Hewlett-Packard -- Portable Computer Division

Research and Development Laboratory

Corvallis, Oregon


```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                      %
%            HP-71 Software            %
%                                      %
%     Internal Design Specification    %
%                                      %
%                                      %
%              VOLUME   I              %
%                                      %
%      Detailed Design Description     %
%                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%        %%%%%%%%%               %%%%
%%%%%%%%        %%%%%%%%%%%           %%%%%%%%
   %%           %%         %%%      %%%      %%%
   %%           %%          %%      %%
   %%           %%          %%      %%%
   %%           %%          %%        %%%%%%
   %%           %%          %%          %%%%%%
   %%           %%          %%               %%%
   %%           %%          %%                %%
   %%           %%         %%%      %%%      %%%
%%%%%%%%        %%%%%%%%%%%           %%%%%%%%
%%%%%%%%        %%%%%%%%%%%             %%%%


%%      %%    %%%%     %%                  %%%%%%
 %%    %%    %%  %%    %%                    %%
  %% %%      %    %    %%                    %%
   %%%       %%  %%    %%          %%%       %%
    %         %%%%     %%%%%%      %%%     %%%%%%
```

ROM Release 1BBBB -- December 1983

00071-90068

****** NOTICE ******

Table of Contents

OVERVIEW — CHAPTER 1

The HP-71 is an advanced portable BASIC handheld computer with built-in calculator capabilities. The proprietary CPU, which has a 512KB address space, is optimized for high-precision BCD math and very low power consumption. The proprietary 64KB BASIC operating system automatically incorporates plug-in software and memory modules, allows optional device interfaces such as HPIL or card, maintains a memory file system that may contain an arbitrary number of files, and has been designed so that independent software vendors may conveniently extend or customize the functionality of the machine. HP-71 software may be programmed in BASIC, FORTH, or assembly language.

The internal design of the HP-71 operating system is documented in three volumes, of which this is the first:

* HP-71 Software Internal Design Specification
  - Volume I: Detailed Design Description
  - Volume II: Entry Point and Poll Interfaces
  - Volume III: Operating System Source Listings

A brief overview of these three volumes, which are known collectively as the HP-71 Software IDS, is given below. Related documents which may also be of interest are:

* HP-71 Hardware Specification

* HP-71 HP-IL Module Internal Design Specification
  - Volume I: Detailed Design and Entry Point Description
  - Volume II: Source Listings

* HP-71 FORTH/Assembler ROM Owner's Manual

For information on how to order any of these documents, please contact Systems Engineering Support in the HP Portable Computer Division Product Support Group at (503) 757-2000.

1.1 Structure of the HP-71 Software IDS

This three-volume document discusses the internal design of the HP-71 Operating System in sufficient detail to allow applications software programmed in BASIC, FORTH or assembly language to use the various resources of the Operating System.

1.1.1 Volume I: Detailed Design Description

This volume, which you are currently reading, documents the operating system memory structure, table formats, configuration, operation, interrupt handling, BASIC tokenization, file system, numerical algorithms, and the interfaces to Language Extension (LEX) files. A summary of important system utilities is also provided. Here is a brief description of the remaining chapters in this volume:

Chapter 2 - System Startup and Memory Configuration

This chapter describes how the HP-71 configures memory at power on, memory reset, or after FREE PORT or CLAIM PORT commands.

Chapter 3 - Memory Structure

This chapter describes how memory is initialized after startup configuration. The meanings of various system pointers and locations in system RAM are also discussed, along with certain memory data structures such as system buffers and the various system stacks.

Chapter 4 - System Control

The master control loop (Main Loop) of the operating system is described in this chapter, as well as the system's handling of interrupts.

Chapter 5 - The BASIC Interpreter

An overview of the structure and operation of the HP-71 BASIC Interpreter is provided in this chapter.

Chapter 6 - Language Extension and Binary Files

This chapter describes the structure and use of LEX and BIN file types. Polling of LEX files by the operating system is also covered.

Chapter 7 - BASIC File Considerations

This chapter discusses specifics of BASIC file applications software.

Chapter 8 - Statement Parse, Decompile, and Execution

This chapter describes the procedures for writing code to support LEX file keywords. Keywords have routines to tokenize (parse) them, list (decompile) them, and to execute them. This chapter also gives a detailed description of the BASIC language tokenization used by the HP-71 BASIC Interpreter.

Chapter 9 - Utilities

This chapter summarizes various groups of operating system entry points which applications software may call to perform system operations.

Chapter 10 - Message Handling

This chapter describes how the HP-71 issues error and warning messages, and how LEX files may interface with this process.

Chapter 11 - File System

This chapter describes the HP-71 file system structure and the various file types which the HP-71 supports.

Chapter 12 - Table Formats

This chapter describes the format of various operating system data structures, such as file information buffers, alternative character set buffers, file type tables, and so forth.

Chapter 13 - Internal Data Representation

This chapter describes how data and operands are internally represented in registers, variables, and arrays.

Chapter 14 - Numeric Computation Algorithms

This chapter describes the overall algorithms and procedures used by the HP-71 in mathematical statistical calculations.

Chapter 15 - Clock System

This chapter describes the internal workings of the HP-71 clock system and related considerations for developing clock system applications software.

Chapter 16 - HP-71 Assembler Instruction Set

This chapter describes the HP-71 assembler instruction set and gives the instruction opcodes and execution cycle times.

Chapter 17 - HP-71 Code Examples

This chapter gives examples of how to perform various operations in HP-71 machine language.

Chapter 18 - HP-71 Resource Allocation

This chapter lists the current allocations of HP-71 Operating System resources such as system buffer ID's, LEX file ID's, poll process numbers, file types, reserved RAM, and so forth. It also describes the procedures by which additional resources may be allocated.

1.1.2 Volume II: Entry Point and Poll Interfaces

This volume documents the entry and exit conditions of the 25 categories of supported system entry points that are available to the assembly language programmer, as well as the interfaces to operating system polls of LEX files. Supported entry point categories include keyboard and display interface utilities, math, parse, decompile, and file utilities, and so forth. An index of entry point names and global symbol values is also included.

HP-71 SUPPORTED ENTRY POINT CATEGORIES

1. Address Calculation Utilities
2. I/O Buffer Utilities
3. System Configuration Utiltities
4. Conversion Utilities
5. Display Utilities
6. Decompile Utilities
7. Execute Utilities
8. File Utilities
9. Function Execute Utilities
10. General Purpose Utilities
11. Keyboard Utilities
12. System Math Functions
13. Math Stack Utilities
14. System Level Math Utilities
15. Parse Utilities
16. Poll Interface Descriptions
17. Pointer Utilities
18. Save Stack Utilities
19. Save Utilities
20. Statement Decompile Utilities
21. Statement Execute Utilities
22. Statement Parse Utilities
23. System Level Major Entry Points
24. Time And Date Utilities
25. Variable Management Utilities

1.1.3 Volume III: Operating System Source Listings

This hefty volume contains the full assembly listings of the 76 modules which comprise the HP-71 operating system. All parts of the operating system are listed, including the mainframe token table, BASIC interpreter, math routines, and supported entry points. The supported entry point interface documentation in Volume II is programmatically extracted, categorized, and indexed from comment blocks in these source modules. Therefore Volume II information reappears in scattered form throughout Volume III.

1.2 Operating System Overview

The HP-71 contains a 64KB operating system kernel which resides at address 0. The kernel performs various control functions, and contains the BASIC interpreter. An internal clock system supports time-dependent applications. External software may be added to the machine in the form of files which are interpreted or executed directly by the kernel. These files may be directly added to the computer through plug-in memory modules, or copied into the computer from external media such as magnetic cards or tape.

There are three types of software files which can be interpreted or executed by the HP-71 standard configuration: BASIC, BIN (Binary), and LEX (Language Extension). A FORTH file type may also be invoked when the HP-71 FORTH/Assembler ROM is present in the machine.

BASIC files may be developed on the HP-71 using the built-in BASIC interpreter. BIN, LEX, and FORTH files may be developed on the HP-71 using the FORTH/Assembler ROM.

Type	Format	Method of Invocation	Mode of Execution
BASIC	Tokenized BASIC statements	RUN, CHAIN, or CALL command	Interpretation
BIN	Machine language (binary)	RUN, CHAIN, or CALL command	Direct execution
LEX	Language extension file; adds BASIC keywords, messages, and functional extensions; written in machine language	Through its added BASIC keywords and by polls from operating system	Direct execution
FORTH	FORTH vocabulary	Through FORTH interpreter	Threaded Interpretation

A BASIC or BIN file can be executed as a program or as a subprogram. However, the great flexibility of the HP-71 operating system is due to the manner in which it automatically incorporates LEX files into the operation of the machine.

A LEX file may contain a BASIC keyword token table which is similar in format to the built-in token tables used by the HP-71 BASIC interpreter. Whenever a LEX file is added to the machine, it is automatically "registered" with the operating system. The BASIC command interpreter then references the LEX file's keywords during lexical analysis, making them automatically a part of the HP-71 command language available to the computer's user.

In addition, a LEX file may contain a message table in order to add its own error/warning messages to the machine, or to override the built-in HP-71 error messages for foreign language localization. (An example of such a LEX file is given in the "HP-71 Code Examples" chapter)

Furthermore, the operating system contains outward hooks, called "polls", by which a LEX file may intercept the operation of the machine at a strategic point to extend or customize that operation. At over 80 points in the operating system code when the system is prepared to perform a special task, such as parsing a device name or terminating execution of a program, it "polls" each LEX file present in the machine to find if one wishes to intercept the task.

The polling mechanism is as follows. The operating system jumps to the LEX file's poll handling code, passing a unique code called a "poll process number" that identifies the task to be done. The LEX file may choose to intercede by honoring the documented interface for that poll process number. In this way very sophisticated and detailed customization of the machine's functionality is possible. Polling is described in detail below. The individual poll interfaces are described in Volume II of this document.

Since there is no logical separation of address space between an application program and the HP-71 operating system, a code in a BIN or LEX file may directly access certain system entry points to perform operations ranging from BCD math to file I/O. Over 1700 such entry points are supported by the HP-71 in such a manner that the absolute addresses of these entry points will remain fixed throughout subsequent releases of the operating system ROMs. The interfaces to these entry points are described in Volume II of this document.

1.2.1 Memory Layout

The general layout of the HP-71 physical address space is shown below. Sections marked with an asterisk indicate RAM areas which may be used by applications software for data storage according to the procedures described in this document.

```
+--------------------+
|  Operating System  |
+--------------------+
|  Memory-Mapped I/O |
|   and Display RAM  |
+--------------------+
|  System RAM        |
+--------------------+
| *Reserved RAM      |
+--------------------+
| *MAIN File Chain   |
+--------------------+
| *System Buffers    |
+--------------------+
|  Command Stack     |
+--------------------+
|  CALC Mode Buffers |
+--------------------+
| *Available Memory  |
+--------------------+
| *Environment Stacks|
+--------------------+
|  Independent RAM,  |
|  ROM Modules       |
+--------------------+
```

1.2.2 File System

The HP-71 has a memory-based file system which has no central directory. The main file system is a chain of files, each with its own identifying file header, in Main RAM.

In general, a plug-in ROM module contains its own file chain in the same format as the main file chain. Similarly, a plug-in RAM module can be maintained as an Independent RAM (IRAM) with its own file chain, or it can be pooled with the Main RAM. The operating system's file operations automatically incorporate all file chains present in memory.

1.3 CPU Overview

The HP-71 CPU is a proprietary CPU optimized for high-accuracy BCD math and low power consumption. The data path is 4 bits wide. Memory is accessed in 4-bit quantities called "nibbles" or "nibs". Addresses are 20 bits, yielding a physical address space of 512K bytes.

There are four working 64-bit registers, five scratch 64-bit registers, two 20-bit data pointer registers, one 4-bit pointer register, a 20-bit program counter, a 16-bit input register, and a 12-bit output register. Return addresses are stored on an eight-level hardware return stack that accepts 20-bit addresses. In addition, there 4 hardware status bits, a carry bit, and 16 program status bits. The lower 12 program status bits can be manipulated as a 12-bit register.

For a more detailed overview of the HP-71 CPU, please see the "HP-71 Assembler Instruction Set" chapter.

1.3.1 Registers

The working registers are used for data manipulation. Registers A and C are also used for memory access. The scratch registers are used to temporarily hold the contents of working registers.

In addition, the lower 20 bits of R4 are used during interrupt processing and therefore are not normally available for data storage.

WORKING REGISTERS		SCRATCH REGISTERS	
Name	Size	Name	Size
A	64 bits	R0	64 bits
B	64 bits	R1	64 bits
C	64 bits	R2	64 bits
D	64 bits	R3	64 bits
		R4	64 bits*

* Note: the lower 20 bits of R4 are modified whenever an interrupt occurs, and are generally unavailable for storage

1.3.1.1 Field Selection

Subfields of the working registers may be manipulated by the use of field selection. The possible field selections range from the entire register to any single nibble of the register. Certain subfields are designed for use in BCD calculations. Others are used for data access or general data manipulation. See the "HP-71 Assembler Instruction Set" chapter for a description of the selectable fields.

1.3.2 Pointer Registers

The Data Pointer registers, D0 and D1, are used to contain addresses during memory access, and are used in conjunction with the working registers.

The P Pointer register is used in Field Selection operations with the working registers.

