column driver hardware, operation, and usage are described in the Hitachi HD61102A data sheet in the "Hardware Specifications".

Writing Dots to the Display

Programs writing directly to the display hardware can write an 8-dot pattern to any column in the LCD. As with characters, the dots in the column being written are represented upside-down in the byte containing that dot pattern. A program cannot write individual dots to the display — the display control registers only allow writing columns of data. (Since a program can read individual columns of data, it could read a column, change a dot, and write the column back. This would have the effect of writing an individual dot.)

Display Control and Status Registers

The display control and status registers are shown below.

Table 9-1. Display Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Keyboard Control	0Eh	7	W
Right LCD Driver Control	12h	0-7 *	W
Right LCD Driver Status	12h	0-7 *	R
Right LCD Driver Data	13h	0-7	R/W
Left LCD Driver Control	14h	0-7 *	W
Left LCD Driver Status	14h	0-7 *	R
Left LCD Driver Data	15h	0-7	R/W

* For the meaning of the bits in these registers, refer to the Hitachi HD61102A data sheet in the "Hardware Specifications".

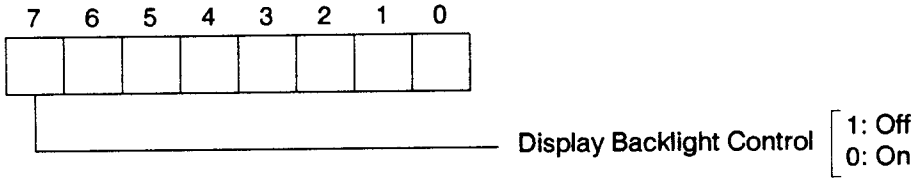


Figure 9-2. Keyboard Control Register (I/O Address 0Eh, Write)

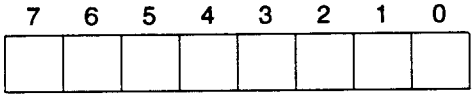


Figure 9-3. Right LCD Driver Data Register (I/O Address 13h, Read/Write)

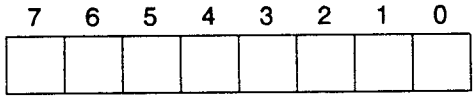


Figure 9-4. Left LCD Driver Data Register (I/O Address 15h, Read/Write)

Writing Characters to the Display

The display software performs the generation of characters from the built-in Roman-8 character set. The first half of the character set (characters 00h-7Fh) consists of standard U.S. ASCII characters. The second half (80h-FFh) contains special characters, including those used by other languages. The display software also displays user-defined characters in the range 80h-9Fh. These will be discussed shortly.

Cursor shape, status, movement, and blinking is also controlled by the display software. The cursor shape is a block to represent shifted keyboard status and an underline to represent unshifted status. The cursor can be either on or off. When on, it is blinked every 500 ms (0.5 s).

Power switch and low battery interrupts can occur while writing data to the display using operating system functions. The system timeout does not occur when writing to the display (channel 0).

The display software processes display control codes for the following actions:

Table 9-2. Display Control Characters

Hex Value	Meaning
01h (SOH)	Turn on cursor.
02h (STX)	Turn off cursor.
06h (ACK)	High tone beep for 0.5 second.
07h (BEL)	Low tone beep for 0.5 second.
08h (BS)	Move cursor left one column. When the cursor reaches the left end of the line, it will back up to the right end of the previous line. When the cursor reaches the top left corner, backspace will have no effect.
0Ah (LF)	Move cursor down one line. If the cursor is on the bottom line, the display contents will scroll up one line.
0Bh (VT)	Clear every character from the cursor position to the end of the current line. The cursor position will be unchanged.
0Ch (FF)	Move cursor to upper left corner and clear the display.
0Dh (CR)	Move cursor to left end of current line.
0Eh (SO)	Change keyboard to numeric mode (underline cursor).
0Fh (SI)	Change keyboard to alpha mode (block cursor).
1Eh (RS)	Turn on display backlight.
1Fh (US)	Turn off display backlight.

Control codes not listed in this table are ignored — that is, no character is displayed for those codes.

NOTE While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This time may be important to serial and bar code port handlers.

Operating System Functions

The display software implements the following operating system functions:

Table 9-3. Display-Related Operating System Functions

Function Name	Function Code
PUT_CHAR	03h
PUT_LINE	04h
CURSOR	05h
WRITE	13h
DISPLAY_ERROR	18h

User-Defined Characters

The HP-94 allows the font for 32 characters to be redefined: character codes 80h-9Fh, the control codes for the upper 128 characters of the built-in Roman-8 character set. The operating system will use these redefined characters only when a program is running — they will not be used in command mode. When a program is executed (either with the S (start) command or by autostarting), the operating system searches for a type A font definition file named SYFT. If this file is found, and is the correct type, then the dot pattern for characters 80h-9Fh will be taken from it. If SYFT does not exist, characters in that range will be displayed as blanks.

Character mapping will occur whenever characters 80h-9Fh are displayed on the LCD using the PUT_CHAR and PUT_LINE functions, or the WRITE function for channel 0 (functions 03h, 04h, and 13h). PUT_CHAR and PUT_LINE are used by the BASIC I/O keywords PRINT, PRINT USING, PRINT #, PRINT #...USING, and PUT #.

Structure of SYFT Font Definition File

The SYFT font file must contain definitions for 32 characters. If it does not, some characters will be constructed from the contents of the file immediately following SYFT (higher in memory). While this will not have any harmful side effects, it is unlikely to provide useful characters. Unlike type A program files, SYFT does not require a program header.

There are six bytes per character in SYFT, one for each of the six columns of data to be defined in the character's 6 × 8 character cell. All six bytes can be used for dot information. The built-in Roman-8 character set leaves the rightmost column of each character blank to provide intercharacter spacing, but that is not required.

All characters are mapped upside-down. The upper dot of a column of a character is bit 0 of the byte containing the byte for that column. This is illustrated in the earlier picture of a 6 × 8 character cell.

To create SYFT, enter the dot patterns (upside-down) into an assembly language source file, then assemble and link the file. Run HXC on the resulting EXE file, specifying file type A and handheld file name SYFT.

Relationship to User-Defined Keys

The HP-94 has 16 keys which have no predefined use: the alphabetic keys whose unshifted keycaps (lower right corner) are unmarked. These are shown as *f1-f16* on the keyboard layout in the "Keyboard" chapter, and correspond to character codes 80h-8Fh, half of the control codes for the upper 128 characters of the built-in Roman-8 character set.

Whether or not these keys cause the corresponding user-defined character to be echoed to the display depends on which operating system function was used to read the keyboard. GET_CHAR and READ for channel 0 (functions 01h and 12h) do not echo user-defined characters, while GET_LINE (02h) does. The only BASIC I/O statements that echo to the display while accepting keyboard input are INPUT and INPUT #, and they both use GET_LINE.

Even when echoing of keyboard input occurs, it will still track the behavior of user-defined characters — that is, echoed as blanks if no SYFT exists or if the machine is in command mode, and echoed as user-defined characters if SYFT exists and a program is running.

10

Serial Port

Contents

Chapter 10 Serial Port

10

Serial Port

The HP-94 serial port is a read/write port controlled by an OKI MSM82C51A Universal Asynchronous Receiver Transmitter (UART). This is a CMOS UART compatible with the Intel 8251A. (It is actually a USART, but the 94 does not provide the additional hardware needed for synchronous operation.) The details of the UART hardware, operation, and usage are described in the Oki MSM82C51A data sheet in the "Hardware Specifications" elsewhere in this manual.

Signal Levels

The serial port signal levels are 0 to V_{cc} (~0-5) volts. Not all devices can operate at those levels, and may require the HP 82470A RS-232-C Level Converter. The converter changes the 0 to V_{cc} signal levels into +9 to -9 volts for those devices that require it. Refer to the "Hardware Specifications" for details on the signal levels as well as the connector pinouts for the serial port and the level converter.

Enabling or Disabling the Serial Port

The 82C51 can be enabled or disabled under software control. Power is supplied to the level converter only when it is enabled; it is only at this time that serial port has any power consumption. When the 82C51 is enabled, the 94 provides a baud rate clock at 16 times the desired baud rate. Before a program transmits or receives with the 82C51, the UART must be set in 16× mode. When the 82C51 has received an entire byte of serial data (including the start and stop bits) and checked for errors (parity, framing, and UART overrun), the serial port data received interrupt (type 53h) will be issued.

Initializing the Serial Port

Below are the things that must be done to initialize the serial port in the OPEN routine of a user-defined serial port handler.

- Take over the existing serial port interrupt vector.
- Set the baud rate clock value.
- Turn on power to the serial port, and wait 60 ms to allow the level converter to power up. This turn-on delay may not accommodate the turn-on or reset time required by the RS-232 device connected to the serial port.

  (Note: when turning off the serial port, the CLOSE routine should wait 60 ms after the 82C51 is disabled to allow signals to stabilize.)
- Reset the 82C51, and set it to the desired initial state.

- Enable the serial port interrupt.

Processing the Serial Port Data Received Interrupt

When the data received interrupt occurs, the following actions should be taken by the interrupt service routine. These are in addition to whatever data processing is done in the routine, and to normal interrupt routine overhead such as reading the end of interrupt register.

- Check if an 82C51 error occurred. If so, clear it.
- Read the data from the serial port data register.

NOTE While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This time may be important to serial port handlers.

Serial Port Control and Status Registers

The serial port control and status registers are summarized below.

Table 10-1. Serial Port Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	3	W
Interrupt Status	00h	3	R
Interrupt Clear	01h	3	W
Baud Rate Clock Value	0Ah	0-2	W
Main Control	0Bh	2	W
Main Status	0Bh	2	R
Serial Port Data	10h	0-7	R/W
Serial Port Control	11h	0-7 *	W
Serial Port Status	11h	0-7 *	R

* For the meaning of the bits in these registers, refer to the Oki MSM82C51A data sheet in the "Hardware Specifications".

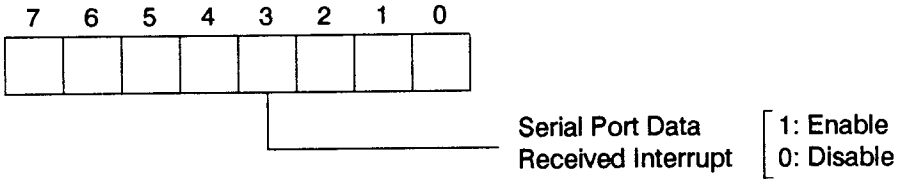


Figure 10-1. Interrupt Control Register (I/O Address 00h, Write)

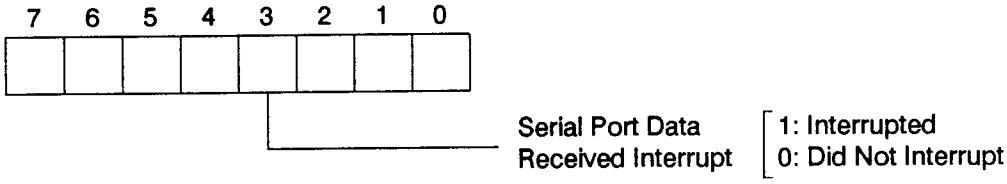


Figure 10-2. Interrupt Status Register (I/O Address 00h, Read)

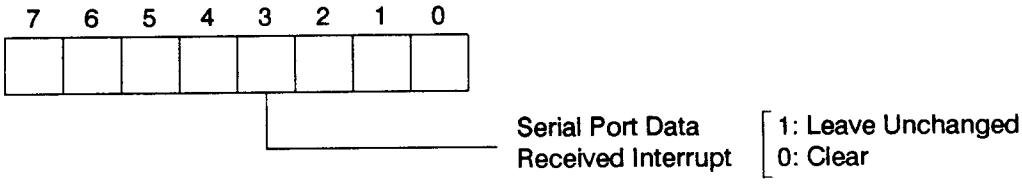
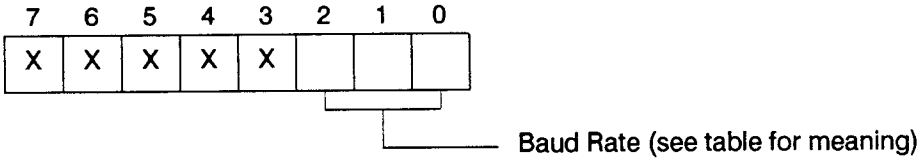


Figure 10-3. Interrupt Clear Register (I/O Address 01h, Write)



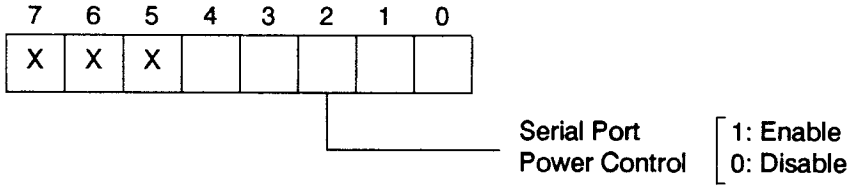
X = don't care

Figure 10-4. Baud Rate Clock Value Register (I/O Address 0Ah, Write)

Table 10-2. Baud Rate Clock Values

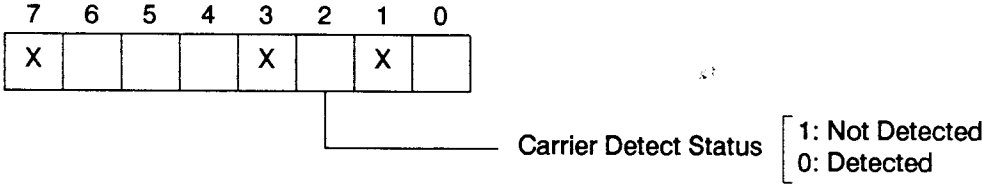
Baud Rate Clock Value	Baud Rate	Frequency (kHz) *
0	19200 †	307.2
1	9600	153.6
2	4800	76.8
3	2400	38.4
4	1200	19.2
5	600	9.6
6	300	4.8
7	150	2.4

* The actual clock frequency is 16 times the desired baud rate.
† Available but not supported.



X = don't care

Figure 10-5. Main Control Register (I/O Address 0Bh, Write)



X = ignore

Figure 10-6. Main Status Register (I/O Address 0Bh, Read)

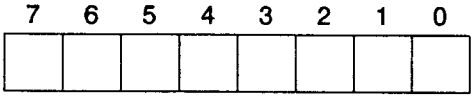


Figure 10-7. Serial Port Data Register (I/O Address 10h, Read/Write)

Built-In Serial Port Handler

The built-in serial port handler is the one used when the serial port is opened and the null string ("") is provided as the handler name. This handler is always used by the C (*copy*) operating system command and by the resident debugger when using the serial port as the console, even when user-defined handlers are available. The handler is designed for use with general serial devices that do not perform hardware handshaking.

Built-In Serial Port Handler Capabilities

The built-in serial port handler provides the following capabilities:

- Full Duplex Communications
  Two-way simultaneous communications.
- Received-Data Buffering
  Received data is placed in a 64-byte buffer. There is no transmit buffer.
- Speeds
  Speeds can be set from 150 to 9600 baud (19200 baud is available but not supported).
- Data Bits
  Seven or eight.
- Parity
  Odd, even, or no parity.
- Stop Bits
  One or two stop bits.
- XON/XOFF Software Handshaking
  When enabled, this option allows received XON (11h) and XOFF (13h) characters to start and stop HP-94 transmissions, and causes XON and XOFF characters to be sent to start and stop host transmissions.
- Null Stripping
  When enabled, this option causes any received NUL characters (00h) to be stripped from and not counted as received data, and not placed in the receive buffer.
- Terminate Character Control
  When defined, a received terminate character will end the wait for a fixed-length block of data, even if all the data has not been received. A terminate character will be sent after sending every block of data.
- Control Lines
  RTS and DTR are raised when the serial port handler is opened, and lowered when the handler is closed. CTS is monitored indirectly by checking if the TxRDY status bit in the 82C51 goes high within three byte-times after attempting to transmit a byte. In addition, V_{rs} (switched V_{cc}) is supplied to power the level converter when the handler is opened, and not supplied when the handler is closed.

The table below describes how the built-in serial port handler behaves. It shows the action taken by the handler routines as well as during its interrupt service routine, not including normal handler activities described in the "User-Defined Handlers" chapter. Note that certain actions, such as sending an XON or responding to a received terminate character, will only occur if the appropriate options were enabled when the handler was opened.

Table 10-3. Behavior of Built-In Serial Port Handler

Routine	Activities
CLOSE	Complete transmission of current byte Disable interrupt 53h Flush receive buffer Lower RTS and DTR Wait 60 ms for signals to stabilize Disable 82C51 and turn off power to serial port
IOCTL	Do nothing
OPEN	Flush receive buffer Enable 82C51 and supply power to level converter Wait 60 ms for level converter turn on Initialize operating configuration * Raise RTS and DTR Enable interrupt type 53h Send single XON Ignore parity, framing, overrun, and receive buffer overflow errors
POWERON	Do nothing
READ	Monitor and report low battery, power switch, and timeout errors † Report errors detected in interrupt service routine Send XON when receive buffer emptied End and report error 74h (116) if terminate character detected Return data from receive buffer
RSVD2	Do nothing
RSVD3	Do nothing
TERM	Do nothing
WARM	Perform all OPEN routine activities except sending XON
WRITE	Monitor and report low battery, power switch, and timeout errors † Monitor CTS indirectly and report error DAh (218) if lost Write data to 82C51 Send terminate character at end of data
Interrupt Service	Monitor parity, framing, overrun, and receive buffer overflow errors Read data from 82C51 and accumulate data into receive buffer Disable transmission when XOFF received Enable transmission when XON received Send XOFF for each byte when buffer 3/4 full Strip nulls (00h)

* Baud rate, data format, XON/XOFF handshaking, null stripping, and terminate character.

† System timeout restarts after each byte received or transmitted.

The errors reported by the built-in serial port handler are shown in the following table.

Table 10-4. Errors Reported by Built-In Serial Port Handler

Routine	Errors
CLOSE	None
IOCTL	None
OPEN	65h
POWERON	None
READ	74h,76h,77h,C8h,C9h,CAh,CBh,CCh,CDh,CEh,CFh,D0h
RSVD2	None
RSVD3	None
TERM	None
WARM	None
WRITE	76h,77h,C8h,DAh
Interrupt Service *	C9h,CAh,CBh,CCh,CDh,CEh,CFh,D0h
† Detected by interrupt service routine, but reported by READ routine.	

Parameters at OPEN Time

When the built-in serial port handler is opened, DS : DX must point to a three-byte parameter area. The meanings of the parameters are shown below. In these figures, the offsets are from DS : DX.

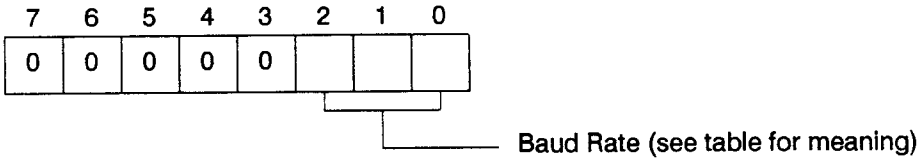
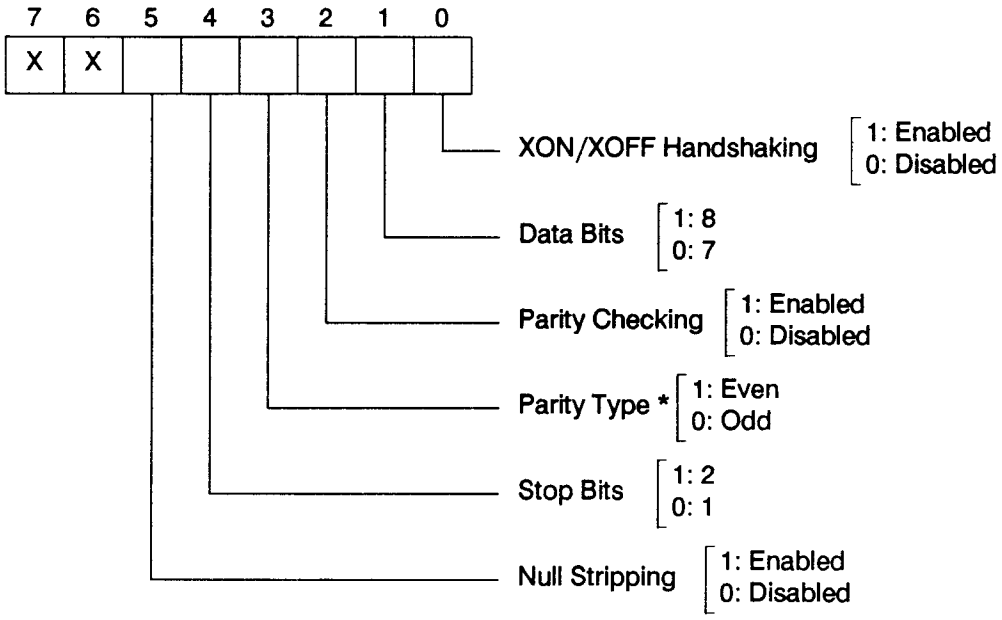


Figure 10-8. Baud Rate — Parameter Byte 1 (Offset 00h)

Table 10-5. Built-In Serial Port Handler Baud Rate Values

Value	Baud Rate
0	19200 *
1	9600
2	4800
3	2400
4	1200
5	600
6	300
7	150
* Available but not supported.	



X = don't care

Figure 10-9. Data Format — Parameter Byte 2 (Offset 01h)

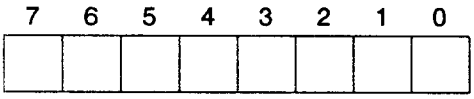


Figure 10-10. Terminate Character † — Parameter Byte 3 (Offset 02h)

The default values for the parameters are 01h (9600 baud), 0Dh (XON/XOFF enabled, 7 data bits, parity checking enabled, even parity, one stop bit, and null stripping disabled), and 00h (no terminate character).

Control Line Behavior

The 82C51 can monitor or control only a subset of the standard RS-232 control lines. Of those lines not monitored, one can be monitored indirectly, and one can be monitored using other HP-94 hardware. Not all these hardware capabilities are actually used by the built-in serial port handler. The usage is summarized below.

---

* The parity type is ignored if parity checking is disabled.

† To disable use of the terminate character, set it to zero.

Table 10-6. Control Line Behavior

Control Line		Monitored or Controlled By Hardware	Monitored or Controlled By Built-In Handler
Symbol	**Name**		
CTS	clear to send	monitored *	monitored *
DSR	data set ready	monitored	not monitored
DCD	data carrier detect	monitored	not monitored
RTS	request to send	controlled	controlled
DTR	data terminal ready	controlled	controlled

* Monitored indirectly by checking if the TxRDY status bit in the 82C51 goes high within three byte-times after attempting to transmit a byte.

A user-defined serial port handler could use all the lines supported by the hardware. Refer to the "User-Defined Handlers" chapter for details on how to write a user-defined serial port handler.

When the serial port is disabled, the control lines are turned off (set to 0 volts).

Operating System Functions

The serial port software implements the following operating system functions:

Table 10-7. Serial Port-Related Operating System Functions

Function Name	Function Code
BUFFER_STATUS	06h
OPEN	0Fh
CLOSE	10h
READ	12h
WRITE	13h

11

Bar Code Port

Contents

Chapter 11 Bar Code Port

11

Bar Code Port

The HP-94 bar code port is a read-only port designed to connect to bar code scanning devices such as wands. The port provides power to the external device. Interrupt control, timing for light and dark transitions, and light or dark state is available to programs reading bar code data.

Bar Code Port Power and Transition Detection

The main control register is used to enable power to the bar code port (and to the device attached to it) and, independently, to enable transition detection at the port. Once the port is powered and detecting transitions, interrupt type 52h will be issued whenever a transition occurs at the port — either light-to-dark or dark-to-light. When the interrupt occurs, the light or dark state is indicated by reading the main status register.

Bar Code Timer

The bar code timer is a 12-bit count-up timer with a 26 μs interval. This resolution allows timing intervals from 26 μs to 106.7 ms. Because it is a count-up timer, it must be set using the complement of the desired number of intervals. When the timer overflows (counts up to zero), interrupt type 51h is generated. This is usually used to indicate the end of a scan.

When the timer reaches zero, it is automatically reset to its starting value and restarted. If the count value has to be set to a specific value, the timer must be stopped first. Unlike the system timer, the bar code timer can be reset to zero while it is still running.

When the bar code port transition interrupt occurs, the timer value can be captured (i.e., placed in the timer data registers where it can be read) to indicate how long the bar code port has been at the current state. Then the timer can be reset to zero to continue counting up for the next transition. The value can be captured while the timer is still running.

Initializing the Bar Code Port

Below are the things that must be done to initialize the bar code port.

- Take over the existing bar code port transition and timer interrupt vectors.
- Turn on power to the bar code port, and enable transition detection.
- Set the bar code timer to the desired initial value (or clear it), and start the timer.
- Enable the bar code port transition and timer interrupts.

Some of the initialization activities will be done in the OPEN routine of a bar code port handler, while

others will be done in the READ routine. This will be discussed shortly.

Processing the Bar Code Port Transition Interrupt

When the transition interrupt occurs, the following actions should be taken by the interrupt service routine. These are in addition to whatever data processing is done in the routine and to normal interrupt routine overhead such as reading the end of interrupt register.

- Capture the current timer value into the timer data registers (04h and 05h) by writing to the timer value capture register (07h).
- Read the captured timer data from the timer data registers.
- Reset the timer to the desired value. If it is a specific value, stop the timer with the timer control register (06h), set the values, and restart it. If it is only necessary to clear the timer, do so by writing to the timer clear register (08h).
- Determine if the state after the transition is light or dark by reading the main status register (0Bh).

Bar Code Port Timing Constraints

The bar code port transition interrupt occurs on every transition. This requires an order of magnitude more processing time than the serial port, since its interrupt occurs only after the 82C51 has received 10-12 transitions (bits) of serial data. Experience has shown that it is unlikely that a bar code port handler can be run "in the background" to simply fill a receive buffer. When other interrupts occur, the CPU interrupt flag will be cleared while the corresponding interrupt service routine executes. This results in periods of time when bar code port transition interrupts occur but cannot be processed, and therefore may be missed.

To deal effectively with these timing constraints, a bar code port handler should only process bar code data during its READ routine. The transition and timer interrupts should only be enabled then, and certain other interrupts should be disabled to prevent transitions from being missed. The machine should essentially become dedicated to the sole task of reading bar code transitions for the duration of the READ operation. This is in contrast to a serial port handler, which can run "in the background", save data in its receive buffer when interrupts occur, and return the data in the buffer when its READ routine is called.

The particular interrupts that should be disabled are the system timer (50h) and serial port data received (53h). The latter has the side effect that data cannot be received by the serial port while bar code labels are being scanned. The former has the side effect that the events paced by the system timer will not occur for the period of time that the timer interrupt is disabled. Refer to the "Timers" chapter for details. There are utility routines available to perform some of these tasks (scan keyboard and blink cursor) without clearing the CPU interrupt flag. Refer to the appendixes for details.

The low main battery voltage (54h) and power switch (55h) interrupts should remain enabled, since those events need to be monitored by the handler to determine if it should abort a read operation.

NOTE While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. While checking to see if the beeper needs to be turned off, interrupts are disabled for ~50 μs. These times may be important to bar code port handlers.

Bar Code Port Control and Status Registers

The bar code port control and status registers are shown below.

Table 11-1. Bar Code Port Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	1-2	W
Interrupt Status	00h	1-2	R
Interrupt Clear	01h	1-2	W
Bar Code Timer Data	04h	0-7	R/W
Bar Code Timer Data	05h	0-3	R/W
Bar Code Timer Control	06h	0	W
Bar Code Timer Value Capture	07h	None	W
Bar Code Timer Clear	08h	None	W
Main Control	0Bh	3-4	W
Main Status	0Bh	0	R

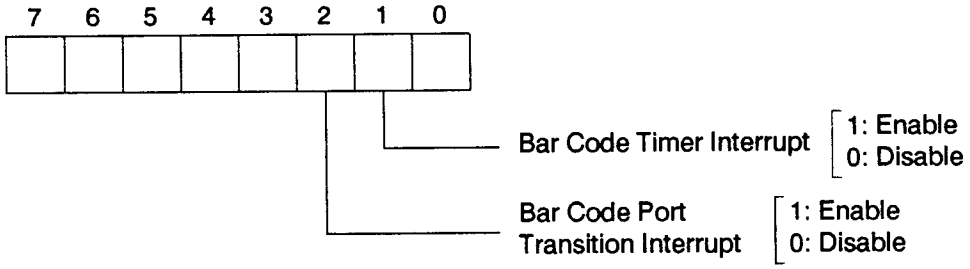


Figure 11-1. Interrupt Control Register (I/O Address 00h, Write)

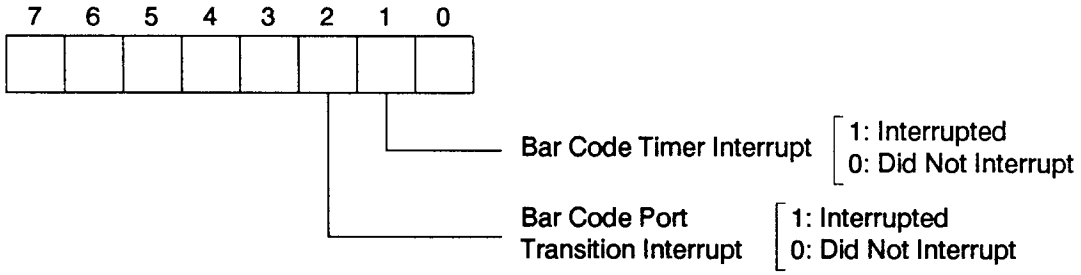


Figure 11-2. Interrupt Status Register (I/O Address 00h, Read)

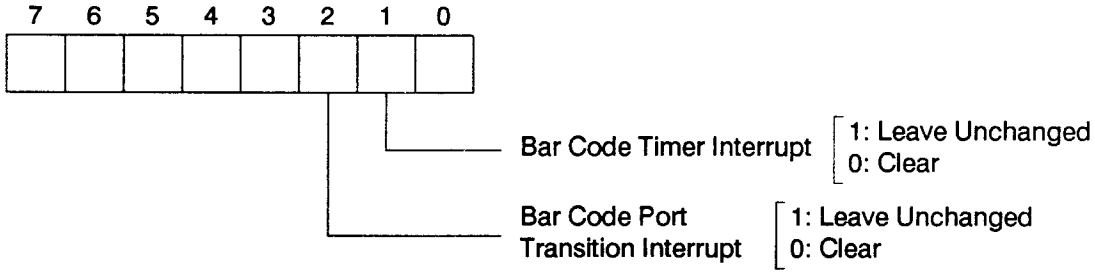


Figure 11-3. Interrupt Clear Register (I/O Address 01h, Write)

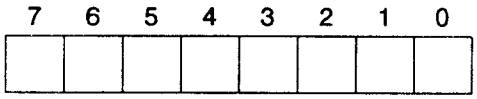
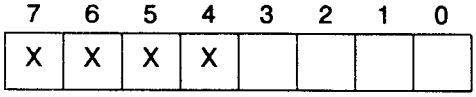


Figure 11-4. Bar Code Timer Data Register * (I/O Address 04h, Read/Write)

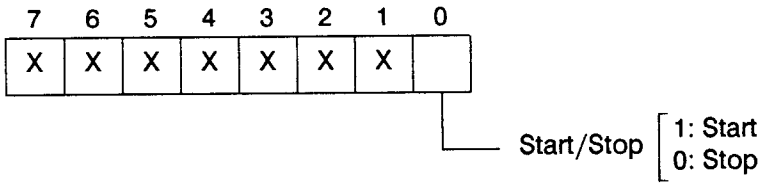


X = ignore

Figure 11-5. Bar Code Timer Data Register † (I/O Address 05h, Read/Write)

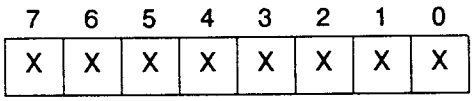
* Lower 8 bits of the 12-bit timer value.

† Upper 4 bits of 12-bit timer value.



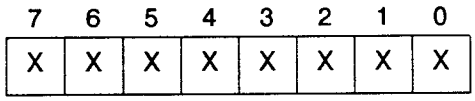
X = don't care

Figure 11-6. Bar Code Timer Control Register (I/O Address 06h, Write)



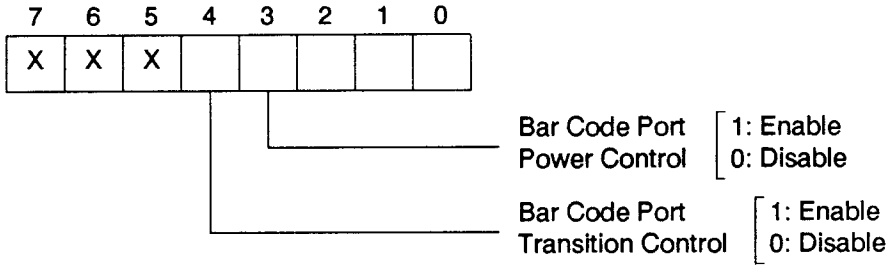
X = don't care

Figure 11-7. Bar Code Timer Value Capture Register (I/O Address 07h, Write)



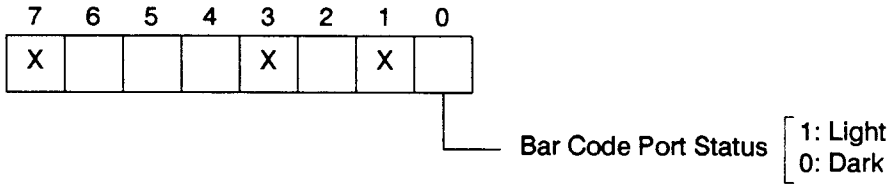
X = don't care

Figure 11-8. Bar Code Timer Clear Register (I/O Address 08h, Write)



X = don't care

Figure 11-9. Main Control Register (I/O Address 0Bh, Write)



X = ignore

Figure 11-10. Main Status Register (I/O Address 0Bh, Read)

12

Timers

Contents

Chapter 12 Timers

12

Timers

The HP-94 has two timers available other than the real-time clock: the system timer and the bar code timer. These use a different time base than the real-time clock, and their accuracy is ±0.1%.

Table 12-1. HP-94 Timers

Timer Name	No. of Bits	Time Interval	Timer Type	Overflow Interval	Overflow Interrupt	Maximum Time
System	8	0.417 ms	up	5 ms	50h	106.7 ms
Bar Code	12	26 μs	up	— *	51h	106.7 ms
* Not defined by the operating system. Defined only by bar code port handler.						

System Timer

The system timer is an 8-bit count-up timer with an interval of 0.417 ms. It is initialized to -12 (-0Ch), so it overflows (counts up to zero) every 5 ms (12 * 0.417 = 5 ms, complemented because it is a count-up timer). When the system timer overflows, interrupt type 50h is generated. This interrupt is used to pace six different events in the operating system, shown below. While these events are checked and appropriate action is taken, interrupts are enabled except during the beeper event.

Table 12-2. Events Checked By System Timer Interrupt Routine

Timing Event	How Often Event Checked	How Often Action Taken
Scan Keyboard	5 ms	Put key into key buffer after 25 ms debounce Start key repeat if key still down after 675 ms Repeat key every 115 ms
Turn Off Beeper	10 ms	Turn beeper off after current beep time expires
Blink Cursor	100 ms	Blink cursor every 500 ms
System Timeout	1 s	Turn off machine or execute user-defined power switch/timeout routine after current system timeout expires
Display Backlight Timeout	1 s	Turn off backlight after current backlight timeout expires
Background Timer	1 s	Execute background timer interrupt routine every 1 s

NOTE While the beeper is checked to see if it needs to be turned off, interrupts are disabled for ~50 μs. This time may be important to bar code port handlers.

System Timeout

The system timeout is the time after which the machine will automatically turn off. It can be set from 0-1800 seconds using the TIMEOUT function (09h). The timeout is in effect while the machine is waiting for keyboard input or for data to be received at the serial or bar code ports. It will abort read operations from channels 0-4 and write operations to channels 1-4. It will not abort create, read, write, or delete operations for channels 5-15. The operating system will take one of the following actions when the system timeout expires:

- Turn off the HP-94.
  This is the default behavior if the program has not defined a power switch/timeout routine using the SET_INTR function (0Ah). The next time the machine is turned on, it will cold start.
- Execute the user-defined power switch/timeout interrupt routine.
  If the program has defined a power switch/timeout routine with SET_INTR, that routine will be executed with a FAR CALL (and therefore must end with a FAR RET). The AL register will be set to 76h, the timeout error, and the DS register will be set to the value specified when SET_INTR was called. This will only occur during a running program, not in command mode. When timeouts are monitored during I/O by a user-defined handler, the handler must execute the user-defined interrupt routine.
- Ignore the system timeout.
  If the program has disabled the system timeout by setting the timeout value to 0 with TIMEOUT, the operating system will ignore the system timeout.

The TERM routine of any open user-defined handlers will *not* be executed. Since each handler must monitor the system timeout itself, that handler will be the only one waiting on I/O when the timeout expires. Consequently, it is the only one that needs to terminate I/O.

Display Backlight Timeout

The display backlight timeout is the time after which the machine will automatically turn off the display backlight. This timeout is in effect whenever the backlight is on. It can be set from 0-1800 seconds using the TIMEOUT function.

When the display control code is processed to turn on or off the display backlight, the operating system controls the backlight state when keyboard scanning is done. If the system timer is disabled, no scanning is done, so the backlight will not be controlled. If a program disables the system timer, it must turn on the backlight explicitly using the keyboard control register, and then turn the backlight off explicitly after the timeout expires.

CAUTION Leaving the display backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.

Background Timer

The background timer is a one-second heartbeat timer that the machine provides for assembly language programs to use. Once a second, the operating system will issue a FAR CALL to the address in interrupt vector 1Ch, the background timer interrupt.

To take over the background timer interrupt, the program must do the following:

- Read the background timer interrupt vector (address 1Ch * 4 = 00070h), and save it in the program's scratch area.
- Write the address of the program's background timer interrupt routine into the vector location. The instruction pointer (IP) offset should be stored at the first word, and the code segment (CS) address should be stored at the second word.

To use the background timer interrupt, the program must do the following:

- When the interrupt routine is called, perform whatever processing is necessary.
- At the end of the routine, execute a FAR JMP to the address of the previous background timer interrupt routine.