DATA POINTER REGISTERS

```
     +--------------------+          +--------------------+
  D0 |      20 bits       |      D1  |      20 bits       |
     +--------------------+          +--------------------+
```

P POINTER REGISTER

```
      +----------+
   P  |  4 bits  |
      +----------+
```

1.3.3 Input, Output, and Program Counter Registers

The input/output registers are used to communicate with the HP-71 bus. The program counter points to the next instruction to bne executed by the CPU.

INPUT AND OUTPUT REGISTERS

```
      +-----------+             +-----------+
   IN |  16 bits  |         OUT |  12 bits  |
      +-----------+             +-----------+
```

PROGRAM COUNTER REGISTER

```
      +-----------+
   PC |  20 bits  |
      +-----------+
```

1.3.4 Status and Carry Bits

The operating system uses 4 of the program status bits to indicate the state of the operating system. The remaining 12 program status bits are generally available to applications software.

CARRY:	1 bit
PROGRAM STATUS:	16 bits (lower 12 act as the ST register)
HARDWARE STATUS:	4 bits

1.4 HP Support For HP-71 Software Development

HP encourages independent software vendors to develop software for the HP-71. There are a number of system resources, such as unique LEX file ID numbers and system buffer numbers, which may need to be allocated to a particular vendor's software. The procedures for allocating these resources is described in the "HP-71 Resource Allocations" chapter.

Any requests for further information should be directed to Systems Engineering Support in the HP Portable Computer Division Product Support Group at (503) 757-2000.

SYSTEM STARTUP AND MEMORY CONFIGURATION — CHAPTER 2

2.1 System Configuration Overview Including RAM and ROM

```
00000  +---------------------+ LOW
       | Operating System ROM |
20000  +---------------------+
       |  Memory Mapped I/O   |
       |  and Display RAM     |
2F400  +---------------------+     --------------------
       |     System RAM       |              ^
2F986  +---------------------+              |
       |    Reserved RAM      |              |
CONFST +---------------------+              |
       | Configuration Buffer |     Display Driver RAM
       +---------------------+              |
       |                      |              |
       |        User          |              |
       |                      |              v
30000  +  -    -    -    -    +     --------------------
       |                      |              ^
       |       Memory         |              |
       |                      |     Soft Configured and
       |                      |         Plug-in RAM
       |                      |              |
       |                      |              v
RAMEND +---------------------+     --------------------
       |                      |
       |    Plug-In ROMs      |
       |        and           |
       |  Independent RAMs    |
       |                      |
FFC00  +---------------------+
       | Config Reserve Area  |     Unusable
FFFFF  +---------------------+ HIGH
```

For a further breakdown of User Memory, see the "Memory Structure" chapter.

2.2 Entering Deep Sleep

When the computer is turned off, the state of the machine is preserved. All variables, pointers and stacks are preserved. A system configuration is performed upon wakeup from deep sleep.

2.3 Startup/Configuration Sequence

System configuration is performed at coldstart, power-on, and FREE and CLAIM execution. Performing configuration consists of determining what chips are resident on the system bus and assigning an address to each chip. While all chips are on the bus "in parallel", an electrical scheme known as "daisy chaining" determines an order in which the chips are found by the CPU when it is performing configuration.

In a daisy chain, each chip has two special lines: daisy-in (DI) and daisy-out (DO). By creating a chain in which daisy-out of one chip is connected to daisy-in of the next chip, you establish an order. Daisy-in to the first chip is (in most cases) a software-switchable electrical line from the CPU (the one exception is port #0, the internal daisy chain, in which daisy-in to the first chip is wired high).

```
   +----------+       +----------+       +----------+
   |          |       |          |       |          |
---|di      do|-------|di      do|-------|di      do|---
   |          |       |          |       |          |
   +----------+       +----------+       +----------+
     | | | |            | | | |            | | | |
     | | | |            | | | |            | | | |
=======================================================
                     system bus
=======================================================
```

When a chip is unconfigured, it does not occupy address space and its daisy-out is held low. If its daisy-in is low, it will not respond to any CPU instructions. If its daisy-in is high, it will respond to two instructions: C=ID, which returns the chip ID to the CPU (see CHIP ID, below), and CONFIG, which assigns an address to the chip and configures it.

When a chip is configured, it does occupy address space and its daisy-out equals its daisy-in. In this state, the chip will NOT respond to C=ID and CONFIG. So once a chip has been configured, the next chip on the daisy chain is able to identify itself and be configured.

The configuration routine examines the daisy chains corresponding to ports #0 through #5 (see PORT#, below) and configures each chip on each daisy chain. A plug-in device may contain more than one chip and may even contain chips of different types (e.g., ROMs and RAMs). The routine builds lists in the configuration buffer area identifying what is plugged in and where it is configured.

2.4 Configuration Routine -- DETAIL

The configuration code assigns addresses to all soft-configurable devices on the system Bus. The code builds three tables in the configuration buffers: system RAM, other memory (ROM, EEPROM, independent RAM, etc.), memory-mapped I/O. The one-byte configuration buffer IDs for the above configuration tables are, respectively, FF, FE, FD. The exact format of the information in the tables is explained in "Table Formats" chapter.

Following is the pre-configuration memory layout:

```
00000-1FFFF: Operating system
2C000-2C01F: Card reader
2E100-2E3FF: Display RAM
2F400-2FFFF: Display Driver RAM
(FFC00-FFFFF: Reserved for configuration garbage dump)
```

Addresses are assigned to devices as follows:

Memory-mapped I/O is configured in the space 20000-2C000.

System RAM is configured contiguously upward from 30000.

> To achieve this contiguous mapping, system RAM is configured in reverse size order. That is, the largest RAM chips are configured first, then successively smaller chips. This assures that 64 Knib RAMS are configured on 64 Knib boundaries, 32 Knib RAMS on 32 Knib boundaries, etc.

Other memory (ROM, independent RAM, EEPROM, etc.) is put in the space between the end of RAM and FFC00.

> The scheme of where each memory device is configured is fairly complex. The configuration code assures that memory devices are configured on legal boundaries and that consecutive chips within a single plug-in are configured contiguously in the order in which they are encountered on the daisy chain. A bit within the chip ID (explained below) is used to identify the physical boundaries of the plug-in memory.

To explain configuration, the following terms are used below:

PORT#: Physical port location (1-5) whose daisy chain is addressed by a bit (0-4) in output register. Port #0 is the internal daisy chain; it includes all built-in devices and the HPIL port. Ports #1-4 are the ports in the front of the machine (#1 is the leftmost port, etc.). Port #5 is the card reader slot.

DEV#: Position of a plug-in (0-15) in a daisy chain. Unless there is a port extender, all plug-ins will be device #0.

SEQUENCE: Consecutive chips in a module to be used as a single entity (e.g., a quad RAM which appears as one plug-in to the user).

DEVICE TYPE: Type of memory (RAM, ROM, etc., or memory-mapped I/O).

DEVICE CLASS: Identifies exact type of memory-mapped I/O device.

2.4.1 CHIP ID

The CHIP ID is a (usually) mask-programmed 20-bit identifier which is read by the CPU on an ID poll (C=ID instruction). A chip responds to the ID poll if two conditions are met:

1) The chip is unconfigured,

2) Daisy-in is high on the chip.

By examining the daisy chains one at a time and configuring each chip as it is found, the software can locate and identify all soft-configurable chips on the bus.

The chip-id contains the following information:

NIBBLE 0: 15-Log2(size).

Memory Size	Nib 0	MM I/O space
1 knib	F	1 word (16 nibs)
2	E	2
4	D	4
8	C	8
16	B	16
32	A	32
64 (max RAM)	9	64
128	8	128
256 (max memory)	7	256
	6	512
	5	1024

NIBBLE 1: (Reserved for future use)

This nibble from the first chip in a sequence is stored in the configuration table for all sequences.

NIBBLE 2: Device type:

0: RAM
1: ROM (includes EPROM, which cannot be written to)
2: EEPROM
3-6: (unassigned)
7-E: Unusable due to COPY command requirements
F: Memory-mapped I/O

NIBBLE 3: For memory:
(Not used)

For memory-mapped I/O, contains device class:

0: HPIL mailbox
1-15: (Unassigned)

(Note: Card reader is hard configured at 2C000-2C01F)

NIBBLE 4:

bits 0-1: (unassigned)

bit 2: Last chip in sequence (see note (*) below). Always assumed high for MM I/O devices, meaning all such devices have their own entry in the Memory-mapped I/O table.

bit 3: Last chip in module. On a ROM, in general, this bit, like the rest of the ID, is mask-programmed. On RAMs, this chip is typically pad-programmed so the same parts can be used for all chips in a multi-chip RAM module.

The top two bits (bits 2-3 of nibble 4) are used to determine what chips are in what physical plug-ins. Every sequence of chips (e.g., four identical RAMS in a RAM plug-in, an applications pack containing two ROMS, etc.) results in one entry in the configuration tables.

(*) End of sequence (but not module) is identified in one of two ways: 1) next chip returns ID with different value in nibs 0-3, or 2) last chip of sequence has bit 18 set. The second approach is necessary if consecutive, identical chips are to be considered as different sequences, and will probably NEVER be used in the entire lifetime of the machine. But it can be done.

2.4.1.1 Examples

A module containing four 8-Kbit RAMS might return the following sequence of IDs:

```
0000E  0000E  0000E  8000E
```

The resulting table entry would identify the chip size, chip count, device type, physical location, and configuration address of the device.

A module containing two 128-Kbit ROMS, a memory-mapped I/O interface using 2 words of address space, and four 16-Kbit RAMS might present the following sequence of IDs:

```
0010A First ROM            \ one ROM table entry
0010A End of ROM sequence  /
01F0E MM I/O devclass 1    - one MM I/O table entry
0000D Start of RAMS        \
0000D                      | one RAM table entry
0000D                      |
8000D End of module        /
```

```
Restrictions: 16 chips/sequence
              16 sequences/device
              16 devices/port
```

2.5 Configuration Buffer Format

Configuration buffers contain a list of what devices are configured where. The buffers are treated and maintained similarly to system buffers (see chapter on

1) Their ID's are only two nibbles long, and

2) They exist before program memory, while system buffers exist after program memory. This insures that these tables will reside in built-in display driver RAM, rather than some memory which may be removed.

The configuration buffer area is the beginning of non-fixed memory. That is, while its starting location is fixed (first buffer starts at address CONFST), its ending location is not. Configuration buffers are maintained as a linked list whose end is identified with a zero byte. Each buffer has a 5-nibble header consisting of:

```
Buffer ID    First 2 nibbles
Size field   3 nibbles
             This is the size of the data field only.
```

```
CONFST -> +-----------+
          | Buffer ID |       2 nibbles
          ------------+-+
          | Size Field |      3 nibbles
     +--  -------------+
     |    |   Data     |
     |    |            |
     +->  +-----------+-+
          | Buffer ID |
          +-----------+-+
          | Size Field |
     +--  +------------+
     |    |   Data     |
     |    |            |
     |    +------------+
     |         .
     |         .
     |         .
     |    +--+
     +->  |00|
          +--+
```

The header is followed by the data field whose size has been specified in the size field above. Immediately following the data field is either the next buffer ID or a zero-byte identifying the end of the buffer chain.

The contents of the buffers are discussed in the "Configuration Buffer" section in the "Memory Structure" chapter.

2.6 Special Role of High Two Pages in Memory

Provision has been made for allowing devices to be hard-configured in address space without fear that the configuration code will soft-configure something over them. At configuration time, the code examines addresses E0000-E000F. If any of those eight bytes is non-zero, the configuration code will NOT configure anything at or above address E0000. So if a hard-configured device resides there, the space from E0000-FFFFF is reserved and is not available for soft-configured devices. The only time that space may be used is DURING the execution of the configuration code, when it may be needed temporarily for "garbage dump".

2.6.1 Producing a Hard-Configured ROM at E0000

In certain cases it is desirable to produce a ROM which is configured to a fixed location in the HP-71 address space. Hard-configuration is a mask-programmed option which is selected at mask-generation time for the ROM chip. This is because some applications simply cannot be soft-configured. For example, the Debugger ROM must be hard-configured so it will be immune to the configuration code.

Any application which must be hard-configured should either reside at E0000 or reside above E0000 and have something else plugged in which resides at E0000. The presence of some device at E0000 is necessary to insure that the space above E0000 will not be configured over.

2.6.2 Dangers of Hard-Configuring ROMS

There are certain disadvantages to hard-configuring a ROM or other device.

2.6.2.1 Bus Contention

Two devices hard-configured to the same address cannot be plugged in at the same time. Otherwise they will both respond to a READ request at the same time, each contending for use of the bus. This may be electrically harmful to the computer. It will certainly produce useless data, since the results from a bus-contention situation cannot be predicted.

2.6.2.2 Invisible Plug-ins

Aside from noticing that a hard-configured device is there, the operating system will not do anything with the device. Because the device is not soft-configured and therefore has no ID, the operating system has no way of knowing what type of device it is, its size, etc. Its address and its existence will not be recorded in any tables. To use it, there must be some LEXfile around (in a soft-configured device or in main memory) which expects it to be around and knows how to use it.

If, for example, an alternate operating system is written and resides in a ROM hard-configured to E0000, there must also be some LEXfile around which will provide the keyword to give control to that operating system.

2.7 Location of Future System ROMs.

Two possible schemes may be used if the operating system needs to be expanded.

2.7.1 Soft-Configured ROM

Operating system enhancements might be contained in a soft-configured ROM, possibly in a LEXfile. This method would be appropriate for many conceivable enhancements. The disadvantages are that the hard-configured part of the operating system would have to expend some effort to locate the soft-configured part, and there is no guarantee that the soft-configured part will be configured if many devices are plugged in.

2.7.2 Fifth ROM at F8000.

The address space from F8000 to FFFFF might be used to contain a fifth operating system ROM. This would make it unavailable to hard-configured ROMs at E0000 and would require some change to the configuration code. This space is temporarily used during configuration as a "garbage dump" area, but nothing is left configured in the garbage dump area after configuration is done. This means that the configuration code itself certainly could not reside in this fifth ROM.

2.8 Configuration "Garbage Dump"

A definition for a term used in this section: Garbage Dump. During the execution of the configuration code, some plug-ins may be found for which there is no room to configure. Because of the operation of the daisy-chain (a device must be configured before the following device can be configured), it is sometimes necessary to configure such a device "out of the way" so devices after it on the daisy chain can be configured. Such "garbage" devices are configured to end at FFFFF, and to start at whatever location ends them at FFFFF. In other words, a "garbage" 16-kByte ROM would be temporarily configured at F8000. All such devices are unconfigured before the termination of the configuration code. This is referred to throughout this section as "Garbage Dump."

MEMORY STRUCTURE	CHAPTER 3

3.1 Operating System ROM

The operating system is contained in four 16K-byte ROMs hard-configured in the address range 00000-1FFFF. Volume III of this document provides a source code listing of all the operating system modules that fill this address space.

3.2 Memory Mapped I/O and Display RAM

Addr	Name	Memory Size	Comment
	*		
	* * * Display driver addresses		
	*		
2E100	ANNAD1	1	Annunciator column 1
2E101	ANN1.5	1	
2E102	ANNAD2	2	Annunciator column 2
2E104	DD3ST	#2E160-*	Start of display driver 3
2E160	DD3END	#2E1F8-*	End of display driver 3
2E1F8	TIMER3	#2E1FF-*	Timer 3
2E1FF	DD3CTL	1	Display driver 3 control nib
2E200	DD2ST	#2E260-*	Start of display drive
2E260	DD2END	#2E2F8-*	End of display driver 2
2E2F8	TIMER2	#2E2FF-*	Timer 2
2E2FF	DD2CTL	1	Display driver 2 control nib
2E300	DD1ST	#2E34C-*	Start of display driver
2E34C	DD1END		End of display driver 1
2E34C	ANNAD3	2	Annunciator column 3
2E34E	ANNAD4	2	Annunciator column 4
2E350	ROWDVR	#2E3F8-*	Row Drivers
2E3F8	TIMER1	#2E3FE-*	Timer 1
2E3FE	DCONTR	#2E3FF-*	Display contrast nibble
2E3FF	DD1CTL	#2F400-*	Display driver 1 control nib

3.2.1 Display Driver Addresses

The HP-71 display contains two columns of annunciators on the left followed by 132 columns of dots and two more columns of annunciators.

The columns are addressed as follows:

```
          *******************************************
          ****        SLAVE DISPLAY DRIVER II   ****
          *******************************************

                       Leftmost column of annunciators

   ANNAD1   (2E100)              -- Bits 0-2 not connected
                         <---    -- Bit 3
   ANN1.5   (2E101)      AC      -- Bit 4
                         USER    -- Bit 5
                         RAD     -- Bit 6
                                 -- Bit 7 not connected

                       Adjacent column of annunciators

   ANNAD2   (2E102)              -- Bits 0-1 not connected
                               f -- Bit 2
                               g -- Bit 3
                             BAT -- Bit 4
                                 -- Bits 5-7 not connected

   DD3ST    (2E104)   Columns 0-45    (46 Columns)
   DD3END   (2E15F)

   TIMER3   (2E1F8)   Timer (least sig. nib (LSB) at lowest address)
                            (6 nibbles)

   DD3CTL   (2E1FF)   Status Nibble:
                                WRITE         READ
                        LSB 0 -- RAM          RAM
                            1 -- RAM          RAM
                            2 --
                        MSB 3 -- Enable Timer
```

```
        ******************************************
        ****      SLAVE DISPLAY DRIVER I     ****
        ******************************************

 DD2ST   (2E200)  Columns 46-93   (48 Columns)
 DD2END  (2E260)

 TIMER2  (2E2F8)  Timer (least sig. nib at lowest address)
                        (6 nibbles)

 DD2CTL  (2E2FF)  Status Nibble:
                            WRITE             READ
                   LSB 0 -- RAM               RAM
                       1 -- RAM               RAM
                       2 --
                   MSB 3 -- Enable Timer


        ******************************************
        ****      MASTER DISPLAY DRIVER      ****
        ******************************************

 DD1ST   (2E300)  Columns 94-131  (38 Columns)
 DD1END  (2E34C)

 ANNAD3  (2E34C)  Right column of annunciators
                         -- Bits 0-2 not connected
                     0   -- Bit 3
                     1   -- Bit 4
                     2   -- Bit 5
                     3   -- Bit 6
                     4   -- Bit 7

 ANNAD4  (2E34E)  Rightmost column of annunciators
                         -- Bits 0-2 not connected
                   ((*)) -- Bit 3
                    ---> -- Bit 4
                    PRGM -- Bit 5
                    SUSP -- Bit 6
                    CALC -- Bit 7

 ROWDVR  (2E350)  Row Lines (16 Nibbles).  Should be set
                  by software as follows: 8001400220041008

 TIMER1  (2E3F8)  Timer (least sig. nib at lowest address)
                        (6 nibbles)

 DD1CTL  (2E3FF)  Display Control Nibble:
                            WRITE             READ
                   LSB 0 -- Display On        Same
                       1 -- Display Blink     Same
```

```
                    2 -- Display Test   Very Low Bat
                MSB 3 -- Enable Timer   Low Bat
```

3.3 System RAM

```
Addr   Name          Memory Size   Comment
-----  ------        -----------   ------------------------

       *
       * * *  Start of interrupt RAM
       *
2F400  INTR4         16            (R4 and D0)
2F410  INTA          16            (A reg)
2F420  INTB          16            (B reg)
2F430  INTM          8             (Misc stuff)
       *
       *  INTM is mode-Pointer-Carry-Return stack
       *
       * * *  End of interrupt RAM
       *
       CMOSTV EQU    #168F         Value for CMOS test word
2F438  CMOSTW        4             CMOS test word
2F43C  VECTOR        5             Interrupt vector
2F441  ATNDIS        1             Attention disable flag
2F442  OFFFLG
2F442  ATNFLG        1             Attention key hit flag
2F443  KEYPTR        1             Key buffer pointer
2F444  KEYBUF        15*2          Key buffer
2F462  KEYSAV                      (LSB = Bottom Row)
2F462  KCOLD         1             14th column keymap
2F463  KCOLC         1             13th
2F464  KCOLB         1             12th
2F465  KCOLA         1             11th
2F466  KCOL9         1             10th
2F467  KCOL8         1             9th
2F468  KCOL7         1             8th
2F469  KCOL6         1             7th
2F46A  KCOL5         1             6th
2F46B  KCOL4         1             5th
2F46C  KCOL3         1             4th
2F46D  KCOL2         1             3rd
2F46E  KCOL1         1             2nd
2F46F  KCOL0         1             1st

2F470  DISINT        1         Interrupt ignore flag
                                      used in keyscan
       *
       * Pseudo-device Display Driver Memory
```

```
       *
2F471  WINDST     2         Window start
2F473  WINDLN     2         Window len
2F475  DSPSTA     6         User status save, Dsp status
2F47B  ESCSTA     1         Escape status
2F47C  FIRSTC     2         Buffer position of 1st char
2F47E  CURSOR     2         Buffer position of cur
2F480  DSPBFS     2*96      96 char buffer (2 nibs/char)
2F540  DSPBFE
2F540  DSPMSK     96/4      96 bits (4 bits/nib)
       *
       *  System Pointer Allocations
       *
2F558  MAINST     5         Main Program Memory Start
2F55D  UPD1ST               Start of Update Addresses #1
2F55D  CURRST     5         Current File Start
2F562  PRGMST     5         Current Program Start
2F567  PRGMEN     5         Current Program End
2F56C  CURREN     5         Current File End
2F571  IOBFST               Start of System buffers
2F571  MAINEN     5         Main Program Memory End
2F576  IOBFEN               End of System buffers
2F576  CLCBFR     5         Calc Mode Pointers
2F57B  RFNBFR     5
2F580  RAWBFR     5
2F585  CLCSTK     5         Calc Stack token stream start
       *                     SYSEN,OUTBS,AVMEMS collapsed
       *                     here at end of CALC mode
2F58A  SYSEN      5         End of RAM used by System
       *                     OUTBS and AVMEMS collapsed
       *                     here at end of Parse,
       *                     Decompile,TRANSFORM
2F58F  OUTBS      5         Output Buffer Start
       *                     Output Start for Parse/Decomp
2F594  AVMEMS     5         Available Memory Start
2F599  UPD1EN               End of Update Addresses #1
       *
2F599  TASTK
2F599  MTHSTK               Arithmetic Stack
2F599  AVMEME     5         End of Available Memory
       *                      (AVMEME collapsed to SAVST
       *                        after statement ex
2F59E  SAVSTK               Save Area Stack Pointer
2F59E  TFORN
2F59E  FORSTK     5         FOR/NEXT Stack
2F5A3  TGSBS
2F5A3  GSBSTK     5         GOSUB Stack
2F5A8  ACTIVE     5         Active Variable Space
2F5AD  CALSTK     5         CALL Stack
2F5B2  RAMEND     5         End of Memory
       *
```

```
        *  Variable List Pointers
        *
2F5B7   PRMPTR          7               Parameter Chain Pointer
2F5BE   CHNLST                          Variable Chain Pointer List
2F5BE                   26*7            26 Chains (7 nibs/chain)
        *
        * The following pointers are position dependent
        *
        *   PCADDR through TMADR3 adjusted by RFADJ+
        *   PCADDR through DATPTR saved by CALL
        *   CNTADR through TMADR3 zeroed by CLRSTK/CLPSTK
        *
2F674   UPD2ST                          Start of Update Addresses #2
2F674   DSPCHX          5               Pointer to external display
2F679   PCADDR          5               Program Counter  Stmt Length
2F67E   CNTADR          5               Continue Address
2F683   ERRSUB          5               ON ERROR-GOSUB Return Address
2F688   ERRADR          5               ON ERROR Statement Address
2F68D   ONINTR          5               ON INTRPT Statement Address
2F692   DATPTR          5               DATA Statement Pointer
2F697   TMRAD1          5               ON TIMER#1 Statement Address
2F69C   TMRAD2          5               ON TIMER#2 Statement Address
2F6A1   TMRAD3          5               ON TIMER#3 Statement Address
2F6A6   UPD2EN                          End of Update Addresses #2
        *
        * The following Timer Intervals are position dependent
        *     with TMRAD1 - TMRAD3
        *
2F6A6   TMRIN1          8               TIMER#1 Interval
2F6AE   TMRIN2          8               TIMER#2 Interval
2F6B6   TMRIN3          8               TIMER#3 Interval
        *
2F6BE   STSAVE          3               Status saved during Expr Exec
2F6C1   LDCSPC          5               Addr of space after line #
2F6C6   INBS            5               Input buffer start
2F6CB   AUTINC          4               Increment value for AUTO
2F6CF   LEXPTR          5               Temporary storage for RESPTR
2F6D4   CMDPTR                          Command Stack pointer
2F6D4   INADDR          5               Stmt Len ptr: Parse/Decomp
        *
2F6D9   SYSFLG          16              System flags
2F6E9   FLGREG          16              User flags
2F6F9   TRPREG                          IEEE exception traps
2F6F9   INXNIB          1               Inexact result trap
2F6FA   UNFNIB          1               Underflow trap
2F6FB   OVFNIB          1               Overflow trap
2F6FC   DVZNIB          1               Divide by zero trap
2F6FD   IVLNIB          1               Invalid result trap
        *
        *  Random Number Seed
        *
```

```
2F6FE  RNSEED        15
       *
       *  Alarm Clock System RAM
       *
2F70D  NXTIRQ        12               Time of next SREQ
2F719  ALRM1         12               ON TIMER #1
2F725  ALRM2         12               ON TIMER #2
2F731  ALRM3         12               ON TIMER #3
2F73D  ALRM4         12               Time of timeout
2F749  ALRM5         12               Time of WAIT expiration
2F755  ALRM6         12               Time external alarm expires
2F761  PNDALM        2                Bitmap of pending alarms
       *
       *  Storage needed for accuracy factor stuff
       *
2F763  TIMOFS        12               Time error offset
2F76F  TIMLST        12               Time last set
2F77B  TIMLAF        12               Time of last AF correction
2F787  TIMAF         6                Accuracy factor
       *
2F78D  IS-TBL                         Table of "IS" assignments
2F78D  IS-DSP        7
2F794  IS-PRT        7
2F79B  IS-INP        7
2F7A2  IS-PLT        7
       *
2F7A9  MBOX^         3                HP-IL Mailbox pointer
2F7AC  LOOPST        1                HP-IL loop status
2F7AD  STATAR        3                STATISITICAL ARRAY NAME
2F7B0  TRACEM        1                TRACE MODE (0,2,4,6)
2F7B1  DSPSET        1                Display device set up on HPIL
       *
2F7B2  LOCKWD        8*2              Password
       *
2F7C2  RESREG        34               Result register
       *
       *
       *  ERR# through ERRL# are position dependent
       *
2F7E4  ERR#          4                Execution Error Number
2F7E8  CURRL         4                Current Line# Referenced
2F7EC  ERRL#         4                Execution Error Line#
       *
       *  Snapshot Buffer and Return Stack Save Buffer
       *
2F7F0  SNAPBF        16+16+5+5+5      Snapshot Buffer
2F81F  RSTKBp        1                Return Stack Save Buffer Ptr
2F820  RSTKBF        16*5             Return Stack Save Buffer
       *
2F870  MLFFLG        1                Multi-Line Function FLag
       *
```

```
2F871  STMTR0                    Statement scratch RAM
2F871  S-R0-0      5
2F876  S-R0-1      5
2F87B  S-R0-2      5
2F880  S-R0-3      1
       *
2F881  STMTR1
2F881  S-R1-0      5
2F886  S-R1-1      5
2F88B  S-R1-2      5
2F890  S-R1-3      1
       *
2F891  STMTD0      5
2F896  STMTD1      5
       *
2F89B  FUNCR0                    Function scratch RAM
2F89B  F-R0-0      5
2F8A0  F-R0-1      5
2F8A5  F-R0-2      5
2F8AA  F-R0-3      1
       *
2F8AB  FUNCR1
2F8AB  F-R1-0      5
2F8B0  F-R1-1      5
2F8B5  F-R1-2      5
2F8BA  F-R1-3      1
       *
2F8BB  FUNCD0      5
2F8C0  FUNCD1      5
       *
       *  TRANSFORM Scratch RAM
       *
2F8C5  TRFMBF      60            Used by TRANSFORM command
       *
       *
2F901  SCRTCH                    Scratch RAM
2F901  SCRST0      4*16          Scratch stack (Mantissas & s
2F941  SCREX0      5             Scratch stack exponent
2F946  SCROLT      2             Character scroll timer
2F948  DELAYT      2             Display timeout value
2F94A  NEEDSC      1             Scroll mode needed
2F94B  PRMCNT      2             CALL parameter count
2F94D  DPOS        2             Current DISP column
2F94F  DWIDTH      2             DISP width
2F951  SCREX1      5             Scratch stack exponent 1
2F956  PPOS        2             Current PRINT column
2F958  PWIDTH      2             PRINT width
2F95A  EOLLEN      1             Length of ENDLINE stri
2F95B  EOLSTR      2*3           ENDLINE string (3 chars max
2F961  SCREX2      5             Scratch stack exponent 2
2F966  SCRPTR      1             Scratch stack pointer
```

```
 2F967  DEFADR        8               Key definition info
 2F96F  CHN#SV        2               Channel # save
 2F971  SCREX3        5               Scratch stack exponent 3
        *
 2F976  MAXCMD        1               # of Command Stack entries
        *
 2F977  CSPEED        5               Clock speed (Hz/16)
        *
        * The following 10 nibbles are used by HP-IL ROM
        *
 2F97C  ERRLCH        1               Error latch
 2F97D  TERCHR        2               Terminating char for ENTER
 2F97F  HPSCRH        7               HP-IL Reserved.
        *                             (INTPND,ICAUSE,IMASK,LSTDDC)
        *
 2F986  RESERV        48*2            Reserved Memory.
        *
        * Configuration table start
        *
 2F9E6  CONFST
```