The FAR JMP has the effect of daisy-chaining all the background timer interrupt routines together, allowing different programs to share the same interrupt. The last routine in the chain is the default routine, which is simply a FAR RET to end the aggregate background timer interrupt.

If the background timer does not provide enough resolution (1 second) for the program, the program can take over the system timer interrupt (vector at address 50h * 4 = 00140h) in the same manner (save the current interrupt vector, and FAR JMP to it at the end of the interrupt routine). This will provide a 5 ms timing resolution.

CAUTION The background timer routine must not clear the CPU interrupt flag (CLI). Doing so may cause interrupts from hardware devices to be delayed long enough that time-critical interrupt service routines (for open user-defined handlers) may miss their data.

Bar Code Timer

The second timer is the bar code timer, a 12-bit count-up timer with an interval of 26 μs. It is reserved for use by bar code port handlers, so it is never initialized to any value by the operating system. Like the system timer, it must be set using the complement of the desired number of intervals. When it overflows, interrupt type 51h is generated.

When either timer reaches zero, the timer is automatically reset to its starting value and restarted. If

the count value has to be set to a specific value, the timer must be stopped first. The bar code timer can be reset to zero or have its current value captured while it is still running.

Timer Control and Status Registers

The timer control and status registers are shown below.

Table 12-3. Timer Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	0-1	W
Interrupt Status	00h	0-1	R
Interrupt Clear	01h	0-1	W
System Timer Data	02h	0-7	R/W
System Timer Control	03h	0	W
Bar Code Timer Data	04h	0-7	R/W
Bar Code Timer Data	05h	0-3	R/W
Bar Code Timer Control	06h	0	W
Bar Code Timer Value Capture	07h	None	W
Bar Code Timer Clear	08h	None	W

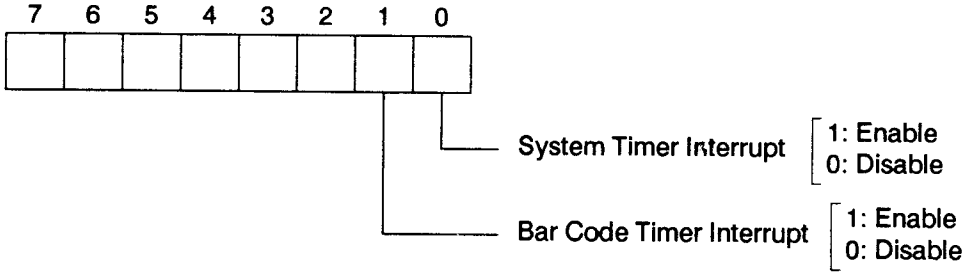


Figure 12-1. Interrupt Control Register (I/O Address 00h, Write)

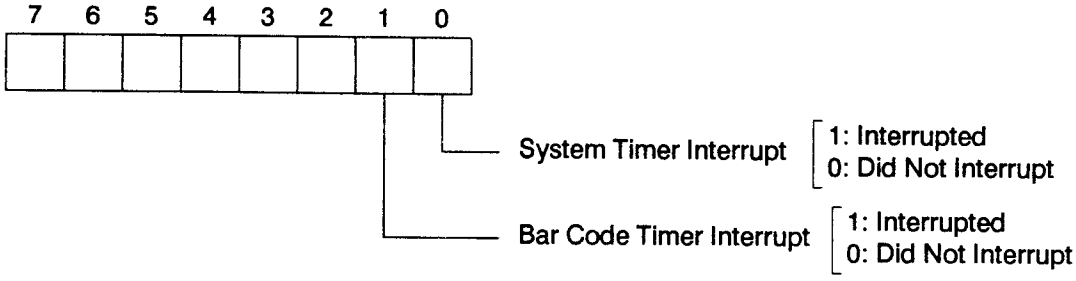


Figure 12-2. Interrupt Status Register (I/O Address 00h, Read)

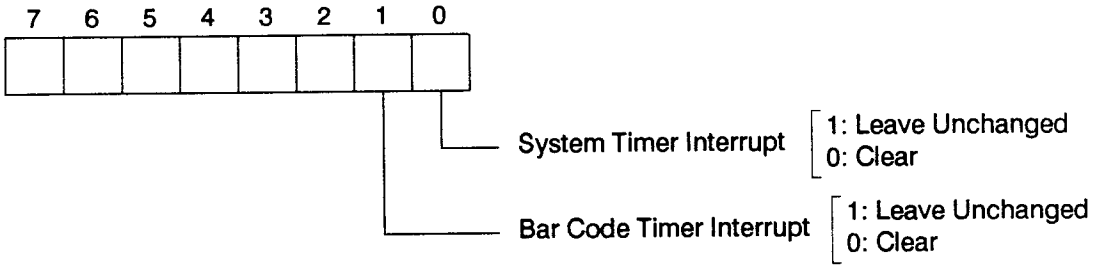


Figure 12-3. Interrupt Clear Register (I/O Address 01h, Write)

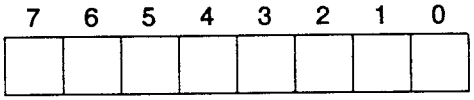
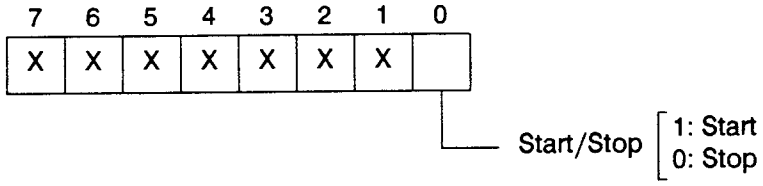


Figure 12-4. System Timer Data Register (I/O Address 02h, Read/Write)



X = don't care

Figure 12-5. System Timer Control Register (I/O Address 03h, Write)

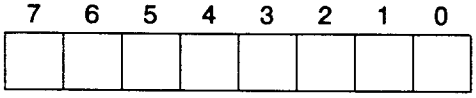
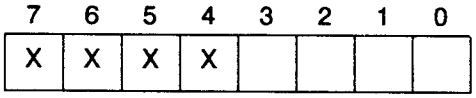
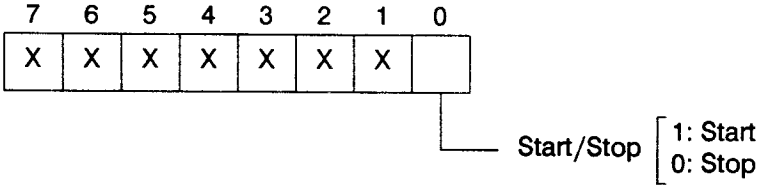


Figure 12-6. Bar Code Timer Data Register * (I/O Address 04h, Read/Write)



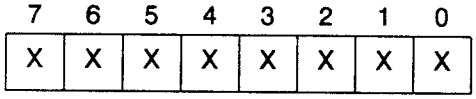
X = ignore

Figure 12-7. Bar Code Timer Data Register † (I/O Address 05h, Read/Write)



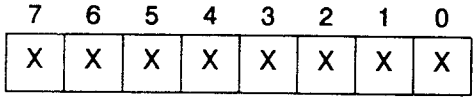
X = don't care

Figure 12-8. Bar Code Timer Control Register (I/O Address 06h, Write)



X = don't care

Figure 12-9. Bar Code Timer Value Capture Register (I/O Address 07h, Write)



X = don't care

Figure 12-10. Bar Code Timer Clear Register (I/O Address 08h, Write)

* Lower 8 bits of the 12-bit timer value.

† Upper 4 bits of 12-bit timer value.

Operating System Functions

The timer software implements the following operating system functions:

Table 12-4. Timer-Related Operating System Functions

Function Name	Function Code
TIMEOUT	09h
SET_INTR	0Ah

13

Power Switch

Contents

13

Power Switch

The HP-94 power switch provides software control for turning the machine off. When the HP-94 is off, pressing the power switch turns the machine on. When the machine is on, pressing the power switch generates interrupt type 55h. The power switch interrupt will abort read operations from channels 0-4 and write operations to channels 1-4. It will not abort create, read, write, or delete operations for channels 5-15. The operating system will take one of the following actions in response to this interrupt:

- Turn off the HP-94.
  This is the default behavior if the program has not defined a power switch/timeout interrupt routine using the SET_INTR function (0Ah). The next time the machine turns on, it will cold start.
- Execute the user-defined power switch/timeout routine.
  If the program has defined a power switch/timeout interrupt routine with SET_INTR, that routine will be executed with a FAR CALL (and therefore must end with a FAR RET). The AL register will be set to 77h, the power switch error, and the DS register will be set to the value specified when SET_INTR was called. This only occurs when the power switch is pressed during a running program, not in command mode.
- Ignore the power switch.
  If the program has disabled the power switch with SET_INTR, the operating system will respond to the interrupt but take no action, thereby ignoring the power switch.

In the first two cases, the TERM routine of any open user-defined handlers will be executed before the action is taken.

To turn the machine off, the operating system writes to the power control register.

Power Control and Status Registers

The power control and status registers are shown below.

Table 13-1. Power Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	5	W
Interrupt Status	00h	5	R
Interrupt Clear	01h	5	W
Power Control	1Bh	None	W

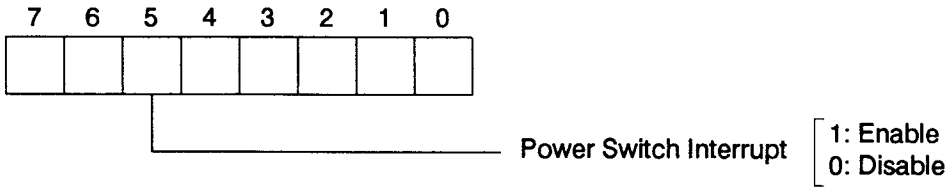


Figure 13-1. Interrupt Control Register (I/O Address 00h, Write)

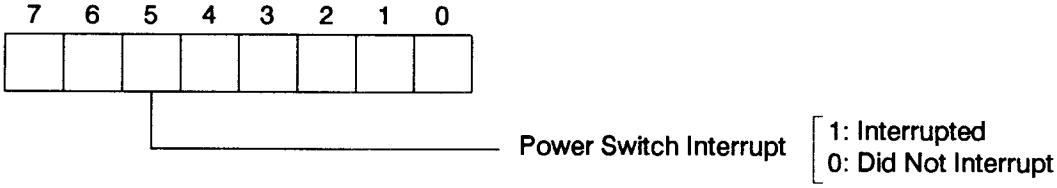


Figure 13-2. Interrupt Status Register (I/O Address 00h, Read)

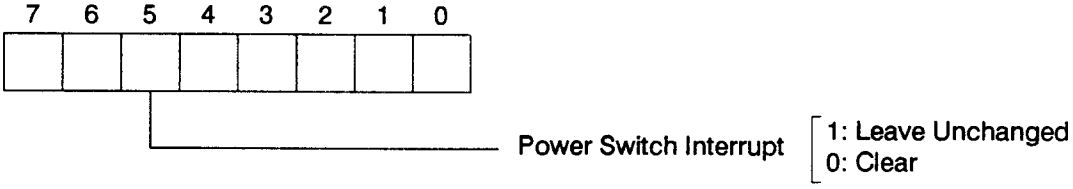
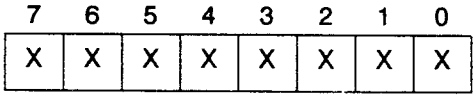


Figure 13-3. Interrupt Clear Register (I/O Address 01h, Write)



X = don't care

Figure 13-4. Power Control Register (I/O Address 1Bh, Write)

Operating System Functions

The power switch software implements the following operating system functions:

Table 13-2. Power Switch-Related Operating System Functions

Function Name	Function Code
END_PROGRAM	00h
SET_INTR	0Ah

14

Batteries

Contents

14

Batteries

The HP-94 contains two types of batteries: nickel-cadmium batteries as the main power source, and lithium batteries for memory backup. Details about the characteristics of these batteries is in the "Hardware Specifications" elsewhere in this manual.

Main Nickel-Cadmium Battery Pack

The main power source for the machine is a rechargeable nickel-cadmium (NiCd) battery pack with a nominal capacity of 900 mAh. The machine operating voltage (which is slightly below the battery pack voltage) is continuously checked by the low battery detection circuitry whenever the machine is on. When the operating voltage drops to 4.6 ± 0.05 volts or below, interrupt type 54h is generated. The low battery interrupt will abort read operations from channels 0-4 and write operations to channels 1-4. It will not abort create, read, write, or delete operations for channels 5-15. The operating system will take one of the following actions in response to this interrupt:

- Halt all machine activities, issue error 200, and wait for the user to press the power switch to turn the machine off.
  This is the default behavior if the program has not defined a low battery interrupt routine using the SET_INTR function (0Ah). The following activities are halted:

Table 14-1. Activities Halted During Default Low Battery Behavior

Activity or Device	Action Taken
Cursor	Turned Off
Interrupts	Disabled
System Timer	Turned Off
Bar Code Timer	Turned Off
Beeper	Turned Off
Keyboard	Disabled
Display Backlight	Turned Off
Serial Port	Disabled
Serial Port Power	Turned Off
Bar Code Power	Turned Off
Bar Code Transitions	Disabled

  The next time the machine is turned on, it will cold start.

- Execute the user-defined low battery routine.
  If the program has defined a low battery interrupt routine with SET_INTR, that routine will be executed with a FAR CALL (and therefore must end with a FAR RET). The DS register will be set to the value specified when SET_INTR was called. This only occurs during a running program,

not in command mode.

In both cases, the TERM routine of any open user-defined handlers will be executed before the action is taken. SET_INTR does not allow disabling the low battery interrupt.

The low battery interrupt only occurs once, when the main battery voltage drops below 4.6 volts. At that point, the program has 2-5 minutes left before the battery voltage drops so low that the machine turns itself off automatically without warning. The low battery interrupt will not occur again until the machine has been turned off and back on. If the battery remains below 4.6 volts while the 94 is off, the machine will not turn back on again until the battery has been recharged enough to bring its voltage above that level (~4.8 volts). (The machine actually turns on, but the operating system turns it off before any memory integrity tests are performed if the voltage is too low.)

The actual amount of time available depends on what is happening when the low battery condition occurs. For example, the display backlight takes more power, as does the HP 82470A RS-232-C Level Converter (if one is connected to the serial port), so less operating time will be available if these are on. The time also depends on how much the battery was charged during its last charging cycle, the ambient temperature, and many other factors. Because the remaining operating time is variable, the program should respond to the low battery interrupt as rapidly as possible by ending its activities (shut off I/O and powered devices, complete file updates that were in progress, etc.), notifying the user that it is necessary to recharge the main battery, and turning the power off.

If the program continues operating until the machine turns itself off automatically, the effect is as if the reset switch was pressed. No data in data files will be lost, since the backup batteries will keep memory intact, but the machine will cold start the next time it is turned on. This means that any data in program variables or scratch areas that did not get saved in a data file will be lost.

Backup Lithium Batteries

The backup power source is user-replaceable 3-volt lithium backup batteries, CR-2032 or equivalent. There is one lithium battery for each major block of RAM: one for the first 64 or 128K (which also backs up the real-time clock), one for the 128K memory board, and one for the 40K RAM card. The mainframe lithium batteries are accessible through the back cover, and the RAM card battery is under a cover on the card. These batteries are only used to preserve the contents of memory when the main NiCd battery pack is completely discharged or disconnected (and there is no recharger connected). They are not used when other power sources are available to preserve memory.

Their state is checked and reported only when the machine is turned on, after all memory integrity tests are performed. Error 210 is reported at power on to indicate low voltage (2.7 volts) of the battery for the first 64 or 128K, while error 211 is reported for the memory board or RAM card battery. Both errors will be reported if both batteries need replacing.

Battery Control and Status Registers

The battery control and status registers are shown below.

Table 14-2. Battery Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	4	W
Interrupt Status	00h	4	R
Interrupt Clear	01h	4	W
Main Status	0Bh	4-6	R

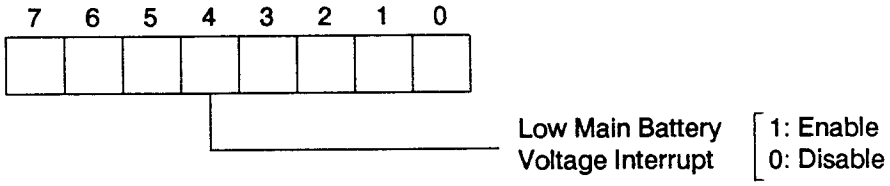


Figure 14-1. Interrupt Control Register (I/O Address 00h, Write)

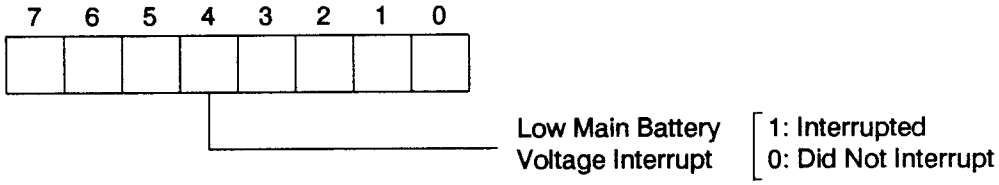


Figure 14-2. Interrupt Status Register (I/O Address 00h, Read)

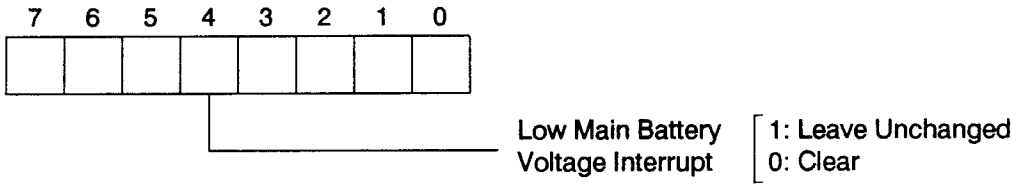
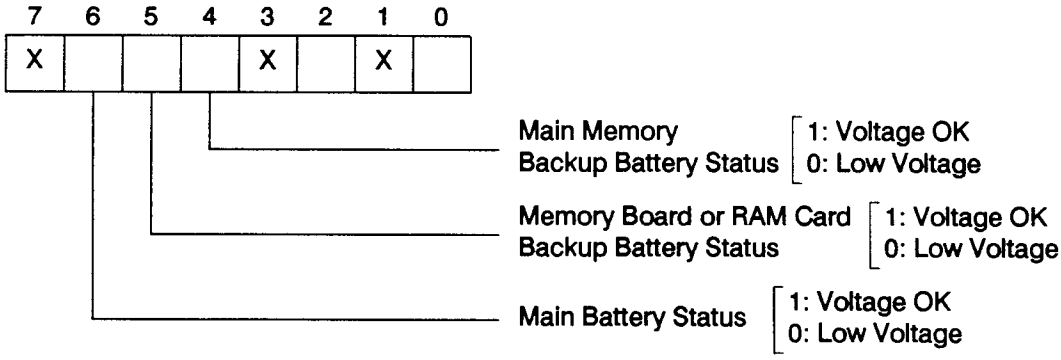


Figure 14-3. Interrupt Clear Register (I/O Address 01h, Write)



X = ignore

Figure 14-4. Main Status Register (I/O Address 0Bh, Read)

Operating System Functions

The battery software implements the following operating system functions:

Table 14-3. Battery-Related Operating System Functions

Function Name	Function Code
SET_INTR	0Ah

15

Real-Time Clock

Contents

15

Real-Time Clock

The HP-94 contains an Epson RTC-58321 real-time clock. Its quartz crystal operates at 32768 Hz, and is backed up by the main memory lithium backup battery if the main NiCd battery is completely discharged or removed. The clock has a one-second resolution, and is accurate to ±50 ppm (~2 minutes/month). The clock supports time, date, and day-of-week functions, but the clock software in the operating system only supports time and date, as well as the T (*time*) operating system command. Leap years are accommodated automatically. The details of the real-time clock hardware, operation, and usage are described in the Epson RTC-58321 data sheet in the "Hardware Specifications".