3.3.1 Interrupt RAM (INTR4 - VECTOR,DISINT)

The interrupt routine uses 56 nibbles of RAM (INTR4, INTA, INTB, INTM) to save the contents of A(W), B(W), C(W), D0, P, Carry, Hex/Dec Mode.

The interrupt routine checks the RAM address VECTOR to see if an alternate interrupt handler has been enabled. Before processing any interrupt, four nibbles at CMOSTW (CMOS test word) are checked to verify that RAM is likely not corrupt. (The CMOS test word is immediately next to the VECTOR address since it is unlikely to accidentally change one address without changing the other.)

If the 5 nibble value at VECTOR is zero then normal interrupt processing is performed.

The nibble at DISINT is used to cause exactly one interrupt to be ignored. If the interrupt routine sees this nibble set to a non-zero value it will return immediately without any processing except to check for a "Module Pulled" interrupt and to zero this nibble. This is used during keyscan to side-step the interrupt that may result when the output register has been used to check individual key columns while doing synchronous (i.e., not from interrupt routine) keyscans.

3.3.2 Keyboard Buffer/Flags (ATNDIS - KEYSAV)

The keyboard system has a fifteen key buffer which is preceeded by a nibble indicating how many keys are in the buffer. This buffer is treated as a FIFO where the oldest key in the buffer is at the lowest address in the buffer (ie. pointed to by KEYBUF). The pointer nibble is named KEYPTR.

In addition to the key buffer, a "bit map" of which keys were down during the latest keyscan is maintained in the fourteen nibbles starting at KEYSAV. There are 4 rows of keys on the keyboard and each nibble of the KEYSAV buffer holds 4 bits representing the state of a particular key column. The least sig. bit of each nibble represents the key in the bottom row of that column and the most sig. bit represents the key in the top row of that column. The 14th key column is pointed to by KEYSAV. KEYSAV+13 points to the 1st key column.

The nibble at ATNFLG is decremented each time the keyscan routine finds the attention key down. It will not however be decremented from 1 to 0 since this would hide the fact that the key was ever pressed. The intention is that this flag can be used both as a flag that the attention key has been pressed and as a convienient way to tell if it has been pressed more than once.

The nibble at ATNDIS is a special location that if non-zero will cause the keyscan routine to treat the attention key as it would any other key. The attention key normally causes the key buffer to be flushed and the ATNFLG flag to be set, as well as setting the Except (S12) global status bit.

3.3.3 Pseudo-Device Display Driver (WINDST - DSPMSK)

The display driver uses a buffer of 96 consecutive bytes to hold the display buffer (DSPBFS). Each of these bytes holds one display character.

The display routines use several additional bytes to describe how the LCD should look. The byte at WINDST is the first LCD character position that should be used to display the contents of the buffer. The next byte (WINDLN) says how many LCD character positions (starting at WINDST) to use to represent the buffer. The first character of the buffer that should be put into the display is held in the byte at FIRSTC. The position of the cursor in the display is held in the byte at CURSOR. All of these bytes are represented base 0 (i.e. value 0 is the lowest possible value).

In addition to these bytes, another six nibbles are used to save status bits. The first three nibbles at DSPSTA are used to store the calling routines status bits while in the various display routines. The next three nibbles are used to hold status bits relevant to the display routines. See the display routines' documentation for a more complete description of these bits.

One nibble (ESCSTA) is used to keep track of the escape status of the display routines. This nibble indicates if the routines are in the middle of an escape sequence.

Following the display buffer is an address called DSPMSK. The 24 nibbles at this address contain 24*4 (96) bits, one of which corresponds to each of the bytes in the display buffer. The lowest address nibble maps to the highest addressed 4 bytes in the buffer and the nibble at DSPMSK+23 corresponds to the first 4 bytes in the display buffer. The most sig. bit of each nibble corresponds to the lowest addressed byte of the group of 4. These bits determine whether a particular character in the buffer is a protected, unreadable character. As characters are sent to the display this bit will be set for the character if the cursor is off. This makes the character unreadable and protects it so that the cursor cannot be positioned over it.

3.3.4 User Memory Pointers (MAINST - RAMEND)

USER MEMORY POINTERS

```
<Low>             +---------------------+
                  |     System RAM      |
                  +---------------------+
                  |    Reserved RAM     |
   CONFST    =    +---------------------+
                  | Configuration Buffer|
   MAINST   -->   +---------------------+ <- MAIN File Chain
                  |                     |     Start
                  |        File         |
                  |                     |
                  +---------------------+
                  |        . . .        |
   CURRST   -->   +---------------------+ <- Current File St
   PRGMST   -->   |                     | <- Current Program
                  |    Current  File    |     Start
   PRGMEN   -->   |                     | <- Current Program
                  |                     |     End
   CURREN   -->   +---------------------+ <- Current File End
                  |                     |
                  |        File         |
                  |                     |
                  +---------------------+
                  |        . . .        |
MAINEN = IOBFST -->+---------------------+ <- System Buffer
                  |   System Buffers    |     Start
IOBFEN = CLCBFR -->+---------------------+ <- System Buffer
                  |    Command Stack    |     End
                  |- - - - - - - - - -  |
                  | CALC Refined Buffer |
   RFNBFR   -->   +---------------------+
                  | CALC Left Raw Buffer|
   RAWBFR   -->   +---------------------+
                  | CALC Right Raw Buffer|
   CLCSTK   -->   +---------------------+
                  | CALC Token Stream   |
    SYSEN   -->   +---------------------+
                  |  Temp Input Buffer  |
    OUTBS   -->   +---------------------+ <- Output Buffer
                  |    Output Buffer    |     Start
   AVMEMS   -->   +---------------------+ <- Available Memory
                  | Available Memory |  |     Start
                  |         .        v  |
<High>            |         .           |
```

```
 <Low>                  |         .          |
                        |         .       ^  |
                        | Available Memory |  |
MTHSTK = AVMEME -->     +--------------------+ <- Available Memory
                        |                 ^  |      End
                        |     Math Stack   |  |
        FORSTK -->      +--------------------+ --+
                        |   FOR/NEXT Stack    |   |
        GSBSTK -->      +--------------------+   |  Current
                        |     GOSUB Stack     |   -->Environment
        ACTIVE -->      +--------------------+   |
                        |   Active Variables  |   |
        CALSTK -->      +--------------------+ --+
                        |     Environment     |   |
                        |  Information Blocks |   |
                        +--------------------+   |
                        |   FOR/NEXT Stack    |   |  Prior
                        +--------------------+   -->Environment
                        |     GOSUB Stack     |   |
                        +--------------------+   |
                        |      Variables      |   |
<High>  RAMEND -->      +--------------------+ --+
```

From Low to High Memory:

MAINST - MAIN File Chain Start = Configuration Buffers End
Points to the first file header in the MAIN file chain

CURRST - Current File Start
Points to the first nibble of the current file header

PRGMST - Current Program or Subprogram Start
Points to first nibble of current program or subprogram

PRGMEN - Current Program or Subprogram End
Points past last nibble of current program or subprogram

CURREN - Current File End
Points past last nibble of current file

MAINEN - MAIN File Chain End = System Buffer Start (IOBFST)
Points past 00 byte at end of MAIN file chain

CLCBFR - CALC Mode Buffer Start = System Buffer End (IOBFEN)

RFNBFR - CALC Mode Refined Buffer

RAWBFR - CALC Mode Raw Buffer

CLCSTK	- CALC Mode Token Stack	
SYSEN	- System RAM End	
OUTBS	- Output Buffer Start	
AVMEMS	- Available Memory Start	= Output Buffer End
AVMEME	- Available Memory End	= Top of Math Stack (MTHSTK)
FORSTK	- Top FOR/NEXT Stack	= Top of Save Stack (SAVSTK)
GSBSTK	- Top GOSUB Stack	= Bottom of FOR/NEXT Stack
ACTIVE	- Active Variable Pointer	= Bottom of GOSUB Stack
CALSTK	- CALL Stack	= Bottom of Active Variables
RAMEND	- User RAM End	= Bottom of CALL Stack

3.3.5 Parameter Chain Pointer (PRMPTR)

The parameters of a user-defined function are pointed to by PRMPTR. The first two nibbles of PRMPTR is the parameter count:

Parameter count	Meaning
00	Currently is not executing an user-defined function
01-0F	Currently is executing an user-defined function, the number of the parameters of the user-defined function = count -1

The next five nibbles of the PRMPTR is the pointer to the chain of parameters. The parameters of the user-defined function are stored in a fashion similar to the program variables, except all parameters are stored in the same chain, regardless of the starting letter of the parameter name.

3.3.6 Variable Chain Pointer List (CHNLST)

Beginning at CHNLST are 26 seven-nibble chain pointers; each pointer is associated with a list of currently-existing variables. A variable is put into a particular list according to the letter of the alphabet which its name contains. For example, variables R, R7, R$, and R3$ are all in the same list. See the section on variables for details on variable list construction. A chain pointer has two parts: a variable count and an address. The variable count is a two-nibble quantity telling how many variables exist in the chain at that time. The address field gives the absolute address of the start of the variable chain.

3.3.7 Statement/Program Execute RAM (DSPCHX-TMRIN3)

The following addresses (DSPCHX through TMRAD3) are updated whenever memory moves within system or user RAM. The symbolic names: UPD2ST and UPD2EN indicate this range.

DSPCHX Zero if no external character display device is active. Otherwise, the contents are used as an address for an external display handler for each character sent via DSPCHA routine.

PCADDR Pointer to statement length byte of statement currently executing.

CNTADR Continue Address of currently halted program.

ERRSUB Return address of ON ERROR GOSUB statement. Prevents infinite loop if error within ON ERROR GOSUB execute

ERRADR Address within ON ERROR statement pointing at GOTO or GOSUB. Remainder of statement is executed when an error occurs within a program.

ONINTR Address within ON INTERUPT (HP-IL) statement pointing at GOTO or GOSUB. Remainder of statement is executed when an interrupt occurs.

DATPTR DATA statement READ pointer.

TMRAD1 - TMRAD3 ON TIMER statement addresses for Timer#1-3, respectively. Points at GOTO or GOSUB within ON TIMER statement. When timer expires, remainder of statement is executed.

3.3.8 Miscellaneous BASIC RAM (STSAVE - INADDR)

TMRIN1 - TMRIN3 ON TIMER statement timer interval for Timer#1-3, respectively. Timer is reactivated for the corresponding timer interval when an ON TIMER...GOTO expires, or on return from an ON TIMER...GOSUB.

STSAVE Saved status bits during Expression Execute (EXPEXC).

LDCSPC Cursor position for decompile of BASIC program lines and user-defined keys.

INBS Input Buffer Start. A floating pointer indicating the start of the input buffer being parsed. Set at the beginning of Line Parse. May point to the Command Stack, Start-up Buffer, TRANSFORM Input Buffer or Direct Execute Key Buffer.

AUTINC AUTO increment value for AUTO command. This RAM location doubles as the AUTO mode flag: If zero, then not in AUTO mode.

LEXPTR Position of Input pointer prior to last NTOKEN call. Used in statement parse.

CMDPTR
INADDR Pointer to statement length byte for statement currently being parsed or decompiled. (Also used for Command Stack pointer - CMDPTR)

3.3.9 System and User Flags (SYSFLG - FLGREG)

There are 64 user flags (numbered 0-63) and 64 system flags (numbered -64 to -1). Within each nibble, the lowest numbered flag is in the least significant bit. These flags are stored in 128 consecutive bits starting at address SYSFLG:

```
 <Low>                  +------+
         SYSFLG -->     | -1   |          System Flags
                        +------+
                        | -2   |
                        +------+
                           .
                           .
                           .
                        +------+
                        | -64  |
                        +------+
         FLGREG -->     |  0   |          User Flags
                        +------+
                        |  1   |
                        +------+
                           .
                           .
                           .
                        +------+
                        |  63  |
 <High>                 +------+
```

The user can set and clear all user flags and those system flags numbered -1 to -32. The user may test the status of all user and system flags.

Refer to the "Table Formats" chapter for a summary of flag assignments.

3.3.10 Traps (INXNIB - IVLNIB)

There are 5 math exception traps stored in 5 consecutive nibbles starting at address TRPREG:

```
    <Low>                                 +-----+
             TRPREG = INXNIB -->  | INX |
                                          +-----+
                      UNFNIB -->  | UNF |
                                          +-----+
                      OVFNIB -->  | OVF |
                                          +-----+
                      DVZNIB -->  | DVZ |
                                          +-----+
                      IVLNIB -->  | IVL |
   <High>                                 +-----+
```

Refer to the "Table Formats" chapter for details on trap settings.

3.3.11 Random Number Seed (RNSEED)

The current random number seed (updated by RANDOMIZE and used by RND) is stored in 15 consecutive nibbles starting at address RNSEED.

3.3.12 Alarm Clock System RAM (NXTIRQ - TIMAF)

The following RAM is used by the internal clock system:

Label	Size(nibs)	Function
NXTIRQ	12	Time of next clock service request
ALRM1	12	Time of next timer#1 request
ALRM2	12	Time of next timer#2 request
ALRM3	12	Time of next timer#3 request
ALRM4	12	Time of 10-minute timeout
ALRM5	12	Time of end of pause
ALRM6	12	Time of external alarm
PNDALM	2	Bitmap of pending alarms
TIMOFS	12	Time error offset
TIMLST	12	Time of last EXACT
TIMLAF	12	Time of last AF correction
TIMAF	6	Accuracy factor

3.3.13 "IS" Table Assignments (IS-TBL)

This table holds information defining the current state of DISPLAY IS, PRINTER IS, KEYBOARD IS and PLOTTER IS. The destination of each of these assignments can theoretically be any HP-IL device or the LCD display; however there are some combinations that don't make sense and should not be allowed. There is a 7 nibble table entry for each of these devices. Each entry has the following format and definition:

```
    Nib 3
 bit 3   bits 2-0
 -----   ---------
   X       0          Address specified
                        Nibs 2-0:  Address, loop#
                                     or FFF if not known
                        Nibs 6-4:  Address, loop#
   X       1          Type specified (loop 0)
   X       2            "      "      (loop 1)
   X       3            "      "      (loop 2)
                        Nibs 2-0:  Address, loop#
                                     or FFF if not known
                        Nib 6:  Sequence #
                        Nibs 5-4:  Accessory id
   X       4          IO buffer for device ID/Volume label
                        Nibs 2-0:  Address, loop#
                                     or FFF if not known
                        Nibs 6-4:  Buffer #
   X       5          Multiple assign buffer
                        Nibs 2-0:  FFF
                        Nibs 6-4:  Buffer #
   X       6          Device ID specified
                        Nibs 2-0:  Address, loop#
                        Nibs 6-4:  Buffer #
   1       7          Unassigned or not HPIL
                        Nibs 2-0:  FFF if not assigned or
                                     Fxx if not HPIL (where xx
                                     is not FF)
                        Nibs 6-4:  FFF if not assigned but
                                     not defined if not HPIL
```

X = 1 if device OFFed, 0 otherwise

3.3.14 HP-IL RAM (MBOX,LOOPST,DSPSET)

MBOX^ Used by HPIL ROM as a pointer to its mailbox. Three nibbles are multiplied by 16 and added to 20000 to get mailbox address.

LOOPST Used by HPIL ROM to keep track of loop status.
Bit 3: Device "OFFed".
Bit 2: Last call to START found HPIL mailbox in device mode.
Bit 1: (Reserved)
Bit 0: (Reserved)

DSPSET Used by HPIL ROM to indicate status of display device.
Bit 3: Display device is set up
* Following ONLY valid if Bit 3 is TRUE!!
Bit 2: Display is a HP82163 video interface (Retransmit line if insert or delete).
Bit 1: Display device is line output only (printer)
Bit 0: Display code was "OFFed" if 0.

3.3.15 STAT Array (STATAR), TRACE Mode (TRACEM)

STATAR Name of the currently selected STAT array
TRACEM Indicates current TRACE Mode:
0 = No TRACE
2 = TRACE FLOW
4 = TRACE VARS
6 = TRACE FLOW, TRACE VARS

3.3.16 LOCK Password (LOCKWD)

The lockword supplied by the user in the lock command is stored in the 16 nibbles starting at LOCKWD. If there is no lockword, these 16 nibbles are all zeroes. The lockword is not encrypted.

3.3.17 Result Register (RESREG)

The result register holds the value of the most recently executed numeric expression. This value is updated whenever a numeric value is DISPlayed, PRINTed, or stored into a variable.

3.3.18 Error Number (ERRN)

The number of the most recent error (ERRN) is stored in RAM location ERR#. This location is set to zero at cold start, and changed only in the message driver (MFERR*). The message number is encoded in four nibbles: abcd, where
ab = LEX ID# in hex
cd = message number in hex.

3.3.19 Current Line (CURRL)

The current line number is stored in CURRL, as a four digit decimal number. At coldstart, CURRL is set to zero. Editing the current BASIC file updates CURRL to the line number being inserted, replaced or deleted. Recalling a BASIC program line to the display changes the current line. The FETCH statment, Cursor Up, Cursor Down, Cursor Top, and Cursor Bottom recall a program line. Executing an EDIT statement changes the current line to the first line of the specified Edit file. A GOTO from the keyboard sets the current line to the specified line number or line number containing the specified label.

During program execution, CURRL is not updated until the program is halted. If program execution halts due to a PAUSE, STOP, or END statement, the line containing the statement becomes the current line. If the program halts due to an implicit END (the last line of the program is reached), CURRL becomes the last line of the program. If program execution halts because the ATTN is hit, the line containing the next statement to be executed becomes the current line.

3.3.20 Error Line Number (ERRL#)

The line number of the most recent execution error (ERRL) is stored in RAM location ERRL#. This location is set to zero at cold start, and changed only in the message driver (MFERR*). CURRL is updated to the new current line and is also placed in ERRL#; it is a four digit decimal number.

3.3.21 Snapshot Buffer (SNAPBF)

This area of RAM is used to temporarily hold a snapshot of CPU registers A, D, D0, D1, and C(A). It is 47 nibbles in size. For details on saving and restoring CPU snapshots, see routines SNAPSV and SNAPRS.

3.3.22 Return Stack Save (RSTKBp,RSTKBF)

This area of fixed RAM holds up to 16 stack levels from the hardware stack. It is administered as a LIFO (last in, first out) circular stack by the routines R<RSTK (saves stack levels) and RSTK<R (restores stack levels). The one-nibble pointer, RSTKBp, contains an index (0 thru 15) of the next 5-nibble slot available for storing a stack level.

When a stack level is stored, the pointer is bumped, and it wraps around to zero when it passes 15. Conversely, the pointer is decremented when a stack level is removed, and the pointer wraps around to 15 when it passes 0. Therefore, if 16 levels have been stored on the stack, storing a 17th level will overwrite the oldest level on the stack.

Note that these saved stack levels are NOT updated when memory moves. Also, these saved stack levels will not necessarily remain intact when EXPEXC is called.

3.3.23 Multi-Line Function Flag (MLFFLG)

MLFFLG is the multi-Line function flag. ENDDEF statement sets it to nonzero. This allows statements to determine if a multi-line user-defined function was invoked during expression execute. They can then know whether memory could have changed. This flag may also be set by other functions that may have changed memory.

To know whether anything could have happened to "memory" during expression execution, this nibble should be cleared before calling expression execute. If it is set upon return, either a user defined function or some other "harsh" function has been invoked during the expression evaluation.

3.3.24 Statement, Function Scratch (STMTR0 - FUNCD1)

Some RAM is maintained specifically to be used as scratch space during statement and function execution. The 42 nibbles starting at STMTR0 are referred to as the statement scratch area, and the 42 nibbles immediately following (starting at FUNCR0) constitute the function scratch area.

The latter 42 nibbles are available during function execution, and all 84 nibbles are available during statement execution. Naturally, the function scratch area will probably be used during expression execution.

Of great importance to users of these scratch areas is the fact that this RAM is Untouched by utility routines, including display routines, message routines and the clock system. Thus, these scratch areas are often used for storing things while calling particularly disruptive utilities.

The exact layout of the statement and function scratch RAM is as follows (broken down into fields and subfields):

LABEL #nibbles comment

-----	--------		-------
			Start of statement scratch
STMTR0	16		16-nibble field
S-R0-0		5	5-nibble subfield
S-R0-1		5	5-nibble subfield
S-R0-2		5	5-nibble subfield
S-R0-3		1	1-nibble subfield
STMTR1	16		16-nibble field
S-R1-0		5	5-nibble subfield
S-R1-1		5	5-nibble subfield
S-R1-2		5	5-nibble subfield
S-R1-3		1	1-nibble subfield
STMTD0	5	5	5-nibble field
STMTD1	5	5	5-nibble field

(total)	42	42	End of statement scratch
			Start of function scratch
FUNCR0	16		16-nibble field
F-R0-0		5	5-nibble subfield
F-R0-1		5	5-nibble subfield
F-R0-2		5	5-nibble subfield
F-R0-3		1	1-nibble subfield
FUNCR1	16		16-nibble field
F-R1-0		5	5-nibble subfield
F-R1-1		5	5-nibble subfield
F-R1-2		5	5-nibble subfield
F-R1-3		1	1-nibble subfield
FUNCD0	5	5	5-nibble field
FUNCD1	5	5	5-nibble field

(total)	42	42	End of function scratch

3.3.25 TRANSFORM Scratch RAM (TRFMBF)

This area of RAM is used during execution of the TRANSFORM command and is OFF LIMITS to any parse, decompile, or transformation related routine. It is 60 nibs in size.

3.3.26 Scratch RAM (SCRTCH)

The area used for the scratch math stack (below) is also used as a general purpose, highly volatile scratch RAM area, labeled SCRTCH. This is to be distinguished from Statement and Function Scratch (above), which is less volatile. The ALMSRV routine uses part of SCRTCH RAM to avoid destroying CPU scratch registers. The display routines also use it during <CR> and <LF> processing by virtue of their calling ALMSRV. Routines which use this space should document their exact usage; this is the only fixed-address general purpose scratch space available for utility routines.

Specifically, the scratch RAM area consists exactly of the area used as the scratch math stack: 69 consecutive nibbles and three 5-nibble chunks punctuated by 11-nibble chunks which are UNAVAILABLE for use as scratch RAM:

```
SCRST0:          69 nibbles      (includes SCREX0)
(unavailable):   11 nibbles
SCREX1:           5 nibbles
(unavailable):   11 nibbles
SCREX2:           5 nibbles
(unavailable):   11 nibbles
SCREX3:           5 nibbles
```

3.3.27 Scratch Math Stack (SCRST0 - SCREXx)

The scratch math stack is a four-high stack for split (21-nibble) numerical values. The 21-nibble form consists of a sign nibble, a 15-nibble mantissa, and a five-nibble exponent. The signs and mantissas are stored consecutively in 64 nibbles starting at SCRST0. SCRST0 must reside between XXX00 and XXX0F in the RAM Map. Each exponent is stored 64 nibbles higher in memory than its corresponding mantissa:

```
                                                       SCRST0
                                                        (Low)

    +-+----------+-+----------+-+----------+-+----------+
  +--|S| Mantissa |S| Mantissa |S| Mantissa |S| Mantissa |
  |  +-+----------+-+----------+-+----------+-+----------+
  |
  +-------------------------------------------------------+
                                                          |
     +-----+-----+-----+-----+-----+-----+-----+-----+    |
     |     | Exp |     | Exp |     | Exp |     | Exp |<-+
     +-----+-----+-----+-----+-----+-----+-----+-----+
                 ^           ^           ^           ^
   (High)        |           |           |           |

             SCREX3      SCREX2      SCREX1      SCREX0
```

A pointer having possible values 0, 1, 2, or 3 points to the current top of the scratch math stack. This pointer is stored in the nibble with address SCRPTR.