The operating system provides the TIME_DATE function (08h) to set or read the time and date. No syntax checking is performed on the time and date when they are set. It is the responsibility of the application program to ensure that the time and date are in the proper format when they are set.

Real-Time Clock Control and Status Registers

The real-time clock control and status registers are shown below.

Table 15-1. Real-Time Clock Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Real-Time Clock Control/Data	0Ch	0-7 *	W
Real-Time Clock Status/Data	0Ch	0-4 *	R

* For the meaning of the bits in these registers, refer to the Epson RTC-58321 data sheet in the "Hardware Specifications".

Operating System Functions

The real-time clock software implements the following operating system functions:

Table 15-2. Real-Time Clock-Related Operating System Functions

Function Name	Function Code
TIME_DATE	08h

16

Beeper

Contents

16

Beeper

The HP-94 beeper is a piezoelectric buzzer that is turned on and off using the main control register. If a program turns the beeper on explicitly, it is responsible for turning it off as well after the appropriate duration. If a program uses the operating system BEEP function (07h), the operating system will turn the beeper off automatically after the specified time has elapsed.

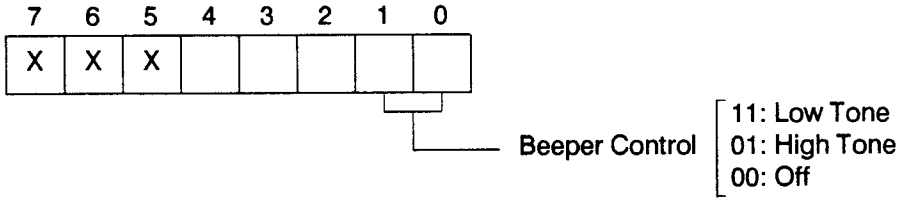
The BEEP function allows specifying beep durations from 0.1 to 25.5 seconds, and either high or low tones. It can be called while the beeper is beeping. If the tone specified is different than the tone in progress, beeping will continue at the high tone and duration — the high tone and its duration will take precedence regardless of the order in which the tones were specified. If the tone specified is the same as the tone in progress, beeping will continue at either the remaining duration or the new duration, whichever is longer.

Beeper Control and Status Registers

The beeper control and status registers are shown below.

Table 16-1. Beeper Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Main Control	0Bh	0-1	W



X = don't care

Figure 16-1. Main Control Register (I/O Address 0Bh, Write)

Operating System Functions

The beeper software implements the following operating system functions:

Table 16-2. Beeper-Related Operating System Functions

Function Name	Function Code
BEEP	07h

17

Reset Switch

17

Reset Switch

The HP-94 has a small reset switch to the left of the power switch. Since the power switch is under program control, it is possible for a program to inadvertently prevent the user from turning off the machine. The reset switch is provided to accommodate this situation.

The reset switch is a hardware power off, not a software power off. When the reset switch is pressed, the machine is turned off immediately. No data in data files will be lost, since the backup batteries will keep memory intact, but the machine will cold start the next time it is turned on. This means that any data in program variables or scratch areas that did not get saved in a data file will be lost.

The TERM routine of any open user-defined handlers will not be executed, and no power-off checksums will be computed. The next time the machine is turned on, it will not compute power-on checksums (although the other memory integrity tests will be performed).

18

Other Hardware

Contents

18

Other Hardware

The HP-94 has some other hardware elements that will be discussed here: read/write memory (RAM), system ROM, custom gate array, earphone jack and external bus connector.

Read/Write Memory (RAM)

HP-94 read/write memory is Toshiba TC5565FL-15L CMOS static RAM (8K × 8). Refer to the "Memory Management" chapter for a detailed description of the memory organization. Major hardware blocks of memory are backed up by user-replaceable lithium backup batteries; refer to the "Batteries" chapter for details.

System ROM

The HP-94 has 32K of EPROM located in the upper 32K of the CPU address space. The system ROM contains all the HP-94 built-in software. Refer to the "Memory Management" chapter for a detailed description of the system ROM organization.

Custom Gate Array

The HP-94 contains a proprietary Hitachi 61L224 custom gate array that combines what would otherwise be several separate integrated circuits (ICs). The following is a list of the major hardware facilities provided by the gate array:

- Interrupt controller for HP-94 hardware interrupts.
- Hardware control registers (except for keyboard, display, and 82C51).
- Power off control.
- System timer.
- Serial port power and baud rate clock.
- Bar code port power control, transition detection, and timer.
- Real-time clock control.
- Beeper tone.
- Chip select address decoding.
- Address/data bus latches.
- Status of data carrier detect (DCD) control line.

Earphone Jack

The earphone jack accepts any standard earphone with a 3.5 mm plug. It allows the user of the machine to hear the beeper (particularly for applications using bar code) in noisy environments.

External Bus Connector

The external bus connector is located on the underside of the HP-94 behind a hard plastic port cover. It brings out all lines from the internal system bus. Details about the external bus connector (pin assignments, voltages, currents, and logic levels are described in the "Hardware Specifications".

Part 2

BASIC Interpreter

1

BASIC Program and Data Structure

Contents

Chapter 1 BASIC Program and Data Structure

1

BASIC Program and Data Structure

BASIC application programs (type B) are interpreted by the HP-94 BASIC interpreter (SYBI). The BASIC application program may be in either RAM or ROM.

BASIC Program Organization

The following figure shows the organization of a BASIC program.

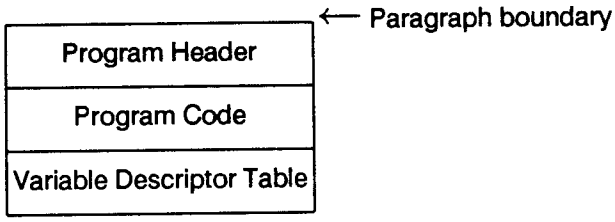


Figure 1-1. BASIC Program Organization

A variable area is necessary to execute a BASIC program. A control information save area is necessary when a CALL statement or an interrupt process routine is executed. The variable area and the control information save area are dynamically allocated in main memory.

BASIC Program Outline

BASIC programs start with a 10h byte program header. The contents of the header are shown below with hex offsets listed on the left side and a brief description on the right. The program code and variable descriptor table are also shown to illustrate their location and size.

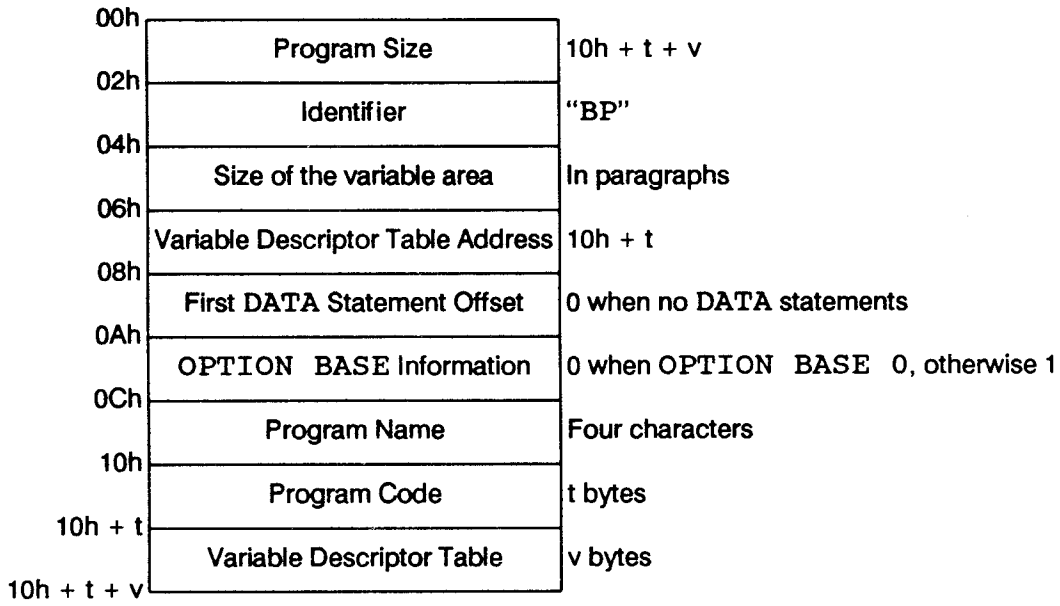


Figure 1-2. Program Header

The following figure shows the organization of the BASIC program code.

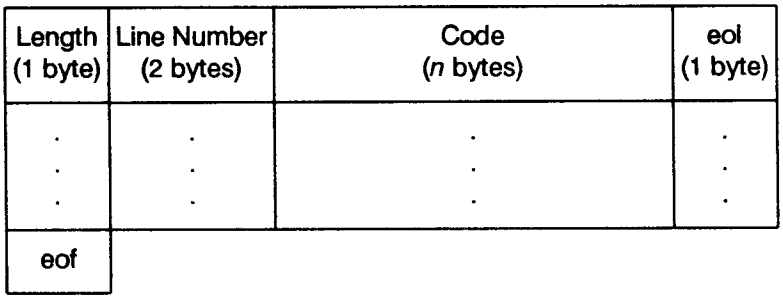


Figure 1-3. Program Code

The information contained in a line of program code is:

- Length: number of bytes in a line = 1 + 2 + *n* + 1 (must be less than 256).
- Line Number: 0 through 32767 (0000h through 7FFFh, least significant 8 bits first).
- eol (end of line) and eof (end of file) are the NUL character (00h).
- Some lines, such as comments, generate no program code.

Type (1 byte)	Length (1 byte)	Segment Address (2 bytes)	Offset Address (1 byte)

Figure 1-4. Variable Descriptor Table

The variable descriptor table contains information about the type, length, and address of each variable. The figure above illustrates the table organization. The meanings of the fields are as follows:

- Type:

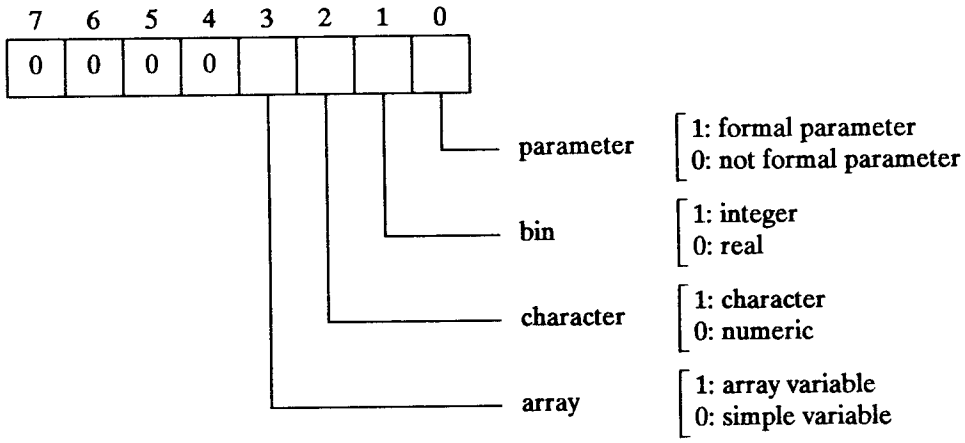


Figure 1-5. Variable Descriptor Type Byte

- Length:

Table 1-1. Variable Descriptor Length Byte

Type	Length in Bytes
Integer	2
Real	8
Character	Dimensioned size (default 8)
Parameter	5 *
Array	Size of one array element

* This entry points to another descriptor entry which contains the actual information for the variable.

- Segment Address, Offset Address:
  The segment address and offset address are a pointer to the variable data in the user variable area. They are relative to the start of the variable area. The first byte of the segment address field contains the least significant 8 bits of the segment address. The offset address contains values in the range 00h through 0Fh. When the parameter bit is 1, the segment and offset addresses are the address of the variable descriptor entry for the parameter.

Intermediate Code

Codes interpreted by the HP-94 BASIC interpreter are called intermediate code.

Table 1-2. Intermediate Code

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0	eol		> =	LET	GET				SQR	FIX5	DMS					
x1			< =	GOTO	PUT				EXP	FIX9	ARD					
x2	bin		< >	GOSUB	PARAM				LOG	MAX	ADS					
x3	ral		>	RETURN	%CALL			TAB	LGT	MIN	FIXE					
x4	chr		<	FOR	DEF			XOR	SGN	RND	TIM					
x5	var		,	NEXT	READ			%CURSOR	ABS	EOF	PI					
x6	prm		;	IF	DATA			%HOME	INT	INPUT$	VER					
x7	fnc		:	ON	RESTORE			%DEL	LEN	TOD$	KEY					
x8	ext	(	#	DIM				AND	IDX	SIN	HEX$					
x9	lin	)		INPUT				OR	NUM	COS	SIZE					
xA	adr	+		PRINT				NOT	COD	TAN						
xB		-		CALL				TO	STR$	ASN						
xC	rem	**		END				STEP	CHR$	ACS						
xD		*		FORMAT				USING	ASC$	ATN						
xE		/		OPEN				MSG	MOD	FRC						
xF		=		CLOSE				SPACE	FIX0	RAD						

Note: A blank entry in the table indicates an unused code.

Table 1-3. Intermediate Code Groups

Code Group	Range of Codes
Operand	00h —— 17h
Delimiter	18h —— 2Fh
Statement	30h —— 47h
Optional Word	73h —— 7Fh
Function	80h —— A9h

Operand Codes

The symbols and formats for the various members of the operand code group are listed below.

Table 1-4. Operand Codes

Code	Symbol	Format	Comments
00h	eol	NUL	End of line
02h	bln	bln | VALUE (L), (H)	Integer constant
03h	ral	ral | real char string | NUL	Real type constant
04h	chr	chr | char string | NUL	Character constant
05h	var	var | ADRS (L), (H)	Variable
06h	prm	prm | ADRS (L), (H)	User-defined function parameter
07h	fnc	fnc | ADRS (L), (H) The ADRS after var, prm, and fnc is the appropriate position in the variable descriptor table	User-defined function
08h	ext	ext | external procedure name | NUL	Entry name (CALL and %CALL)
09h	lin	lin | ADDR (L), (H) lin is used by FORMAT, GOTO, GOSUB, and USING. The ADDR after lin is the relative offset to another line.	Text line address reference
0Ah	adr	adr | ADDR (L), (H) The ADDR after adr is the relative position from the start of the program.	Address of next DATA statement
0Ch	rem	rem | char string | NUL rem is used for the data in DATA statements or the format information in FORMAT statements.	Skipped during execution

Explanation of Operand Codes

The meaning of each operand code is as follows:

- eol (end of line)

  This indicates the end of a line in a program. Multiple statements within the line are separated with a colon (:), character code 27h.

- bin (integer constant)

  This indicates the following two bytes are an integer constant (-32768 through 32767). The first byte is the least significant 8 bits.

- ral (real constant)

  This indicates a real constant which is stored as a character string. Only positive numbers are stored in this format. Negative numbers are expressed as a unary expression.

  e.g. `-123.4` ⟶ - ral **1 2 3 . 4** NUL

- chr (character constant)

  This indicates a character string. It does not include the double quotation marks which specify the beginning and end of a character string. Two successive double quotation marks ("") indicate a double quote ("). Two successive ampersands (&&) indicate an ampersand (&). An ampersand (&) followed by two hexadecimal digits represents a single byte with that hexadecimal value.

- var (variable)

  The two bytes following var are the offset from the start of the variable descriptor table to the variable descriptor table entry. The first byte is the least significant 8 bits.

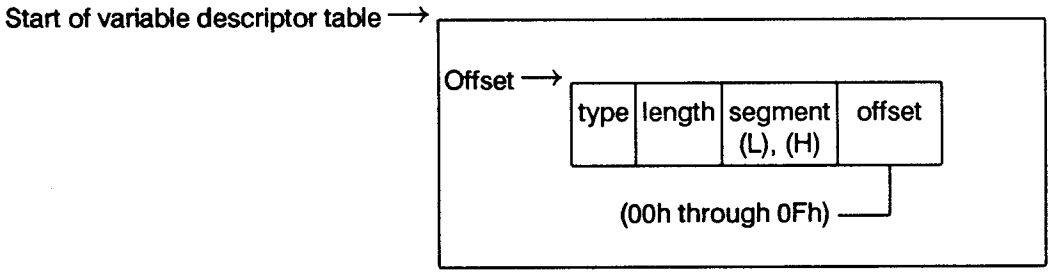


Figure 1-6. Variable Reference

- prm (user-defined function parameter name)

  The prm operand code is used for parameters in subprograms and user-defined functions.

  The two bytes following prm are the offset from the start of the variable descriptor table to the variable descriptor table entry. The first byte is the least significant 8 bits.

  The variable descriptor table entry has a type = 01h ('parameter') and length = 05h. The segment and offset values are relative to the start of the variable area. The variable area indicated by the variable descriptor table entry contains a variable descriptor table entry which has the correct type and length for the parameter. The segment and offset values in the latter entry are set to the actual address of the variable (*not* an offset from the start of the variable area).

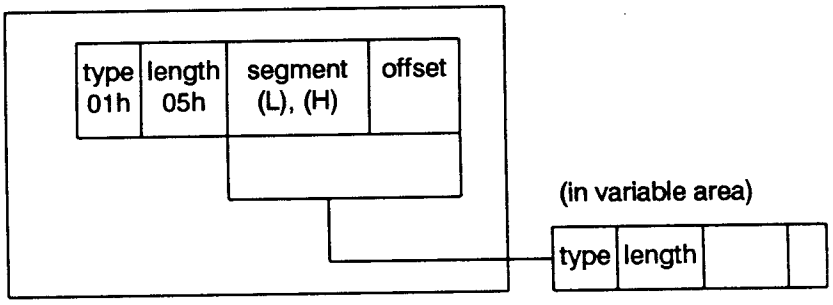


Figure 1-7. Parameters in the Variable Descriptor Table

- fnc (user-defined function)

  The two bytes following fnc are the offset from the start of the variable descriptor table to the variable descriptor table entry. The first byte is the least significant 8 bits.

  The variable descriptor table entry segment address field is the offset from the start of the program to the user-defined function definition. The final byte of this entry (offset) is 00h.

  If the definition contains one or more arguments, the segment address field points to the first argument. If the definition does not contain an argument, the segment address field points to the equals sign (=) which follows the definition.

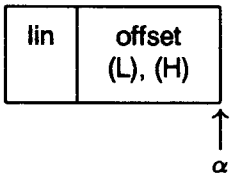
```
DEF FNA=
        ↑

DEF FNA(X,Y)=
        ↑
```

- ext (external program name)

  The subprogram name for CALL and %CALL is indicated with ext.

- lin (line reference)

  The two bytes following lin are an offset to the start of a line. The first byte is the least significant 8 bits. The offset is relative to the byte following the offset (the third byte following lin).



The start of the referenced line is (α) + offset

Figure 1-8. Line Reference

- adr (address reference)

  The adr operand code is used in DATA statements.