3.3.28 DISP/PRINT RAM (SCROLT - EOLSTR)

SCROLT Number of 1/32s of a second to delay between scrolling characters in the display. Infinity is represented by FF. Initialized to 4.

DELAYT Number of 1/32s of a second to delay between scrolling lines in the display. Infinity is represented by FF. Initialized to 16 (10 hex).

NEEDSC 0 if no characters have been sent to display since last ENDLINE key, F otherwise. This keeps track of whether the display needs to be scrolled by calling SCRLLR.

DPOS DISPlay position. Used in DISP statements to keep track of current position in line. 0 means position 1.

DWIDTH DISPlay width. Used to limit number of characters output on any DISPlay line. Infinity is represented by 0. Initialized to 96 (60 hex).

PPOS PRINT position. Used in PRINT statement to keep track of the current position in the line. 0 means position 1.

PWIDTH PRINT width. Used to limit the number of characters output on any PRINT line. Infinity is represented by 0. Initialized to 96 (60 hex).

EOLLEN ENDLINE string length. Number of nibbles in the ENDLINE string. Should be 0,2,4,6. Initialized to 4.

EOLSTR ENDLINE string. Holds up to three characters which are sent to PRINTER IS device whenever an end-of-line is needed. Initialized to CR/LF.

3.3.29 CALL Parameter Count (PRMCNT)

PRMCNT is temporary scratch used by CALL execute to count the number of parameters.

3.3.30 Key Definition Info (DEFADR)

Eight nibbles of RAM used by the KEYRD subroutine for returning a pointer to a key definition. This ram is set by the key read routine (KEYRD). The contents DEFADR have the following definition:

(DEFADR): Length of definition string in bytes (2 nibbles).
(DEFADR+2): Key type: (1 nibble)
0 = Single ASCII character. Includes characters 0-31, which result from hitting special keys (ENDLN, UP-ARROW, etc.).
1 = ASCII control character. Must subtract 64 from the one-byte definition we are pointing to. These characters should be interpreted as text, and should not cause any special action in the editor.
2 = User-defined key--terminating.
4 = User-defined key--non-terminating.
6 = User-defined key--immediate execute.
8-F = Typing aid, with lower 3 bits as follows:
Bit 0: Parenthesis ("(") needs to be added to string.
Bit 1: Trailing space needs to be added to string.
Bit 2: Leading space needs to be added to string.
(Spaces and parenthesis not included in string length field or in definition proper. For example, the f shifted 4 key returns a definition which points to a 3 character string containing "SIN" and has the bit set which indicates that a parenthesis needs to be added to get the actual key definition ("SIN(").)
(DEFADR+3): Address of definition text. (5 nibbles)

3.3.31 Channel Number Save (CHN#SV)

The CHN#SV is used to hold the channel number currently being accessed. Refer to the section on the assign buffer in the "Table Formats" chapter for details.

3.3.32 Number of Command Stack Entries (MAXCMD)

MAXCMD holds the number of Command Stack entries.

3.3.33 Clock Speed (CSPEED)

Each time the system is reconfigured, the clock speed is recomputed and stored in CSPEED. The value is the clock speed divided by 16 (decimal) and stored in Hexadecimal (Hz).

3.3.34 HP-IL Special RAM (ERRLCH - HPSCRH)

ERRLCH Used by error routines; set when error occurs.

TERCHR Terminating character for ENTER and ENTER USING.

HPSCRH 7 nibbles reserved for HP-IL scratch.

3.3.35 Reserved RAM (RESRV)

96 nibbles (48 bytes) are reserved for future use. This memory will be allocated conservatively and through offical channels. Refer to the "HP-71 Resource Allocations" chapter for details on use of this RAM.

3.3.36 System RAM Availability

The following table summarizes which RAM locations may be used by the various routines of built-in and external (LEX file) keywords:

	nibbles avail.	Stmt. Parse	Stmt. Decomp	Stmt.** Exec.	Func. Exec.
SNAPBF	47	Yes	Yes	Yes	Yes
SCRTCH	64+4*5	Yes	Yes	Yes	Yes
Statement Scratch	42	No	Yes	Yes	No
Function Scratch	42	Yes	Yes	Yes	Yes
TRFMBF	60	No	No	Yes	Yes
LDCSPC	5	Yes	No	Yes	Yes
STSAVE	3	Yes	Yes	Yes	No
LEXPTR	5	No	Yes	Yes	Yes
RSTKBF Save Buffer	16*5	Yes	Yes	Yes*	Yes
RESERV Resreved RAM	***	***	***	***	***

* A statement cannot store anything in the RSTKBF area, call Expression Execution (EXPEXC), and expect what was saved to be intact.

** In general, any statement execution may use any memory available to function execution.

*** Reserved RAM may be used only after such usage has been registered and authorized by HP. See the chapter on "HP-71 Resource Allocation" for further information.

3.4 Configuration Buffer

The configuration buffers contain three tables, identifying what memory and I/O devices are configured where. The three tables contain information on System RAM (configuration table ID = FF), Other Memory (IRAM, ROM, EEPROM, etc.; configuration table ID = FE (bROMTB)), and Memory-Mapped I/O (HPIL mailbox, etc.; configuration table ID = FD).

Each table has a five nibble header. The first byte is the table ID (FF, FE or FD); the next three nibbles contain the table length, not including the header. The configuration buffer is terminated by a zero byte.

A configuration table entry is created in one of the three tables for every "sequence". A sequence consists of either:

1) A single memory-mapped I/O chip, or

2) One or more consecutive chips with identical ID's (bits 15-0 of ID) on a daisy chain.

A sequence is ended with:

1) Chip with different ID (which will be the start of a new sequence, obviously).

2) Chip with bit 18 of ID set (marks end of this sequence).

3) Chip with bit 19 of ID set (marks end of physical plug-in module).

A table entry conveys the following information:

Seq Position: Position of this sequence within the module. Since most modules have only one sequence, this is usually zero.

Device #: Position of this module within a consecutive series of modules (i.e., modules on same daisy chain). In the absence of a port extender, this will be zero. (The RAMs on the internal daisy chain may be grouped into logical modules.)

Port #: Identifies which daisy chain contains sequence. Port #0 is internal daisy chain (daisy-in on first chip thereof is tied high). Port #n is the daisy chain activated by output register bit #(n-1).

Size: Since size is always a power of two, the size is represented internally and on the chip ID as the one's complement of log2(size). Size refers to K-nibbles for memory devices and to words (hunks of 16 nibbles) for memory-mapped I/O.

Address: For memory devices, the upper 3 nibbles of the configuration address are given (the lower 2 are always zero). For memory-mapped I/O, the middle 3 nibbles are given (upper nibble is always 1, lower nibble is always 0).

Device type: Identifies type of memory device or if this is memory-mapped I/O device. The possible values are explained in the system configuration overview.

Device class: If sequence is memory-mapped I/O, this identifies which type of memory-mapped I/O device this. There is no device class for memory devices.

Chips in sequence: Identifies how many chips comprise this sequence. Kept in the table as (#chips - 1). Not kept for memory-mapped I/O, since it is always zero (each MM I/O chip results in its own table entry).

Reserved nibble: Nibble #1 from the Chip ID is saved here. That nibble is currently not defined.

Following is the exact format of the configuration buffer table entries:

```
          System RAM              Other Memory
          (cnftable ID FF)        (cnftable ID FE)
          ----------              ------------
NIB 0     Seq position            Seq position
NIB 1     Device #                Device #
NIB 2     Port #                  Port #
NIB 3     15-Log2(size) **        15-Log2(size)
NIB 4 /              *          /
NIB 5 |   Address (kbit)        | Address (kbit)
NIB 6 \                         \
NIB 7     0                       Device type
NIB 8     #chips/seq - 1          #chips/seq - 1
NIB 9     Nibble 1 from ID        Nibble 1 from ID
```

```
          Memory-mapped I/O
          (cnftable ID FD)
          -----------------
NIB 0     Sequence position in dev
NIB 1     Device #
NIB 2     Port #
NIB 3     15-Log2(size)
NIB 4 /
NIB 5 |   Address (words rel to 10000)
NIB 6 \
NIB 7     Device type  (always F)
NIB 8     Device class
NIB 9     Nibble 1 from ID
```

** FREEPORT routine may set this to zero to indicate that the RAM has been removed intentionally. This affects operation of this code in the spot where the old and new tables are compared to determine which RAMs are new and which are missing.

3.5 User Memory

User Memory consists of that portion of Main RAM which follows the Configuration Buffer. It contains the MAIN file chain, system buffers, CALC mode buffers, the command stack, the output buffer, available memory, and the various stacks maintained by the operating system.

3.5.1 MAIN File Chain

Files are stored in a linked list called a file chain. Each file in the chain is immediately preceded by a file header which contains identifying information about the file as well as a pointer to the file header of the next file in the chain. See the "File System" chapter for further information on the contents of a file header.

The start of the MAIN file chain is pointed to by MAINST. The pointer MAINEN, also known as IOBFST, points past the end of the chain, which is marked by a zero byte.

```
                              .                  .
                              . Configuration    .
                              . Buffers          .
                              .                  .
                              +------------------+
<Low>  MAINST ------->| File Header      |
                              | - - - - - - - - -|
                         +---| Offset to Next   |
                         |    +------------------+
                         |    |                  |
                         |    | File Contents    |
                         |    |                  |
                         |    +------------------+
                         +-->| File header      |
                              | - - - - - - - - -|
                         +---| Offset to Next   |
                         |    +------------------+
                         |    |                  |
                         |    | File Contents    |
                         |    |                  |
                         |    +--+---------------+
                         +-->          .
                                       .
                         +---          .
                         |             .
                         |    .+------------------+
                         +-->| File header      |
                              | - - - - - - - - -|
                         +---| Offset to Next   |
                         |    +------------------+
                         |    |                  |
                         |    | File Contents    |
                         |    |                  |
                         |    +--+---------------+
                         +-->|00|                      (00 byte indicates
                              +--+                      end of file chain)
<High>  MAINEN ------->.                  .
                              . System Buffers .
                              .                  .
```

3.5.2 Program Scope

The scope or bounds of the current program are defined by the program start and end pointers PRGMST and PRGMEN, respectively. Program scope may delimit a main program or a subprogram, which may be part of a larger file. Thr program end (PRGMEN) and current file end (CURREN) pointers are equal only when the current file contains a main program and no subprograms.

Note that the program scope pointers may delimit a program in a file that resides in the MAIN file chain, in a ROM, or in an Independent RAM, and therefore have no fixed relationship to the MAIN file chain pointers MAINST and MAINEN.

3.5.3 System Buffers

System buffers are used as general purpose buffers and as I/O buffers. They are maintained immediately following the end of the file chain. They are used for storage or working data and in some respects are more convenient than files for machine language applications. Each buffer is identified by a unique ID. ID's within a certain range are permanently reserved for use by specific applications and LEX files. Permanent ID reservations are assigned to software developers according to the procedures described in the "HP-71 Resource Allocation" chapter in this document. A certain range of ID's are also used and allocated on a temporary or scratch basis by the operating system, and are useful for applications where the temporary ID number can be saved.

There are several useful utilities related to system buffers, which are summarized in the "Utilities" chapter.

3.5.3.1 Format

Each buffer consists of a seven nibble header, followed by the buffer itself. The first nibble indicates if there are any address references in the beginning of the buffer that need to be updated by RFADJ (Reference Adjust); in most cases this nibble will be zero.

The next three nibbles are the buffer ID. The following three nibbles are the buffer length, that is the length of the buffer NOT including the buffer header (an empty buffer has 000 in this field).

The buffer chain is terminated by 0000.

The statement buffer (bSTMT) is always present and must be the first buffer in the buffer chain. This ensures that when executing statements from the statement buffer, PCADDR is not affected by buffer modifications.

Assuming the statement buffer (ID 801) is empty, the buffer chain is as follows:

SYSTEM BUFFER CHAIN

```
                         +---------------+      +---------------+
                         |               |      |               |
                         |               V      |               V
+-+---+---+-+--+---+--------+-----+-+--+---+--------+-----+----+
|0|108|000|#|ID|Len| buffer | ... |#|ID|Len| buffer | ... |0000|
+-+---+---+-+--+---+--------+-----+-+--+---+--------+-----+----+
^                                                             ^
|                                                             |
IOBFST                                                   IOBFEN
MAINEN                                                   CLCBFR
<Low>                                                    <High>
```

3.5.3.2 Update Addresses in System Buffers

If a buffer needs to have address references updated to reflect memory movement, then the first nibble of the buffer header is used. This nibble indicates the number of addresses to update (up to 15). The addresses must immediately follow the buffer header.

At the time a buffer is first created, this nibble is always initialized to zero. All of the System buffer utilities dealing with expanding and contracting existing buffers preserve this nibble. The buffer user is responsible for setting the nibble.

3.5.3.3 Automatic Deletion of System Buffers

Buffers are, by nature, temporary storage areas. Part of the system's maintenance process for buffers is deleting those which are no longer needed.

Whenever the configuration code is executed, all buffers are marked for deletion. The high bit of their buffer ID's is cleared; that is why all buffer ID's are >= 800H). Certain buffers are immediately reclaimed (the statement buffer, the FIB, etc.). Then the configuration poll is performed. All buffers which have not been reclaimed (high bit set) following this poll will then be deleted.

Anyone keeping buffers must reclaim them at every configuration poll (pCONF) or the buffers will go away. This can be done with the I/ORES utility which, given a buffer ID, will find the unreclaimed buffer and reclaim it by setting the high bit of the ID.

3.5.3.4 Permanent Buffers

Permanent buffers ID are allocated through official channels and are dedicated buffer to a particular application. Refer to the chapter on "HP-71 Resource Allocation".

3.5.3.5 Scratch Buffers

The system buffer ID range E00 to FFF is used for scratch buffers, which may be requested by calling IOFSCR, which allocates the next available scratch buffer and returns its ID. Scratch buffers are useful for temporary storage when the buffer ID can be easily saved by the user.

3.5.3.6 System Buffers Used by the Mainframe

The following is a list of system buffers used by the mainframe:

Alternate Character (bCHARS)
Assign (bASSGN)
Card Reader (bCARD)
External Command (bECOMD)
File Information (bFIB)
Immediate Execute Key(bIEXKY)
LEX Entry (bLEX)
Statement (bSTMT)
Statistics (bSTAT)
Startup (bSTART)

The index indicates where more information can be found about these buffers.

3.5.4 CALC Mode Pointers

When CALC mode is in effect, the pointers AVMEMS and AVMEME, which control available memory, are given unusual meanings. They act in coordination with the other CALC mode pointers as described in this section.

The CALC mode pointers define several volatile areas between CLCBFR (which is the beginning of the Command Stack) and FORSTK. Characters accepted by CALC mode are inserted at RAWBFR (which stands for raw input buffer), while the parsing process operates at and advances RFNBFR (refined input buffer).

Anticipated right delimiters, such as commas and right parentheses, are inserted by the parser to the right of RAWBFR. Tokens compiled by CALC mode are appended to the buffer between CLCSTK and SYSEN. The intermediate parse stack resides between AVMEMS and MTHSTK, and intermediate operands reside between MTHSTK and FORSTK.

During most of the parsing operation, system free space is actually between SYSEN and AVMEMS, as shown:

```
                              +----------------------+
 <Low>                        |    System Buffers    |
          CLCBFR -->          +----------------------+
                              |    Command Stack     |
                              |- - - - - - - - - - - |
                              |  CALC Refined Buffer |
          RFNBFR -->          +----------------------+
                              | CALC Left Raw Buffer |
          RAWBFR -->          +----------------------+
                              | CALC Right Raw Buffer|
          CLCSTK -->          +----------------------+
                              | CALC Token Stream    |
  SYSEN = OUTBS   -->         +----------------------+
                              |     Available Mem    |
                              |          |           |
                              |          v           |
          AVMEMS -->          +----------------------+
                              |     Intermediate     |
                              |     Parse  Stack     |
          MTHSTK -->          +----------------------+
                              |     Intermediate     |
                              |       Operands       |
          FORSTK -->          +----------------------+
                              |    FOR/NEXT Stack    |
 <High>                       +----------------------+
```

When the tokens are to be executed, the parse stack is moved to the end of the compiled token stream, so that the top of the Math Stack is free and AVMEMS can assume its normal meaning. When a CALC mode statement is complete, it is already within the Command Stack.

3.5.5 Command Stack

The Command Stack is a doubly linked list of buffers between the CLCBFR and RFNBFR. Outside of CALC mode, SYSEN, CLCSTK, and RAWBFR are equal to RFNBFR.

The Command Stack is initialized to have 5 entries, each containing only a carriage return (ASCII 13). Each entry consists of a 3 nibble length field, command text and a 3 nibble backwards chaining length field.

The first length field is the number of nibbles in the actual text of the command, including the carriage return at the end of the text. The command text is always terminated with a carriage return. The second length field is three nibbles greater that the length of the text to allow chaining backwards through the Command Stack.

The number of entries in the Command Stack is kept in the RAM nibble called MAXCMD. This nibble must correspond to the actual number of entries in the stack. To change the number of entries in the Command Stack, this nibble must be changed as well as creating/deleting entries on the stack to match this number. The MAXCMD nibble is the number of entries minus one; thus the Command Stack can be altered to have from 1 to 16 entries. No mechanism in the mainframe is provided to do this.

COMMAND STACK

```
     <Low>
                               |                      |
     CLCBFR -------+--> +----------------------+
                   |    |        Len(5)        | ----+
                   |    +----------------------+     |
                   |    |                      |     |
                   |    |       Text of        |     |
                   |    |      Command 5       |     |
                   |    |                      |     |
                   |    |    CR (ASCII 13)     |     |
                   |    +----------------------+ <---+
                   +--- |       Len(5)+3       |
              +--> +----------------------+
              |         |        Len(4)        | ----+
              |         +----------------------+     |
              |         |                      |     |
              |         |       Text of        |     |
              |         |      Command 4       |     |
              |         |                      |     |
              |         |    CR (ASCII 13)     |     |
              |         +----------------------+ <---+
              +-------- |       Len(4)+3       |
                        +----------------------+
                        |        Len(3)        |
                        +----------------------+
                        |                      |
                        |          .           |
                                    .
                                      .
                        |                      |
                        |                      |
                  +---> +----------------------+
                  |     |        Len(1)        | ----+
                  |     +----------------------+     |
                  |     |                      |     |
                  |     |       Text of        |     |
                  |     |      Command 1       |     |
                  |     |                      |     |
      RFNBFR      |     |    CR (ASCII 13)     |     |
    = RAUBFR      |     +----------------------+ <---+
    = CLCSTK      +---- |       Len(1)+3       |
    = SYSEN ----------> +----------------------+
                        |                      |
     <High>             |                      |
```

3.5.6 Available Memory

The SYSEN pointer separates the CALC Mode token stream from the temporary input buffer area for BASIC. SYSEN is used by TRANSFORM to mark the beginning of memory available for its input and output buffers.

OUTBS points to the start of the output buffer, used to compile BASIC tokens during statement parse and to regenerate text for statement decompile.

AVMEMS marks the end of the output buffer and the start of available memory. This delimiter is necessary before moving the output buffer to the statement buffer, a program file or to the display buffer.

After statement parse or decompile, the main loop collapses available memory start (AVMEMS), the output buffer (OUTBS) and the system RAM end pointer (SYSEN) to the end of the Command Stack (CLCSTK+RFNBFR).

During statement execution, available memory start is at the end of the Command Stack.

3.5.7 Math Stack

The Math Stack exists between MTHSTK and FORSTK; it is used for intermediate storage of operands during expression execution. Four types of objects are recognized on the Math Stack: real numbers, complex numbers, strings, and array dope vectors. The stack grows from high addresses to low. The pointer, MTHSTK, points to the top of the Math Stack.

Refer to the "Statement Parse, Decompile, and Execution" chapter for details on expression execution and the Math Stack.

3.5.8 Save Stack

The Save Stack is an area of user memory for saving special system information. It resides between the Math Stack and FOR/NEXT Stack, as shown below.

Any new Save Stack allocation is inserted between the current Save Stack contents and the FOR/NEXT Stack. Therefore, unlike the Math Stack or FOR/NEXT stack, the top of the Save Stack is at a higher address than its bottom. The SAVSTK pointer is always positioned past the highest-addressed nibble of the most recently allocated section of Save Stack memory, and is therefore identical to the FORSTK pointer (for which SAVSTK is merely another name). Note that there is no pointer which explicitly marks the bounds between the Math Stack and the Save Stack:

```
<Low>        |                      |
             |   Available Memory   |
             |                      |
  AVMEME --> +----------------------+
 = MTHSTK    |                      | Newest Math Stack Entry
             |      Math Stack      |
             |                      | Oldest Math Stack Entry
             +----------------------+
             |                      | Oldest Save Stack Entry
             |      Save Stack      |
             |                      | Newest Save Stack Entry
  SAVSTK --> +----------------------+
 = FORSTK    |                      | Newest FOR/NEXT Entry
             |    FOR/NEXT Stack    |
             |                      | Oldest FOR/NEXT Entry
<High>       +----------------------+
```

The routine SALLOC will expand the Save Stack by the requested number of nibbles. The memory between available memory end and the end of the Save Stack (between system pointers AVMEME and SAVSTK) is moved down into available memory by the required number of nibbles, and AVMEME is updated accordingly. Since this process preserves all memory between AVMEME and SAVSTK but overwrites the memory immediately before AVMEME, AVMEME must be set to the true top of the Math Stack in order for the Math Stack to be preserved.

Routines which allocate memory recursively on the Save Stack are responsible for removing that memory. The routine "SRLEAS" deletes the requested number of nibbles from the Save Stack and adjusts pointers.

At the end of every statement execution, the available memory end pointer AVMEME is reset to the top of the FOR/NEXT Stack, thereby collapsing the Math Stack and the Save Stack.

The Save Stack is used by POLL to save polling information. It is also used by COPY, TRANSFORM and RUN to save source and destination file information.

3.5.9 FOR/NEXT Stack

At the time a FOR statement executes, information is pushed on the FOR/NEXT Stack. This stack is referenced and/or altered any time a FOR or NEXT statement is encountered.

```
  <Low>       +----------------------+
              |   Return Address     |    5 nibbles
              +----------------------+
              |   Step Value         |   16 nibbles
              +----------------------+
              |   Limit              |   16 nibbles
              +----------------------+
              | Encoding of Var Name |    4 nibbles
 <High>       +----------------------+
```

The encoding of the variable name depends on whether the variable is alpha-digit or not. In the case of an alpha variable, the low byte is the ASCII letter and the following byte is zeroes; for alpha-digit variables, the low byte is the alpha-digit token and the following byte is the ASCII letter. The alpha-digit token has 6 in the high nibble, and the digit in the low nibble.

3.5.10 GOSUB Stack

The GOSUB Stack resides between the FOR/NEXT Stack and the active variable space. The pointer GSBSTK points to the top of the GOSUB Stack. The GOSUB Stack is typically used to save return addresses, such as the return address of a call to a subroutine, but may also be used to store other addresses and indicators.

Associated with each address on the GOSUB Stack is a return type nibble.

```
  <Low>       +-------------+
              | Return Type |              1 nibble
              +-------------+--------+
              | Return Address       |     5 nibbles
 <High>       +----------------------+
```

The return type encoding is:

0	Return to Program
1	Return to Keyboard
2	ON TIMER#1 ... GOSUB
3	ON TIMER#2 ... GOSUB
4	ON TIMER#3 ... GOSUB
8	Machine Code Return
9-E	Special Return Types: Future statement extensions
F	Update Address (Nonzero) or Boundary Address (Zero)

Return to program is the standard GOSUB from within a BASIC program.

Return to keyboard is a GOSUB initiated from the keyboard. The statement buffer is collapsed before returning to the keyboard.

Return from an ON TIMER, return type 2-4, reactivates the appropriate timer before returning to the statement following the GOSUB within a program.

Machine code return is a return to a binary program that called a BASIC program. The routine "PSHMCR" pushes the passed return address on the GOSUB Stack and tags it as a machine code return. The routine "POPGSB" pops an address and return type off the GOSUB Stack.

Special return types: 9-E are available for future statements or statement extensions needing special processing on return from a GOSUB. An example is ON TIMER...GOSUB needing to reactivate the timer before returning. The RETURN statement polls on special return type (pRTNTp) if within the range of 9-E.

A nonzero address of return type "F" indicates an update address. The system will not return control to an update address, but will update the address whenever memory moves. This is a convenient place to store pointers to segments of memory which may move. The routine "PSHUPD" pushes the passed address onto the GOSUB Stack and tags it as an update address. The routine "POPUPD" pops an address and return type off the GOSUB Stack. If an update address is encountered during RETURN execution, it is not popped off and the error "RTN w/o GOSUB" is generated.

A zero address of return type "F" indicates an environment boundary, however. Such an address marks the end of the environment for a user-defined function. If a RETURN statement is encountered and the end of an environment is reached, the error "RTN w/o GOSUB" is generated. The boundary mark is not popped off the GOSUB Stack.

3.5.11 Variable Storage

Variables are kept in memory immediately above (higher address) the GOSUB Stack. Currently active variables exist between the pointers ACTIVE and CALSTK. A complete description of this area is in the "Internal Data Representation" chapter.

3.5.12 User-Defined Function Environment Stacking

When a user-defined function is called, a portion of the local environment is saved in an Environment Save Block which is placed on the CALL stack in much the same manner as the local environment is saved when the CALL statement is executed.