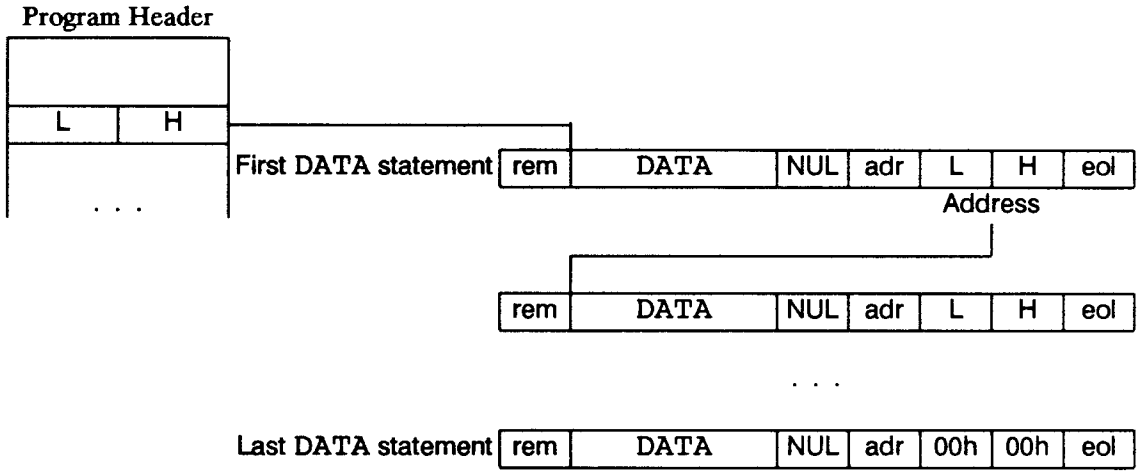


Figure 1-9. DATA Statement Linking

An adr of 0000h indicates the end of the DATA chain.

- rem (non-executed statement)

  The rem operand code indicates character strings for the FORMAT and DATA statements.

  A line with the rem operand code is not executed.

Variable Area

The variable area is allocated in main memory when BASIC program execution begins. It is released when execution ends.

The variable area is allocated or released as a block. The size of the variable area to be allocated is available in the variable area size field of the BASIC program header. The variable area is not allocated if the variable area size field is zero.

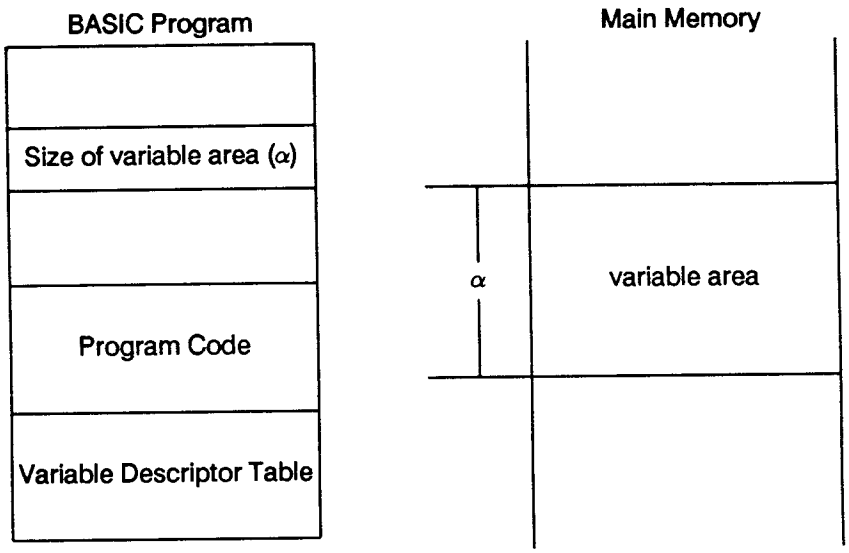


Figure 1-10. Variable Area Allocation

An example of the process of allocating and releasing variable areas is shown in the following figure. The example illustrates the main program (MAIN) calling a second program (program B), which in turn calls a third program (program C). Program C ends, returning control to program B. Program B also ends, returning control to MAIN. MAIN then ends, returning to command mode.

The control information save areas which are allocated between each variable area are omitted in this figure.

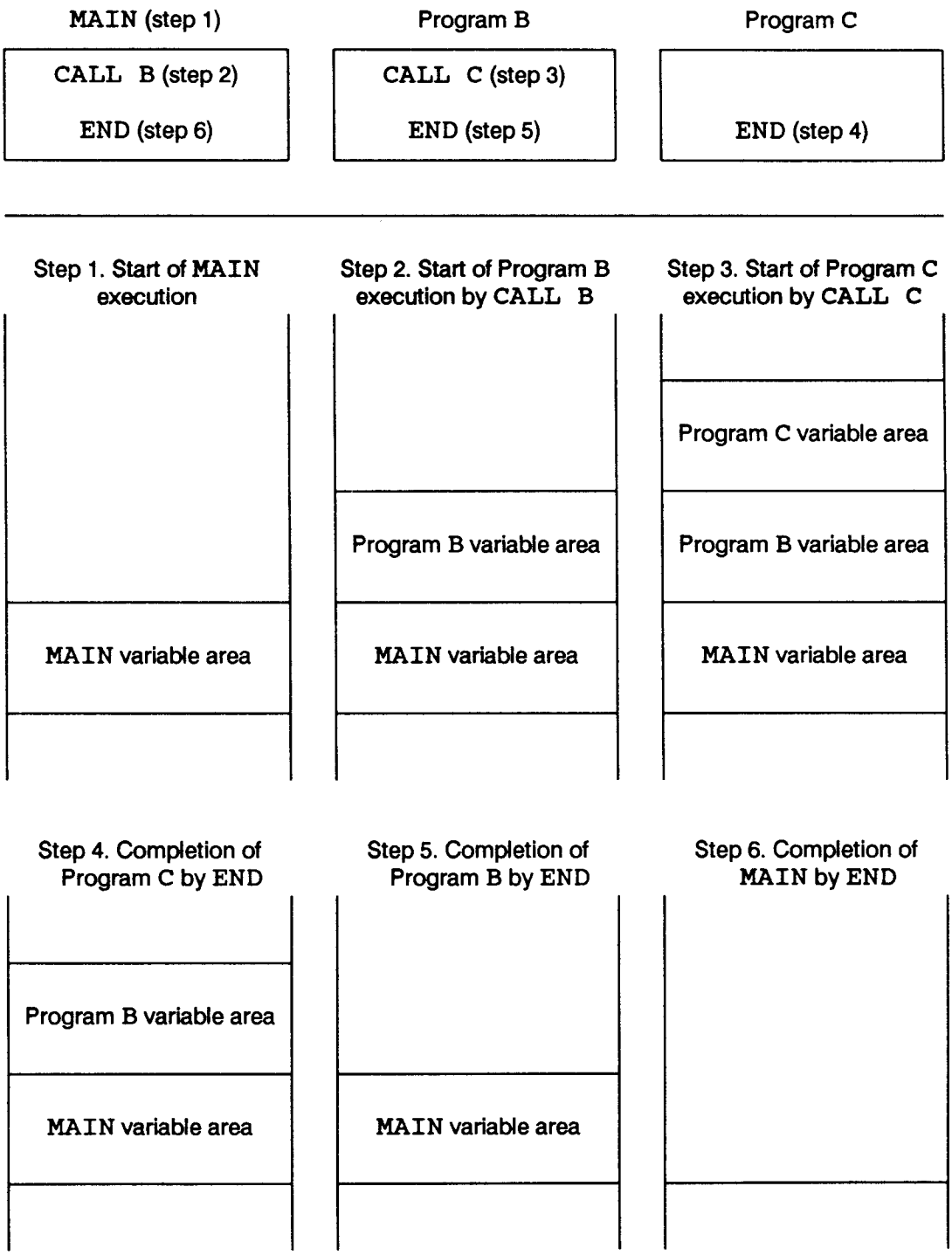


Figure 1-11. Allocating and Releasing Variable Areas

The relationship between the program code, variable descriptor table, and variable area is shown in the following figure.

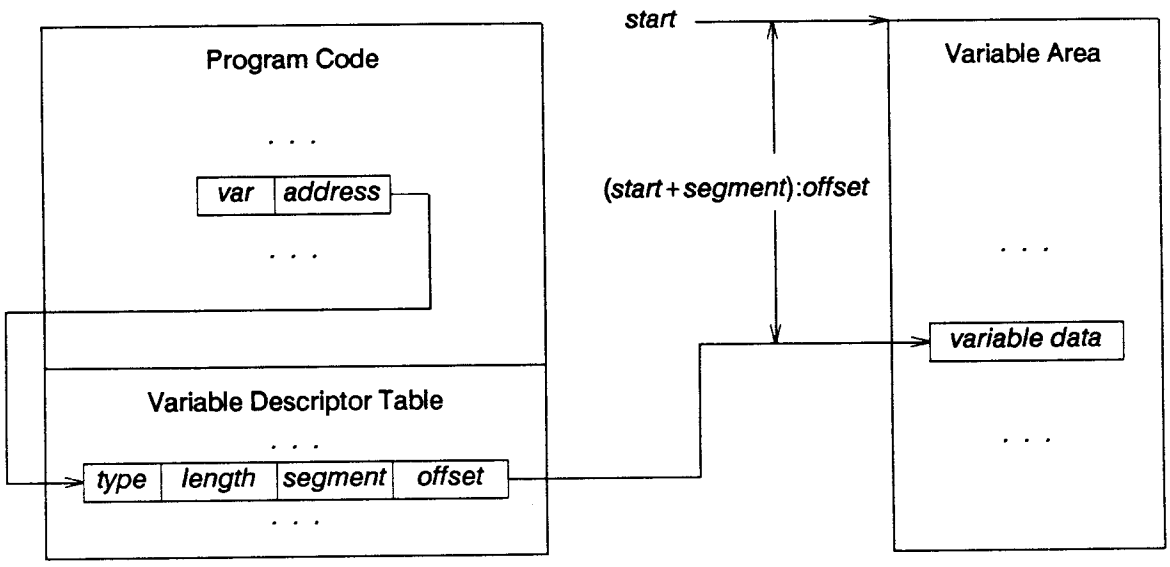


Figure 1-12. Program Code and Variables

The meaning of the items in italics is listed below.

- *var*
  Operand code for a variable
- *address*
  Relative address in the variable descriptor table
- *type*
  variable type
- *length*
  variable or array element length
- *segment*
  variable segment address relative to *start*
- *offset*
  variable offset address relative to *start*
- *start*
  Start of variable area (determined at CALL time)
- *variable data*
  Current value of the variable

An example showing several statements in a BASIC program helps clarify the relationship between program code and variables.

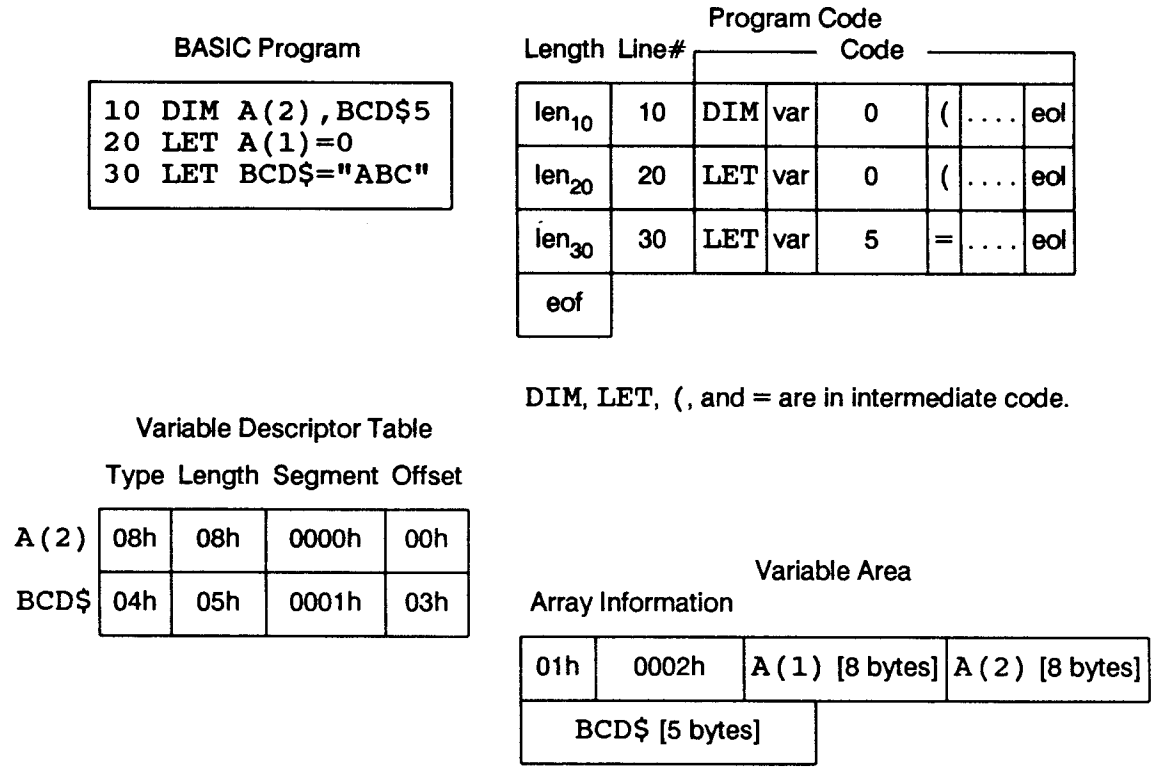


Figure 1-13. BASIC Program and Variable Relationships

Data Structure

There are three data types — real numeric data, integer numeric data, and character data. In addition, each of the data types can be collected into an array. Information about the array is stored preceding the elements in the array. The data in an array is stored consecutively.

Real Numeric Data

The format for real numeric data in the variable area is shown below.

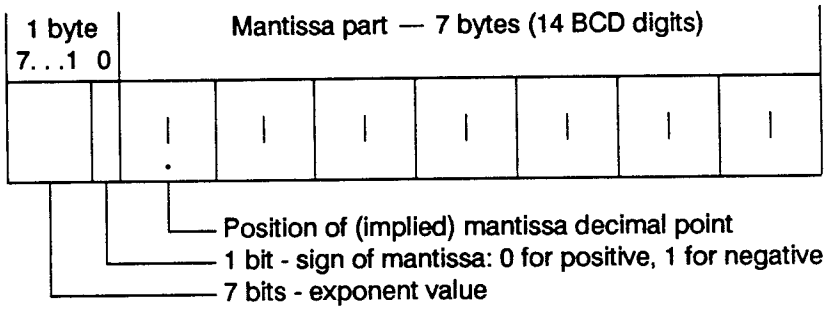


Figure 1-14. Real Numeric Data in the Variable Area

The exponent is in two's complement (binary). Exponent values -64 through 63 indicate 10^{-64} through 10^{63}.

Integer Numeric Data

Integers are stored as two bytes in both the variable area and data files; the first byte contains the most significant 8 bits, and the second byte contains the least significant 8 bits. The range of an integer is -32768 through 32767.

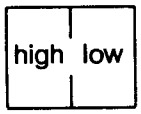


Figure 1-15. Integer Numeric Data in the Variable Area

Character Data

The format for character data in the variable area is shown below.

Figure 1-16. Character Data in the Variable Area

The default value for *n* is 8. A DIM statement can be used to assign values 1 through 255 to *n*. The value of *n* is in the variable descriptor table.

If the character string has fewer than *n* bytes, a NUL (00h) is stored following the last character of the string.

Only the first *n* characters assigned to a character string are stored — excess characters are discarded.

Array Data

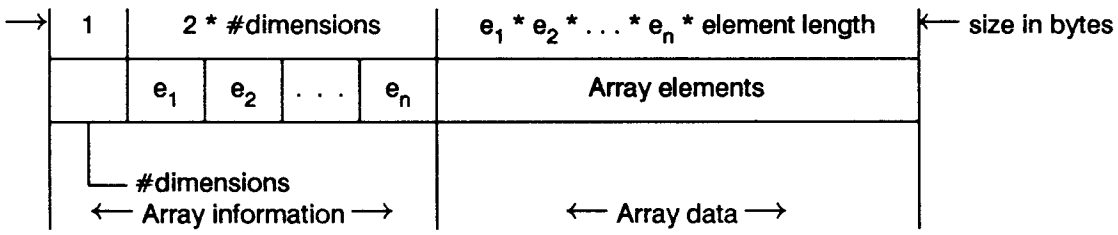


Figure 1-17. Array Data in the Variable Area

The maximum size of an array is 65535 (FFFFh) bytes, including both the array information and the array data. The number of dimensions must be in the range 1 through 255.

In the array information, e_n is the number of elements in that dimension. For `OPTION BASE 0`, the number of elements is the array's upper bound plus 1. Each e_n is stored with the least significant 8 bits in the first byte and the most significant 8 bits in the second byte.

Array elements are stored in row-major order (the right-most subscript varies most rapidly).

Array Examples

The following two examples show how the array information and data would be stored in memory.

Example: `DIM A(2,3)`

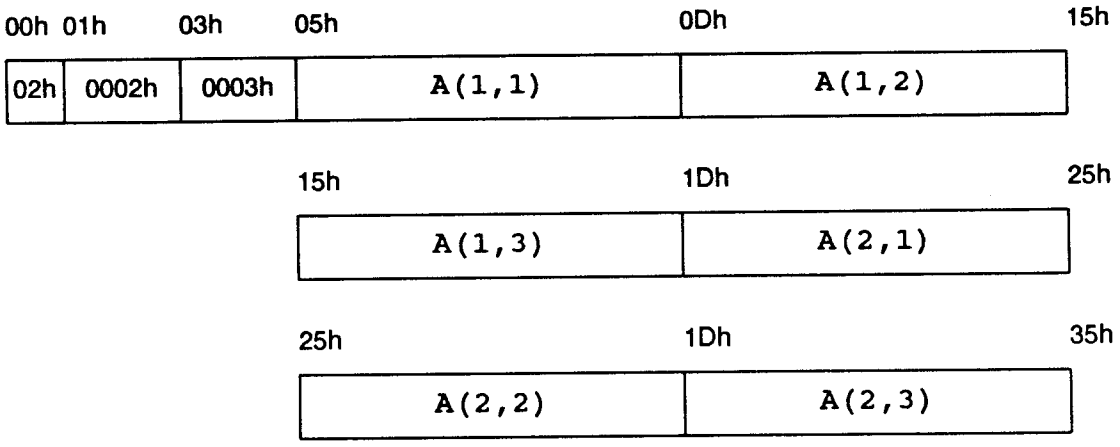


Figure 1-18. Array Data Example: DIM A(2,3)

Example: `OPTION BASE 0:DIM B$6(4)`

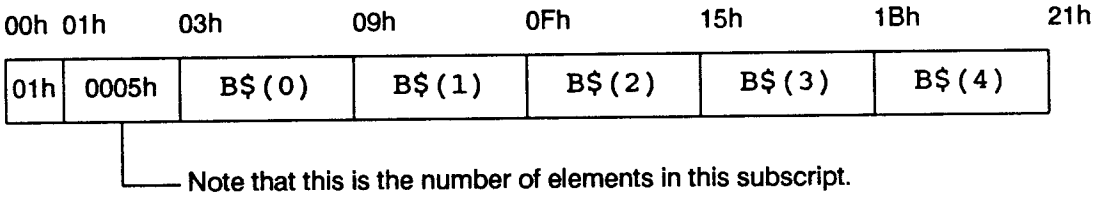


Figure 1-19. Array Data Example: OPTION BASE 0 : DIM B$6(4)

Control Information Save Area

The control information save area is used to save the control information of the currently executing program when a subprogram is called with the CALL statement or when an interrupt causes a jump to an interrupt routine.

The control information save area is allocated in main memory when a CALL statement or interrupt occurs. The control information for the currently executing program is saved in the save area. When the subprogram ends (END) or the interrupt routine ends (%CALL SYRT), the information is restored to the BASIC interpreter control area.

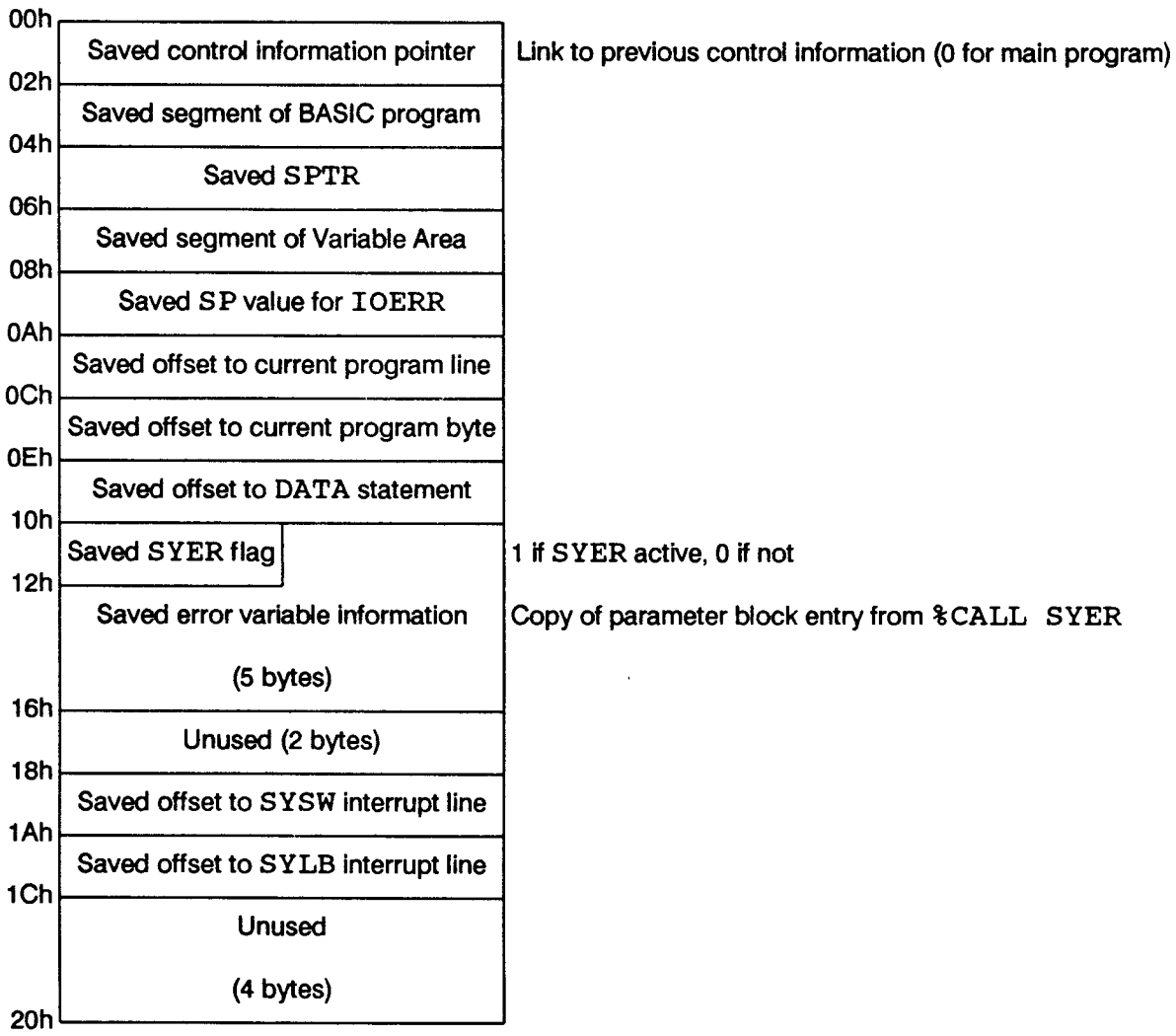


Figure 1-20. Format of the Control Information Save Area

2

Operation Stacks

Contents

Chapter 2 Operation Stacks

2

Operation Stacks

The operation stack area is used for:

- Control stack
- Numeric operation stack
- Character operation stack
- Parameter table entries (for %CALL)

Operation Stack Area

Parameters which are passed by value (constants and expressions) are evaluated, and the result of the expression is stored in the operation stack area.

The character operation stack pointer is CPTR, and the numeric operation stack pointer is SPTR.

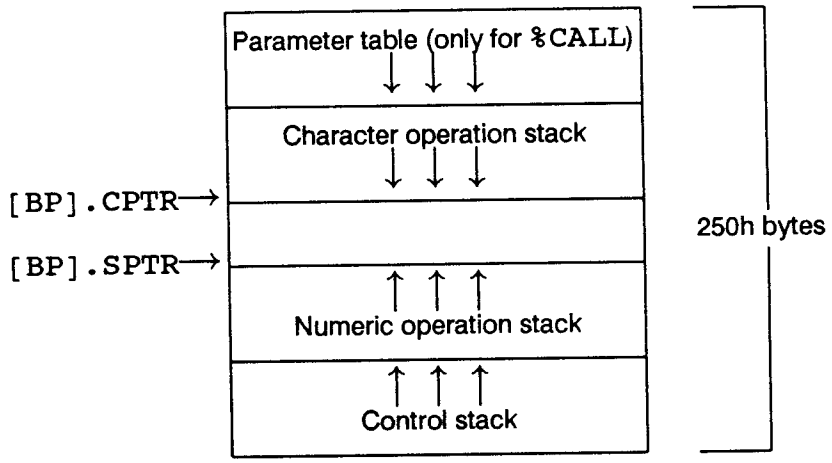


Figure 2-1. Operation Stack Area

Control Stack

The control stack is used to maintain address and variable information for GOSUB and FOR . . . NEXT loops.

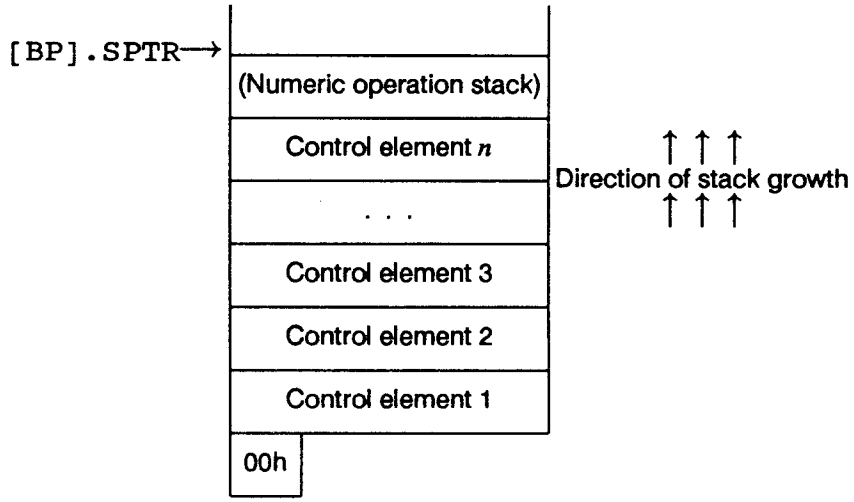


Figure 2-2. Control Stack Operation

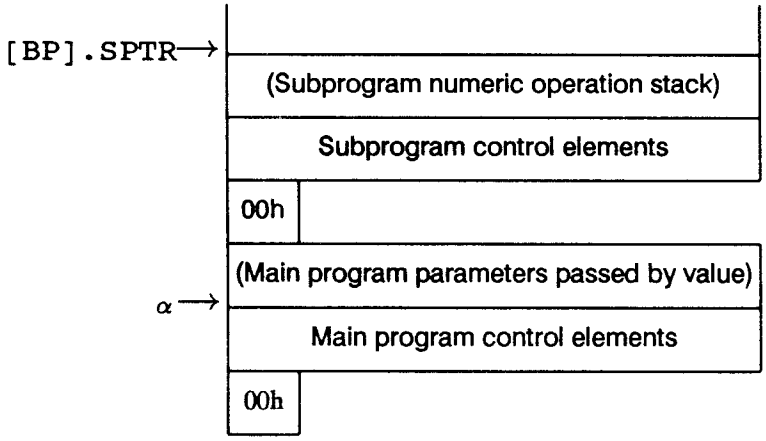


Figure 2-3. Control Stack During Subprogram Execution

Notes:

- α is the SPTR value saved in the control information save area.
- Control stack usage for an interrupt routine is the same as for a subprogram.
- Control elements consist of GOSUB return information and FOR . . . NEXT loop information.
- There is no pointer which separates the numeric operation stack from the control stack.

GOSUB Control Element

The GOSUB control element block size is 05h bytes.

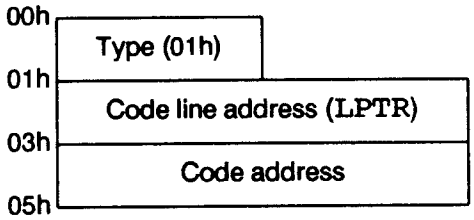


Figure 2-4. GOSUB Control Element

Code line address: The start of the code line containing the GOSUB statement.

Code address: The address of the eol or eos which follows the GOSUB statement.

FOR . . . NEXT Control Element

The FOR . . . NEXT control element block size is 18h bytes.

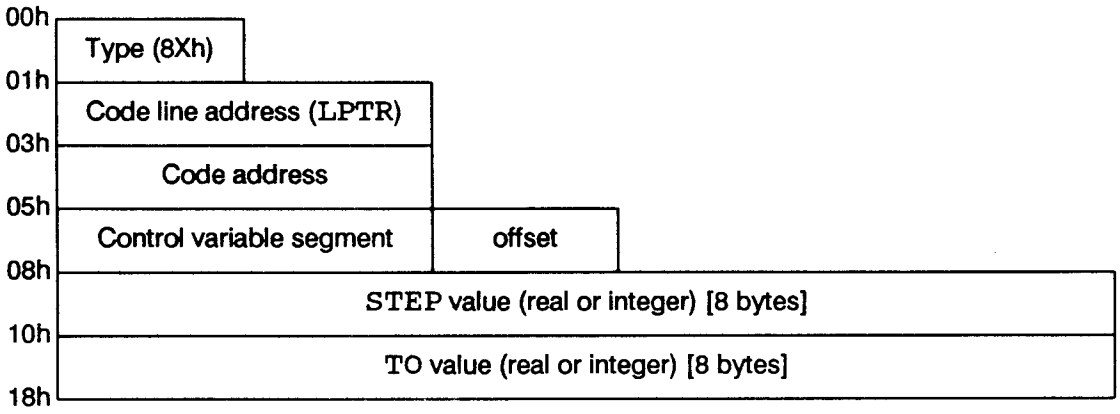


Figure 2-5. FOR . . . NEXT Control Element

Type (8Xh):	Indicates the control variable type (80h = real, 82h = integer).
Code line address:	The start of the code line containing the FOR statement.
Code address:	The address of the eol or eos which follows the FOR statement.
Control variable address:	The segment:offset address of the control variable for the FOR . . . NEXT loop. The offset is a single byte.
STEP value:	The value to be added to the control variable when the NEXT statement is executed. The type of the STEP value matches the type of the control variable (integer or real).
TO value:	The value to which the control variable is compared (after adding the STEP value) when the NEXT statement is executed. The type of the TO value matches the type of the control variable (integer or real).

The FOR . . . NEXT control element is removed from the control stack by the NEXT statement when the loop terminates. If the FOR loop is exited with a GOTO statement, the control element is left on the control stack. The FOR statement searches the control stack for FOR loop control elements before creating a new element. If there is a FOR loop control element with the same variable name, that control element is reused.

Numeric Operation Stack

Numeric parameters passed by value to subprograms are stored on the numeric operation stack (including any character values passed by value). The parameter table contains pointers to these values.

The SPTR and CPTR pointers are compared when pushing a value onto the stack. If SPTR ≤ CPTR, there is an overflow, and "Error MO" occurs.

Numeric values on the stack are always 8 bytes, whether real or integer type. An integer value on the stack starts with two bytes of 00h.

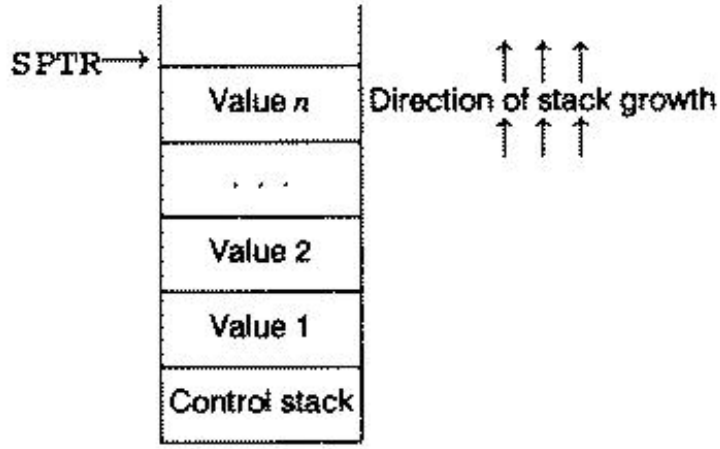


Figure 2-6. Numeric Operation Stack

Real Numeric Data

The format for real numeric data on the numeric operation stack is shown below.

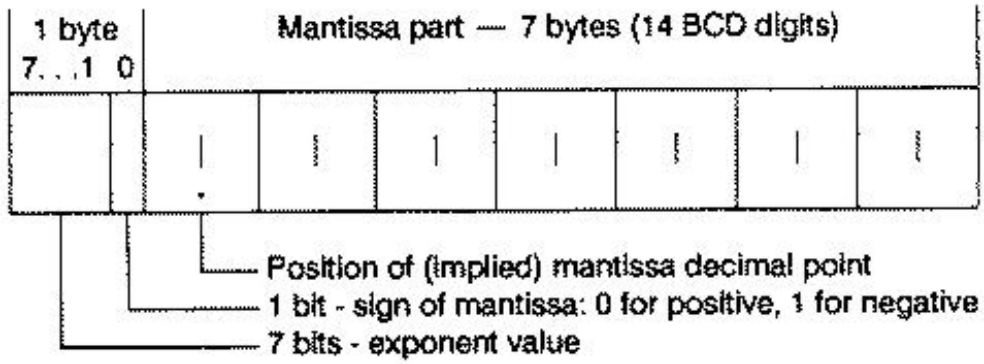


Figure 2-7. Real Numeric Data on the Numeric Operation Stack

The exponent is in two's complement (binary). Exponent values -64 through 63 indicate 10^{-64} through 10^{63}.

Integer Numeric Data

The range of an integer value is -32768 through 32767.

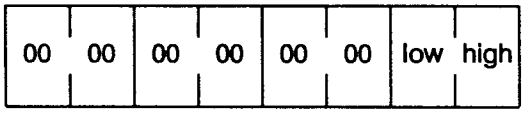


Figure 2-8. Integer Numeric Data on the Numeric Operation Stack

Numeric Operation Stack Example

A + B * C ⟶ D (S means SPTR)

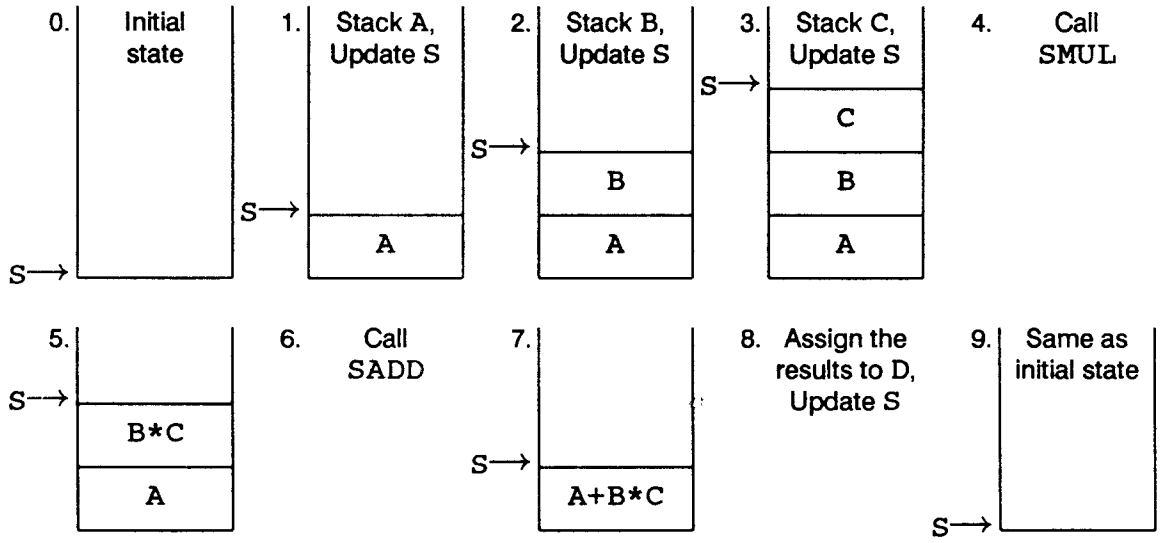


Figure 2-9. Numeric Operation Stack Example: A + B * C ⟶ D

Character Operation Stack

The character operation stack is used by character operators as a temporary storage area.

The SPTR and CPTR pointers are compared when pushing a value onto the stack. If CPTR ≥ SPTR, there is an overflow, and "Error MO" occurs.

A 00h byte must always be written at the byte pointed to by CPTR.

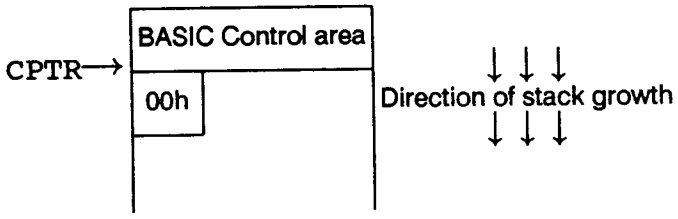


Figure 2-10. Character Operation Stack

Character Operation Stack Example

"ABC"+"DE"

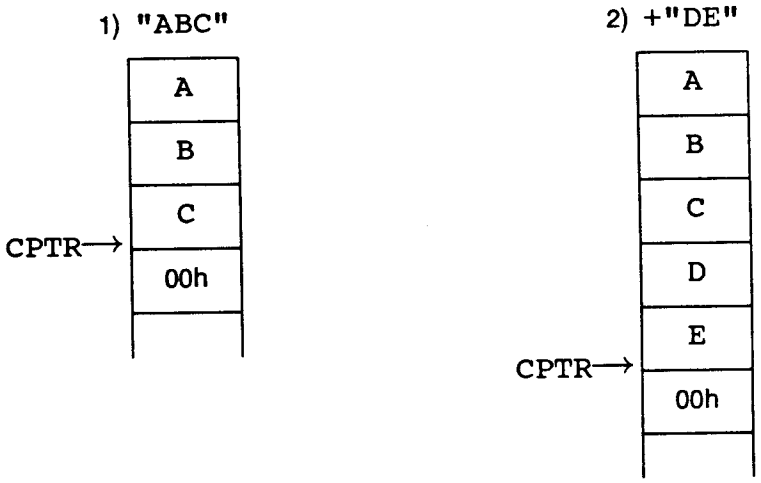


Figure 2-11. Character Operation Stack Example: "ABC" + "DE"

Parameter Table (only for %CALL)

The operation stack area is used by %CALL for the parameter table and for parameters passed by value.

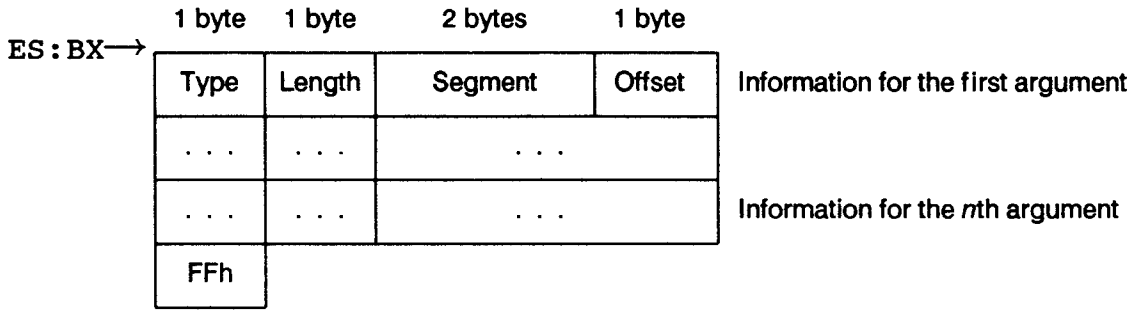


Figure 2-12. Parameter Table Format

The meanings of the fields in the parameter table are as follows:

- Type:

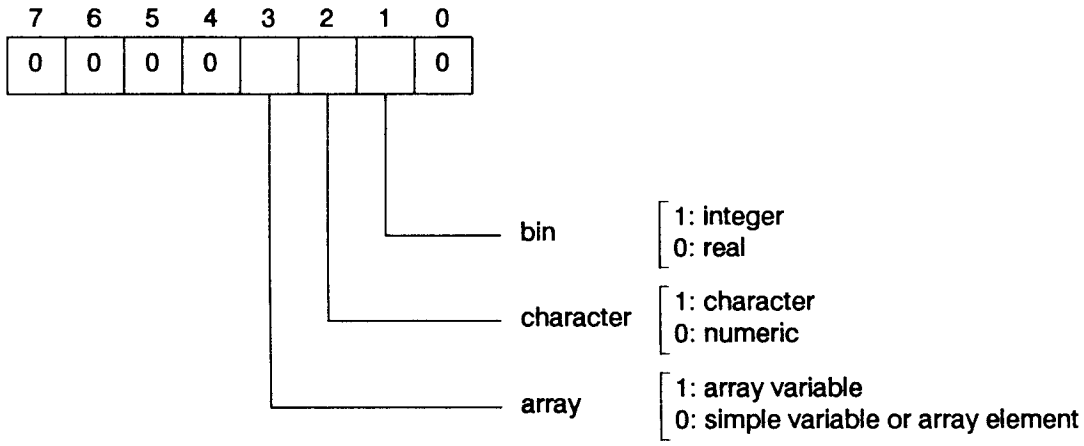


Figure 2-13. Parameter Table Type Byte

Arrays are passed to subprograms with subscript "*".

```
DIM XYZ(10)
  . . .
%CALL ABC(XYZ(*)) : REM pass the entire XYZ array
```

Numeric and string expressions (including constants) are evaluated by %CALL. Numeric values are put on the numeric operation stack as real numbers even if they could be expressed as an integer. String characters are moved from the character operation stack to the numeric operation stack before the subprogram is called.

- Length:

Type	Length in bytes
Integer	2
Real	8
Character	Dimensioned size (default is 8)
Array	Size of one array element

- Segment Address, Offset Address:

  The segment address and offset address contain the actual address of the variable's data area. This is different than in the variable descriptor table, where the address is relative to the start of the variable descriptor table.

  The segment address is a two-byte field; the offset address is a one-byte field with values 00h through 0Fh.

3

Assembly Language Subprograms (Keywords)

Contents

Chapter 3 Assembly Language Subprograms (Keywords)

3

Assembly Language Subprograms (Keywords)

An assembly language subprogram (also called a keyword) is called with the %CALL statement.

The following assembly language subprograms are built into the HP-94: SYAL, SYBP, SYEL, SYER, SYIN, SYLB, SYPO, SYPT, SYRS, SYRT, SYSW, and SYTO.

In addition, SYBD, SYBI, SYFT, and SYOS are reserved file names which must not be used for assembly language subprograms.

For assembly language subprograms which are not built into the HP-94, the file name is the subprogram name. In general, Hewlett-Packard uses SY as the first two characters of its assembly language files, and HN as the first two characters of its user-defined handlers. Names starting with SY and HN should not be used.

Assembly language subprograms must be written so that they can be executed in ROM.

This chapter assumes an understanding of HP-94 program structure. Refer to the "Program Execution" chapter in Part 1, "Operating System".

Program Structure

An assembly language program has a six-byte header followed by the program code. This structure is shown below with hex offsets indicated on the left side.

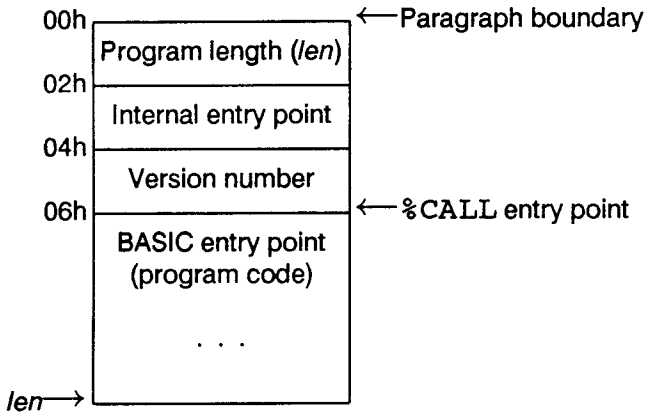


Figure 3-1. Assembly Language Subprogram Structure

See the "Program Execution" chapter in Part 1, "Operating System" for more information.

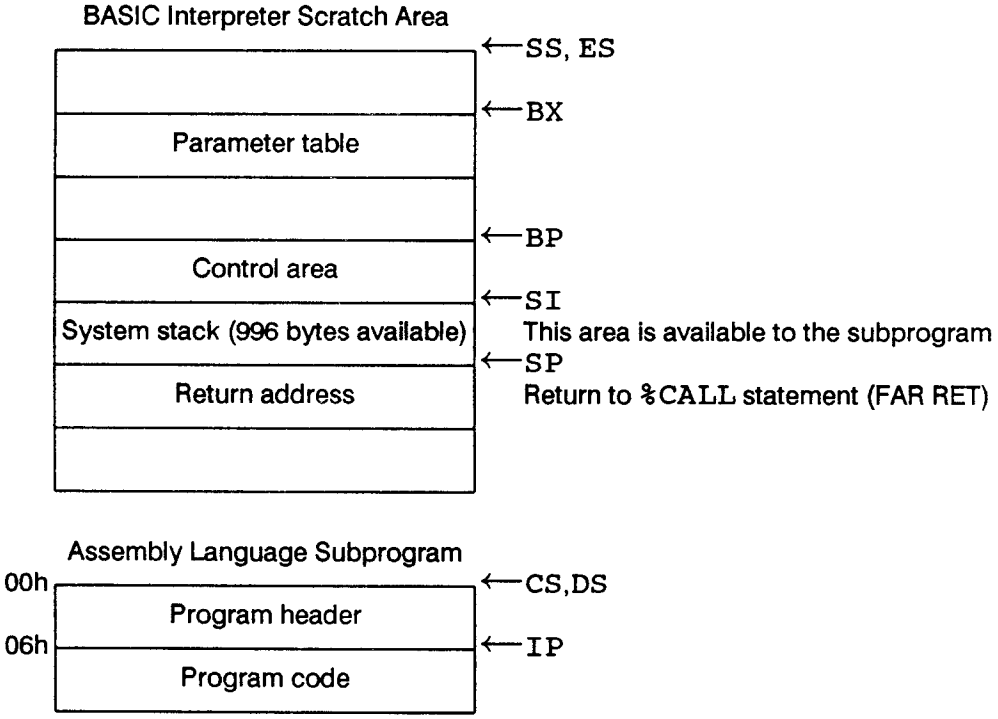
BASIC Call and Return

A BASIC program calls an assembly language program with the %CALL statement. When the assembly language routine finishes executing, a FAR RET is used to return to the BASIC interpreter.

BASIC Interpreter %CALL Procedure

The BASIC interpreter calls the assembly language subprogram at its entry point with a FAR CALL.

Contents of the CPU registers when an assembly language subprogram is called:



The direction flag is clear (CLD).

Interrupts are enabled (STI).

AX contains the value of SPTR before %CALL built the parameter table (not needed unless the subprogram uses IOERR; see IOERR for more information and an example).

The contents of registers which are not shown are not defined.

Parameter Table Format

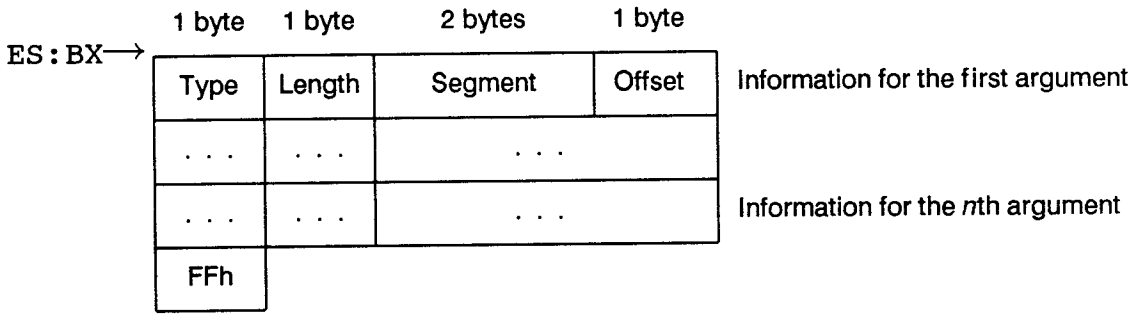


Figure 3-2. Parameter Table Format

The meanings of the fields in the parameter table are as follows:

- Type:

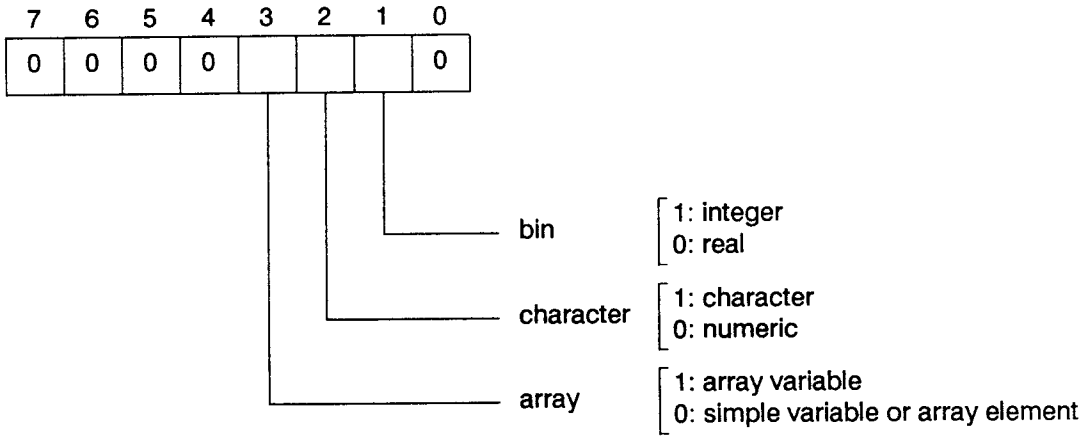


Figure 3-3. Parameter Table Type Byte

Arrays are passed to subprograms with subscript "*".

```
DIM XYZ(10)
  . . .
%CALL ABC(XYZ(*)) : REM pass the entire XYZ array
```

Numeric and string expressions (including constants) are evaluated by %CALL. Numeric values are put on the numeric operation stack as real numbers even if they could be expressed as an integer. String characters are moved from the character operation stack to the numeric operation stack before the subprogram is called.

- Length:

Type	Length in bytes
Integer	2
Real	8
Character	Dimensioned size (default is 8)
Array	Size of one array element

- Segment Address, Offset Address:

  The segment address and offset address contain the actual address of the variable's data area. This is different than in the variable descriptor table, where the address is relative to the start of the variable descriptor table.

  The segment address is a two-byte field; the offset address is a one-byte field with values 00h through 0Fh.

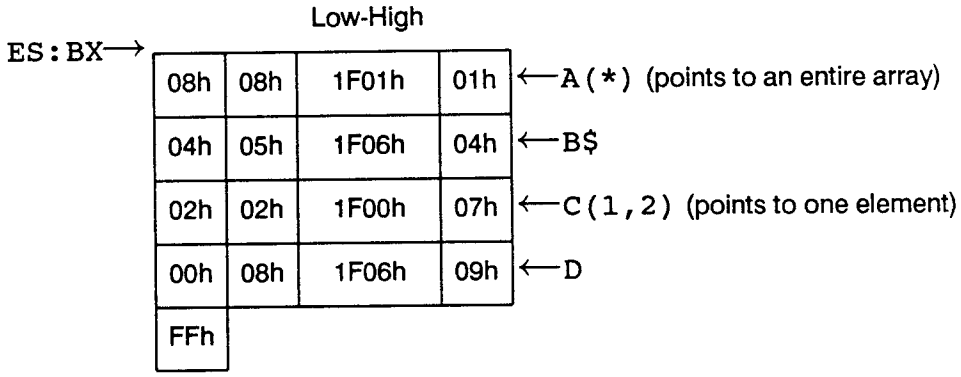
%CALL Example

```
 10 INTEGER C
 20 DIM A(10),B$5,C(3,2)
 30 D=1
    . . .
100 %CALL AB(A(*),B$,C(1,2),D)
```

When line 100 is executed, %CALL creates a parameter table (shown below) in the operation stack area and passes a pointer to it in ES : BX.

Assume that the BASIC variable area segment address is 1F00h.

Parameter Table



Variable Area

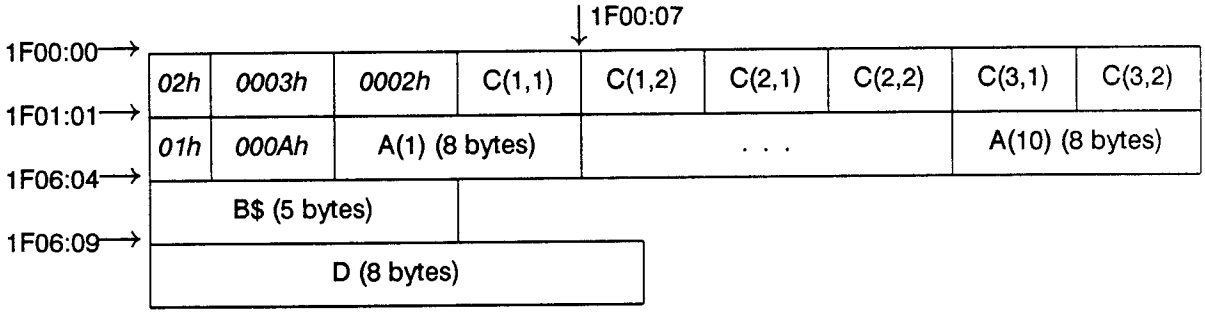


Figure 3-4. %CALL Example: Calling an Assembly Language Subprogram

Note: The values in *italics* are array information.

Assembly Language Subprogram Return to BASIC

When an assembly language subprogram returns to the BASIC program that called it, the following conditions should exist.

- The SS, BP, and SP registers must have the same value as when the assembly language subprogram was called.
- The direction flag must be clear (CLD instruction).
- Interrupts must be enabled (STI instruction).
- A FAR RET must be used to return to the BASIC Interpreter.

Access to BASIC Interpreter Utility Routines

This section describes how to access BASIC Interpreter utility routines for decimal math, stack manipulation, number conversion, and parameter processing from an assembly language program.

In the following table, CSEG is the segment address of the BASIC interpreter.

An assembly language subprogram can easily determine the value of CSEG by examining the return stack. The word at SS:SP+2 is the segment address of the BASIC interpreter.

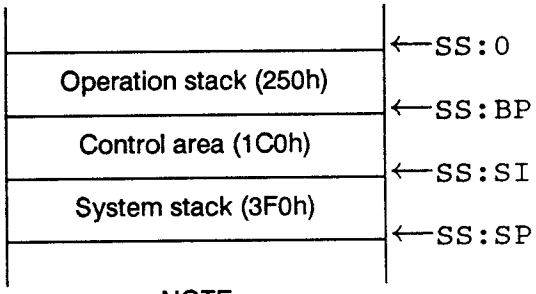
The regular entry point of the BASIC interpreter is CSEG:0. If the interpreter is called at CSEG:6 (as the operating system S command does), it immediately returns to the operating system.

If an error is detected by a BASIC interpreter utility routine, either the ERROR routine or the IOERR routine is called. The line number and the program name displayed in the error message point to the %CALL keyword.

BASIC Interpreter Code

CSEG:00h	JMP CSEG:51h (interpreter start)	
02h	Identifier "IP"	
04h	Release No.	Version No.
06h	JMP CSEG:44h (Exit to O.S.)	
08h	Data part size (paragraphs)	
0Ah	Operation stack size (bytes)	
0Ch	Control area size (bytes)	
0Eh	offset from SS:BP to SPTR	
10h	offset from SS:BP to CPTR	
12h	offset from SS:BP to SYSSTK	
14h	JMP SADD	(FAR RET)
18h	JMP SSUB	(FAR RET)
1Ch	JMP SMUL	(FAR RET)
20h	JMP SDIV	(FAR RET)
24h	JMP SPOW	(FAR RET)
28h	JMP SNEG	(FAR RET)
2Ch	JMP TOREAL	(FAR RET)
30h	JMP TOBIN	(FAR RET)
34h	JMP ERROR	(FAR RET)
38h	JMP IOERR	(FAR RET)
3Ch	JMP GETARG	(FAR RET)
40h	JMP SETARG	(FAR RET)
44h	EXIT (returns to the operating system)	
51h	BASIC Interpreter code	

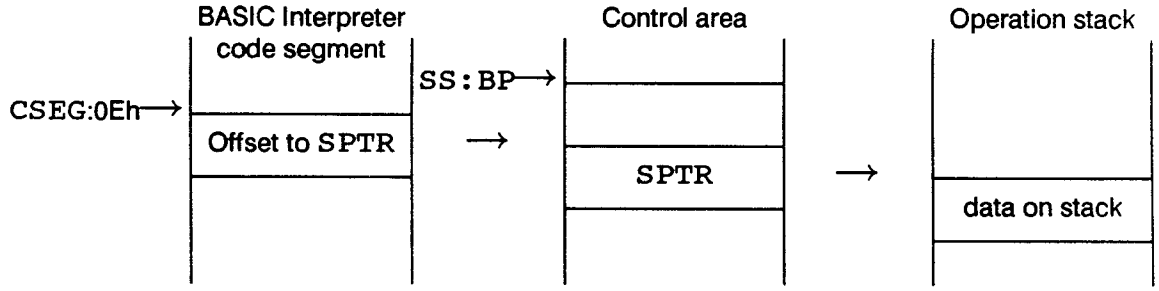
BASIC Interpreter Scratch Area



NOTE
The scratch area is allocated in main memory by the BASIC interpreter after a cold start.

Using a Utility from an Assembly Language Subprogram

Many of the utility routines require their data to be on the numeric operation stack. The numeric operation stack pointer (SPTR) must be set up to use these routines.



The SPTR address relative to BP is stored in the BASIC interpreter header at location CSEG:0Eh.

See the "Operation Stack" section for more information about using the numeric operation stack.

4

BASIC Interpreter Utility Routines

Contents

Chapter 4 BASIC Interpreter Utility Routines

4

BASIC Interpreter Utility Routines

This chapter describes the BASIC interpreter utility routines. These utilities allow assembly language subprograms to use the decimal math routines in the BASIC interpreter and simplify the passing of parameters between BASIC programs and assembly language subprograms. Utility routines are also available for reporting errors detected in the assembly language subprograms and for converting between real and integer data.

BASIC Interpreter Utility Routine Descriptions

BASIC interpreter utility routine descriptions consist of the following:

- A brief description of the routine.
- The calling sequence for the routine.
- Notes on the use and behavior of the routine.
- A summary of the parameters passed to the routine and the parameters that the routine must return.

Registers Passed to BASIC Interpreter Utility Routines

The BASIC interpreter utility routines all expect BP to point to the BASIC interpreter control area. Other registers which are expected are mentioned in the "Input:" section for each routine.

ERROR

Display an error message and return to the operating system command mode.

Calling sequence:

```
FAR CALL CSEG:34h
```

Notes:

- If the code in AL is not in the table below, the code is displayed as three decimal digits.
- ERROR returns to command mode after displaying the message.

Input:

AL = code
(See table below)

Output:

```
Error CC nnnnn pppp
```
CC: Characters corresponding to the code
nnnnn: Error line number
pppp: Program name

Table 4-1. Codes for ERROR Utility Routine

Hex	Decimal	CC	Meaning
01h	1	SY	Syntax error
02h	2	TY	Data type mismatch
03h	3	CN	Conversion error
04h	4	RT	RETURN or SYRT error
05h	5	DT	Data error
06h	6	IL	Illegal argument
07h	7	BR	Branch destination error
08h	8	MO	Memory overflow
09h	9	NF	Program not found
0Ah	10	AR	Array subscript error
0Bh	11	CO	Conversion overflow
0Ch	12	EP	Missing END statement
0Dh	13	DO	Decimal overflow
0Eh	14	IR	Insufficient RAM
0Fh	15	FN	Illegal DEF FN statement
10h	16	UM	Unmatched number of arguments
11h	17	BM	BASIC interpreter malfunction
12h	18	LN	Nonexistent line
13h	19	IS	Illegal statement

GETARG

Convert a numeric parameter from %CALL into a binary value and return the value.

Calling sequence:

```
FAR CALL CSEG:3Ch
```

Notes:

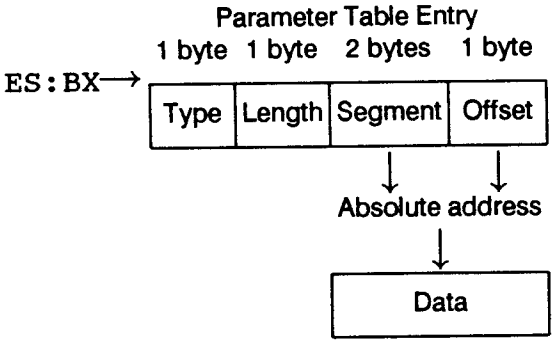
- If the parameter is an array or is of character type, `Error TY` occurs.
- If there is no parameter (type = FFh), `Error UM` occurs.
- If the parameter is negative and a negative number is not allowed, `Error IL` occurs.
- If the parameter is out of range, `Error IL` occurs. The valid range depends on the contents of register CL, as shown in this table:

Table 4-2. GETARG Result Flag (Register CL)

CL	Length	Positive/Negative	Range of values
0	word (16 bits)	positive or zero only	0 through 32767
1	double word (32 bits)	positive or zero only	0 through 2^{31}-1
2	word	negative allowed	-32768 through 32767
3	double word	negative allowed	-2^{31} through 2^{31}-1

Input:

ES:BX points to a parameter table entry
CL is the flag byte (see below)



Output:

If destination is a word:
AX = binary value
DX = (undefined)
If destination is double word:
AX = low word of binary value
DX = high word of binary value

Figure 4-1. GETARG Parameter Processing

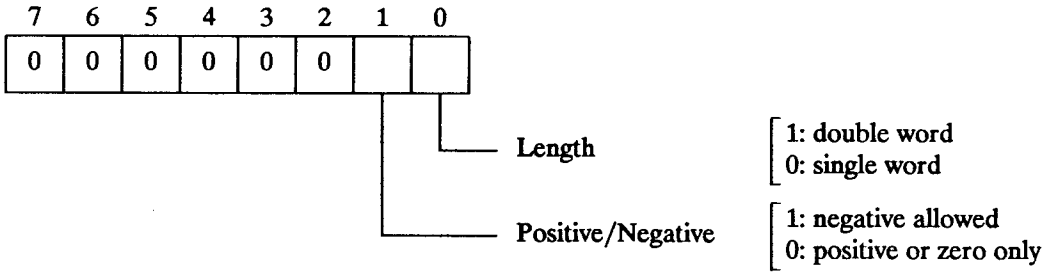


Figure 4-2. GETARG Result Flags (Register CL)

If error trapping (%CALL SYER) is not in effect, display an error message and return to the operating system command mode.

Calling sequence:

```
FAR CALL CSEG:38h
```

Notes:

- Assembly language subprograms must set up certain registers before calling IOERR. See the example program below to set up these registers.
- If error trapping (%CALL SYER) is in effect, the error number variable is set to the error code. BASIC execution resumes at the next line (not statement) of the BASIC program. IOERR does *not* return to the assembly language routine which called it.
- If error trapping (%CALL SYER) is not in effect, a call to IOERR has the same effect as a call to ERROR.

Input:

AL = code
(See Appendix B)
BP, SS, and SP unchanged from %CALL
SPTR restored (value was in AX after %CALL)

Output:

```
Error NNN nnnnn pppp
```

NNN: Error code (3 decimal digits)
nnnnn: Error line number
pppp: Program name

```
IOERR_OFFSET            equ     038h
SPTR_OFFSET             equ     0Eh

MYSEG                   segment public 'MYSEG'
                        assume cs:MYSEG
EXAMPLE                 proc    far
START:
PROG_SIZE               dw      FINISH-START
ASM_ENTRY_ADR           dw      offset START_ASM
VERSION                 dw      0100h           ; Version 1.00
;
; This is an outline of an assembly-language subprogram which shows how
; to save and restore the value of SPTR before a call to IOERR.
;
START_ASM:
                        push    ax              ; Save SPTR value on stack
;
;******* (user's code omitted here) *******
;
JMP_IOERR:                      ; AL contains the error code for IOERR
                        cld                     ; Needed only if code executed a STD
                        sti                     ; Needed only if code executed a CLI
                        pop     dx              ; Recall SPTR value to DX from stack
                        pop     cx              ; Drop BASIC interpreter offset
                        pop     es              ; Pop BASIC interpreter segment (CSEG)
                        push    es              ; Push CSEG for IOERR entry
                        mov     cx,IOERR_OFFSET
                        push    cx              ; Push offset for IOERR entry
                        mov     si,es:SPTR_OFFSET ; SI = offset of SPTR
```

```
                        mov     ss:[bp+si],dx   ; Restore SPTR
                        ret                     ; Jump to IOERR
;
NORMAL_RETURN:
                        pop     dx              ; Throw away (unused) SPTR value
                        ret                     ; Return to BASIC interpreter
;
EXAMPLE                 endp
FINISH:
MYSEG                   ends
                        end
```

SADD

Add two numbers on the operation stack.

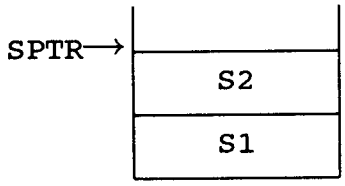
Calling sequence:

```
FAR CALL CSEG:14h
```

Notes:

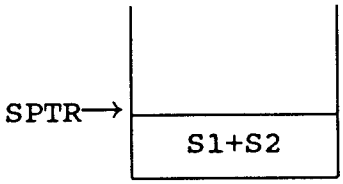
- The numbers can be either real numbers or integers. The result is an integer only if both numbers were integers, and the result fits in an integer.
- SADD does not use the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



S1+S2: numeric value

SDIV

Divide two numbers on the operation stack.

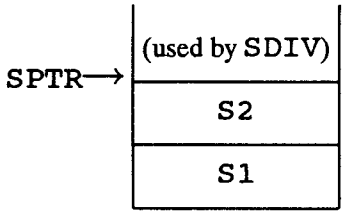
Calling sequence:

```
FAR CALL CSEG:20h
```

Notes:

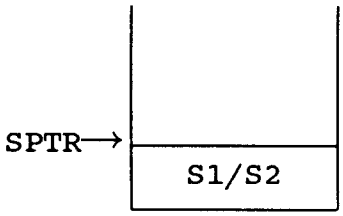
- The numbers can be either real numbers or integers. The result is always a real number.
- SDIV uses the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



S1/S2: numeric real value

SETARG

SETARG converts a binary value into the type of a numeric parameter from %CALL (either real or integer) and stores the value into the parameter.

Calling sequence:

```
FAR CALL CSEG:40h
```

Notes:

- AX contains the binary value (-32768 through 32767).
- If there is no parameter (type = FFh), `Error UM` occurs.
- If the parameter is an array or is of character type, `Error TY` occurs.
- SETARG uses 8 bytes of the operation stack as a scratch area.

Input:

AX is a binary value
ES : BX points to a parameter table entry

Output:

Contents of AX placed in parameter.

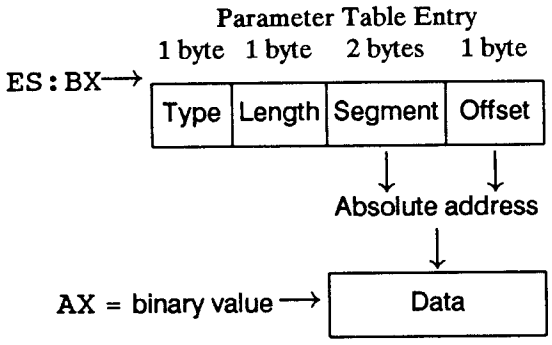


Figure 4-3. SETARG Parameter Processing

SMUL

Multiply two numbers on the operation stack.

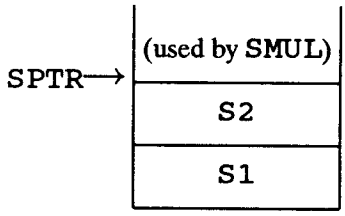
Calling sequence:

```
FAR CALL CSEG:1Ch
```

Notes:

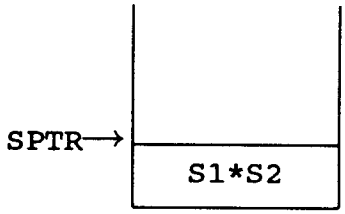
- The numbers can be either real numbers or integers. The result is an integer only if both numbers were integers, and the result fits in an integer.
- SMUL uses the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



S1*S2: numeric value

SNEG

Change the sign of a number on the operation stack.

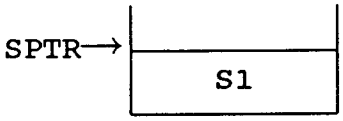
Calling sequence:

```
FAR CALL CSEG:28h
```

Notes:

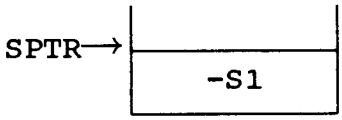
- The number can be either a real number or an integer. The result is an integer if the number was an integer.
- SNEG does not use the operation stack as a scratch area.

Input:



S1: numeric values

Output:



-S1: numeric value

SPOW

Exponential operation for two numbers on the operation stack.

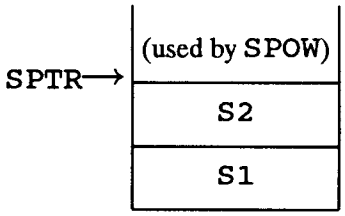
Calling sequence:

```
FAR CALL CSEG:24h
```

Notes:

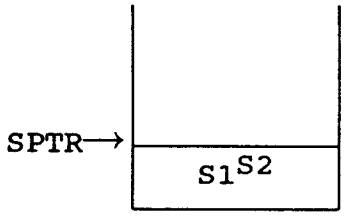
- The numbers can be either real numbers or integers. The result is always a real number.
- SPOW uses the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



$S1^{S2}$: numeric real value

Subtract two numbers on the operation stack.

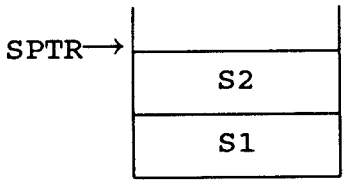
Calling sequence:

```
FAR CALL CSEG:18h
```

Notes:

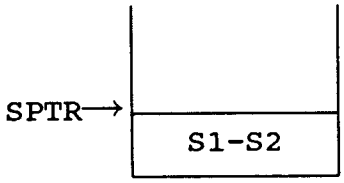
- The numbers can be either real numbers or integers. The result is an integer only if both numbers were integers, and the result fits in an integer.
- SSUB does not use the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



S1-S2: numeric value

TOBIN

Convert a number at SS : BX to an integer.

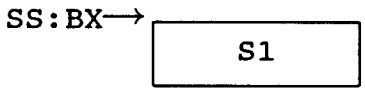
Calling sequence:

```
FAR CALL CSEG:30h
```

Notes:

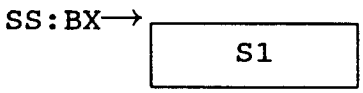
- The value is left unchanged if SS : BX points to an integer.
- The fractional part of the real number, if any, is truncated.
- An error occurs if the real number is not within the range -32768 through 32767.
- TOBIN does not use the operation stack as a scratch area.

Input:



S1: numeric real or integer data

Output:



S1: numeric integer data

TOREAL

Convert an integer or real number at SS:BX to a real number.

Calling sequence:

```
FAR CALL CSEG:2Ch
```

Notes:

- TOREAL does not use the operation stack as a scratch area.

Input:

SS:BX→ [S1]

S1: numeric real or integer data

Output:

SS:BX→ [S1]

S1: numeric real data

5

I/O Statement and Handlers

Contents

5

I/O Statements and Handlers

The *BASIC Reference Manual* has tables associated with the BASIC I/O keywords (GET #, INPUT #, INPUT$, PRINT #, PRINT # ... USING, and PUT #) which describe the interaction between the keywords and the built-in handlers for channels 1 through 4. This chapter describes the interactions between these BASIC keywords and user-defined handlers for channels 1 through 4.

Input Keywords (GET #, INPUT #, INPUT$)

GET #, INPUT #, and INPUT$ all process incoming data in a different way.

- GET # reads data directly into the input variables.
- INPUT # reads data into a 256-byte internal buffer, then copies the data to the input variables.
- INPUT$ reads data and places it on the character stack. The data is then copied to the variable with the BASIC assignment operation.

The following table summarizes how each of the input keywords responds to conditions generated by user-defined handlers.

Table 5-1. Response of Input Keywords to Handler-Generated Errors

Condition	GET #	INPUT #	INPUT$
CR LF received.	N/A.	Characters received from the device (except the CR LF) are placed in the input variable. Input is aborted if no other characters were received.	N/A.
Character from terminate character string received.	N/A.	N/A.	Characters received from the device (including the terminate character) are placed in the input variable.
Short record detected (error 115).	Ends input for that variable.	The short read error generates a garbage byte, and the error is ignored (input operation for that variable not ended).	Ignored (input operation not ended).
Terminate character detected (error 116).	Ends input for that variable.	Ignored (input operation for that variable not ended).	Ignored (input operation not ended).
End of data (error 117).	Ends input for that variable. *	Characters read up to the EOD are placed in the input variable. Input is aborted if no characters were received before the EOD.	Ignored (input operation not ended). *
Timeout (error 118).	Input aborted.	Input aborted.	Input aborted.
Power switch pressed (error 119).	Input aborted.	Input aborted.	Input aborted.
Low battery (error 200).	Input aborted.	Input aborted.	Input aborted.
Errors 201-208.	Input aborted.	Input aborted.	Input aborted.

* The behavior of GET # and INPUT$ is altered if INPUT # has been used with the channel and the last INPUT # aborted input due to an EOD.

- GET # is not affected except that program execution continues on the next line of the program (not the next statement, if GET # is in a multistatement line).
- INPUT$ will abort input after reading one character. Program execution continues on the next line of the program (not the next statement, if INPUT$ is in a multistatement line).

When input is aborted, program execution continues on the next line of the program (not on the next statement, if the GET #, INPUT #, or INPUT$ statement is in a multistatement line).

"Input aborted" has different meanings for GET #, INPUT #, and INPUT$.

GET #: The input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are placed in the input variable. This may result in part of the previous value of the variable being overwritten. All subsequent variables in the input list are unchanged. This is in contrast to INPUT # and INPUT$, in which any received data for that variable is discarded.

When input is aborted because of a numeric error, the I/O length reported by SYIN is set to the number of bytes actually received up to that point, since that data has already been placed in the input variable.

INPUT #: No data has been received or the input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are discarded. The current input variable and all subsequent variables in the input list are left unchanged (note that variables prior to the one at which input was aborted will already have been changed). This is in contrast to GET #, in which any received data for that variable is saved.

When input is aborted because of a numeric error, the I/O length reported by SYIN is set to 0, since no data is placed in the input variable.

INPUT$: No data has been received or the input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are discarded. The input variable is left unchanged. This is in contrast to GET #, in which any received data is saved and the variables are set to 0 or the null string.

When input is aborted because of a numeric error, the I/O length reported by SYIN is set to 0, since no data is placed in the input variable.

Output Keywords (PRINT #, PRINT # . . . USING, PUT #)

This table summarizes how each of the output keywords (PRINT #, PRINT # ... USING, PUT #) responds to errors generated by user-defined handlers.

Table 5-2. Response of Output Keywords to Handler-Generated Errors

Condition	PRINT #	PRINT # . . . USING	PUT #
Timeout (error 118).	Output aborted.	Output aborted.	Output aborted.
Power switch pressed (error 119).	Output aborted.	Output aborted.	Output aborted.
Low battery (error 200).	Output aborted.	Output aborted.	Output aborted.
Errors 201-208.	Output aborted.	Output aborted.	Output aborted.
Lost connection while transmitting (error 218).	Output aborted.	Output aborted.	Output aborted.

"Output aborted" means that the output operation has been interrupted. When output is aborted, the output operation is ended. Subsequent variables in the output list are not output.

When output is aborted, program execution continues on the next line of the program (not on the next statement, if PRINT #, PRINT # ... USING, or PUT # is in a multistatement line).

When output is aborted because of a numeric error, the I/O length reported by SYIN is set to the number of bytes actually sent up to that point, since that data has already been written to the device.