The following diagram shows the structure of memory immediately after a user-defined function has been called:

```
<Low>
   |                               | <== New MTHSTK  \
   |                               | <== New FORSTK   +-Same value
   +-------------------------------+ <== New GSBSTK  /  initially
   |  F00000 GOSUB Stack Boundary  |
   +-------------------------------+ <== New CALSTK
   |     User-Defined Function     |
   |     Environment Save Block    |
   +-------------------------------+
   |      Extended Parameter       |
   |            Storage            |
   +-------------------------------+ <== PRMPTR
   |  Last Parameter of Function   |
   +-------------------------------+
   |               .               |
   |               .               |
   |               .               |
   +-------------------------------+
   | First Parameter of Function   |
   +-------------------------------+
   |        Function Value         |
   +-------------------------------+ <== Old MTHSTK (value before
   |                               |       the user-defined
   |                               |       function is called)
   |                               |
   +-------------------------------+ <== Old FORSTK
   |                               |
   +-------------------------------+ <== Old GSBSTK
   |                               |
   +-------------------------------+ <== ACTIVE
   |                               |
<High>
```

3.5.12.1 Environment Save Block

The User-defined Function Environment Save Block is located after the end of the GOSUB Stack (which is marked by F00000). It contains the following data:

ENVIRONMENT SAVE AREA

USER-DEFINED FUNCTION SAVE BLOCK FORMAT

Return address	5 nibbles	-+ These pointers
PCADDR saved	5	| are adjusted
STMTD0 saved	5	| when memory
3 hardware return addresses	15	-+ moves.
STMTD1 saved	5	
STMTR0 saved	16	
STMTR1 saved	16	
Offset to previous MTHSTK	5	
Offset to previous FORSTK	5	
Offset to previous GSBSTK	5	
Previous parameter count	2	
Offset to previous PRMPTR+2	5	
STSAVE saved	3	
CHN#SV	2	
Return type	1	

Return address - Continue execution address when ENDDEF is executed.

PCADDR,STMTD0 - Updated when memory moves.

Hardware return stack,addresses - Three addresses will be popped off the hardware return stack and saved. This means if an assembly routine calls the expression execution routine, only the last three return addresses in the hardware return stack will be preserved.

STMTD1 - This saved pointer will be adjusted when a new variable is created while executing a user-defined function.

STMTR0 - This is the same as S-R0-0 ... S-R0-3. If the first five nibbles of STMTR0(S-R0-0) contain a memory address (>10000 Hex) and the first hardware return address saved is =STORE, S-R0-0 will be adjusted when a new variable is created.

STMTR1 - This is the same as S-R1-0 ... S-R1-3.

Offset to previous MTHSTK,.. PRMPTR+2 - These pointers are saved as relative addresses. Adding the offset to where it is saved points to the previous pointer.

Return type - 0 : User-defined function is called from a program statement.
1 : User-defined function is called from a keyboard expression.
8 : User-defined function is called by a Binary routine.

3.5.12.2 Extended Parameter Storage

The value of string or complex parameters is stored in this area. The extended value is pointed to by the parameter value.

3.5.13 Subprogram CALL Environment Stacking

When a subprogram is called, a new local environment must be created. Before this can happen, the old calling environment must be saved by "pushing" it onto the CALL Stack. The process is performed in three steps.

First, an area is opened immediately before the current FOR/NEXT Stack to hold information blocks which contain pointers and other data about the current environment. The operating system writes one save block and then issues a poll to allow any LEX files present to add other blocks. This area is called the Environment Save Area, and is described below. It is also referred to as the Subprogram Save Stack.

Next, the current environment is "pushed" onto the CALL Stack by adjusting the pointer CALSTK to the start of the newly created Environment Save Area.

Finally, the new local environment is created and the pointers ACTIVE, GSBSTK, FORSTK, and MTHSTK are adjusted as shown below. The initial active variables are the parameters passed to the subprogram.

```
 <Low>               |               |
                     |               | <== New MTHSTK \
                +--  +---------------+ <== New FORSTK  \ Initially
  New           |    |               | <== New GSBSTK  / same value
  Local         |    | - - - - - - - | <== New ACTIVE /
  Environment   |    |  (parameters) |
                +--  +---------------+ <== New CALSTK
                |    |  Environment  |
                |    |     Save      |
  Stacked       |    |     Area      |
  Environment   |    +---------------+ <== Old FORSTK
                |    |               |
                |    |    CALLing    | <== Old GSBSTK
                |    |               |
                |    |  Environment  | <== Old ACTIVE
                |    |               |
                +--  +---------------+ <== Old CALSTK
 <High>              |               |
```

Each CALL statement adds a level to the CALL Stack by saving the current environment and each END SUB removes a level from the CALL Stack by restoring the previous environment.

The CALL Stack is bounded by the CALSTK and RAMEND pointers (when CALSTK equals RAMEND there are no saved environments).

3.5.13.1 Environment Save Area

The execution of CALL stacks more than just the GOSUB Stack, the FOR/NEXT Stack and the local variables. It creates an area below (in lower memory) the FOR/NEXT Stack to hold information about the environment which is being suspended. This area is called the Environment Save Area or the Subprogram Save Stack.

It is filled by a linked list of information blocks called Environment Save Blocks. Each block may contain a list of addresses to be updated when memory moves, as well as other data. The block begins with a 2 nibble ID followed by a 5 nibble link field which points to the next block in the list. This is followed by a 1 nibble field specifying a number (0 to 15) of 5 nibble update addresses (which will be updated when memory moves), and then that number of update addresses. Any remaining area in the block may be used for arbitrary data and is not updated.

The first save block is created by the mainframe CALL statement. Its ID is 00, and marks the end of the linked list. This block is always 89 nibbles in total length.

At CALLing time, after the mainframe creates its save block, it polls (pCALSV) to give LEX files a chance to add a save block to this area. Each poll handler that has anything to save is expected to create another block (growing into available memory) in the same format.

The save block created by the mainframe has the following contents:

ENVIRONMENT SAVE AREA

MAINFRAME SAVE BLOCK FORMAT

```
<Low>
                                 LEX ID (00)   2 nibbles
                         Entry length (04F)   3
   Number of addresses to update (A)   1          --+
                         +--  CURRST saved   5            |
                         |    PRGMST saved   5            |
                         |    PRGMEN saved   5            |
        Addresses        |    CURREN saved   5            |
         updated         |    PCADDR saved   5            |
       when memory       |    CNTADR saved   5            |
          moves          |    ERRSUB saved   5            |
                         |    ERRADR saved   5            |  84 nibs
                         |    ONINTR saved   5            |   = 04F hex
                         +--  DATPTR saved   5            |
            +-- Offset to previous FORSTK   5            |
            |   Offset to previous GSBSTK   5            |
    Misc.   |   Offset to previous ACTIVE   5            |
    Info    |   Offset to previous CALSTK   5            |
            |       Parameter count saved   2            |
            | Offset to previous PRMPTR+2   5            |
            +--                Return type   1          --+
<High>
```

LEX ID

For the block created by the mainframe this field is 00. This indicates the end of the linked list and that the suspended FOR/STACK, GOSUB Stack and variables follow immediately. For blocks created by lex files, this field should be filled in with the LEX ID of the file creating it. It serves as a tag field to identify the block later when the return from subprogram causes the Restore CALLing Environment poll (pCALRS).

Entry Length

This field is always 84 (04F hex) for the block created by the mainframe. This number includes everything in the block starting from the next nibble (the update address count nibble) to the end of the block (the return type nibble). This length does not include the LEX ID field or the entry length field itself.

Number of Addresses to Update
For the mainframe, this nibble is always 10 (A hex), reflecting the number of following pointers that require updating when memory moves. Blocks created by LEX files may have from 0 to 15 addresses updated.

Addresses to be Updated
The previous field specifies how many 5 nibble addresses are included here. The 10 address fields in the mainframe block are used to save the following memory pointers for restoration later: CURRST, PRGMST, PRGMEN, CURREN, PCADDR, CNTADR, ERRSUB, ERRADR, ERRSUB, ERRADR, ONINTR and DATPTR. Whenever program memory moves, these addresses stored here will be updated to reflect the new address of the thing they point to.

Miscellaneous Information
After the addresses to be updated described above, the remainder of the block has a format specified individually for that type of block. The block created by the mainframe has the following fields:

Offset to prevdious FORSTK ... CALSTK
These pointers of the calling program environment are saved as relative addresses. Adding the offset to where it is saved points to the previous pointer.

Parameter Count
One byte field. If zero then currently not in a user-defined function; if nonzero, then represents parameter count - 1 of the user-defined function.

PRMPTR
This is a 5 nibble pointer to the first parameter in the user-defined function's parameter chain.

Return type
If =0, CALL is from a BASIC program.
If =1, CALL is from a Binary program.

3.6 Plug-in ROM and Independent RAM

The format of a plug-in ROM module is the same as for a RAM module configured as an Independent RAM, with the exception of the first eight nibbles of the module which contain the Stand Alone Module ID. Either form of plug-in memory module contains a file chain, starting in the ninth nibble, that is identical in format to the MAIN file chain.

Throughout the following discussion, the term ROM will be used as a general name for a stand alone memory module, whether it be a plug-in ROM module or an Independent RAM.

3.6.1 Standard Configuration

The general format of every stand alone memory module is as follows:

```
+-------------------------+  <--- Module Start
| Stand Alone Module ID   |
+-------------------------+  <--- Module Start + 8
|                         |
|                         |
|       File Chain        |
|                         |
|                         |
+ - - - +-----------------+
|00 byte|                 .  <--- 00 byte ends chain
+-------+                 .
.                         .
.      Unused Space       .
.                         .
...........................  <--- End of Module
```

3.6.2 Stand Alone Module ID

The Stand Alone Module ID field is used to distinguish an Independent RAM from other forms of memory modules. For Independent RAMs, this field has the hex value B3DDDDDE (the B is in the lowest-addressed nibble of the module). For ROMs and all other forms of memory modules, this field may have any value except the IRAM value.

3.6.3 File Chain Layout

Each file entry in the chain begins with a file header which contains the file name and other identifying information about the file. The format of the file header is the same as that used in the MAIN file chain, and is described in the "File System" chapter. As in the MAIN file chain, a stand alone module file chain is terminated by a zero byte in the first character of a file header name field.

```
          +----------------+  <--- Module Start
          | Stand  Alone   |
          |  Module  ID    |
          +----------------+  <--- Start of File Chain
          | File  Header   |       (Module Start + 8)
          |                |
          | - - - - - - - -|
      +---| Offset to Next |
      |   +----------------+
      |   |                |
      |   | File  Contents |
      |   |                |
      |   +----------------+
      +-->| File  Header   |
          |                |
          |- - - - - - - - |
      +---| Offset to Next |
      |   +----------------+
      |   |                |
      |   | File Contents  |
      |   |                |
      |   +--+-------------+
      +-->|00| <----------------- 00 byte ends chain
          +--+
```

3.6.4 Take Over ROM

Take-over ROMs come in two flavors: soft-configured and hard-configured.

3.6.4.1 Hard-Configured Takeover ROM

A hard-configured take-over ROM must be plugged into port 1, where, by virtue of shorting certain lines together, it will disable the system ROMS. This ROM should be hard-configured in the address space occupied by the HP-71 system ROMs, as it is replacing them.

A problem occurs when installing such a ROM: where is the CPU's program counter? This is a problem when 1) the takeover ROM is plugged in, and must resume execution from the HP-71 ROM, and 2) the takeover ROM is unplugged, and HP-71 must resume execution. It is virtually impossible for HP-71 to guarantee the position of the PC, except during deep sleep. During deep sleep, the PC spends most of its time pointing just past the SHUTDN in the deep sleep routine. However, the processor does occasionally wake up to process clock system requests and whatever else may request service.

If the hard-configured takeover ROM uses memory in such a way that it is incompatible with the HP-71 operating system, the ROM should perform its own version of cold start when it is plugged in and unplugging it should force the HP-71's built in operating system to perform a coldstart.

A few simple rules will facilitate this:

1) HP-71 should be turned off when plugging in a hard-configured takeover ROM.

2) The takeover ROM should expect control to be passed to it at the address just past HP-71's deepsleep SHUTDN (address = 5E2). This is where the PC is most likely to be.

3) The takeover ROM should be at a shutdown when unplugging it.

4) The takeover ROM shutdown should position the PC at the HP-71 coldstart code (label CLDST1).

5) The takeover ROM should use a different CMOS testword from HP-71, this will cause the built in operating system to coldstart as soon as it is reenabled (at time of next interrupt). In general, the CMOS test word should be unique for each take over ROM and should be used to determine if memory is "okay" for that particular hard configured take over ROM.

It is conceivable that a hard configured takeover ROM might be made compatible with the built-in operating system so that is may be plugged in or removed without loss of memory contents. In this case, the ROM should use the same CMOS test word as the built in operating system.

3.6.4.2 Soft-Configured Takeover ROM

A soft-configured takeover ROM avoids many of the problems of a hard-configured ROM. It is useful for adding subsystems to the HP-71, such as a pocket secretary. It can simply grab control of the machine at an appropriate time, such as Wake-up poll or Powerdown poll. This is essentially a mode, not a new machine.

In general, a soft-configured takeover ROM should not mess with HP-71 operating system RAM. It is an extension of HP-71, and more than likely is interacting with HP-71 code in the system ROMs.

A major limitation of soft-configured take-over ROMs is that it is very difficult for them to change the system's configuration. Doing a bus reset (unconfigure all chips) will unconfigure itself, making it impossible to execute any more code from the ROM. A soft-configured ROM, barring some very clever programming, will have to live with the HP-71 system configuration.

3.7 Available Memory Management

The term "available memory" refers to the area of RAM between the boundaries pointed to by AVMEMS (available memory start) and AVMEME (available memory end). This region supplies the memory for new allocations on the various system stacks, which cause AVMEME to grow toward AVMEMS. This region also supplies the memory for the system's output buffer, which is used to hold the tokens output by the parsing process and for various other system functions which cause AVMEMS to grow toward AVMEME.

In addition, activities which increase the size of the main RAM file chain (such as creating or enlarging a file in the chain), the size of the system buffer area (creating or enlarging a system buffer), or the size of the Command Stack, will also cause AVMEMS to grow toward AVMEME.

A minimum amount of available memory is therefore necessary for the operating system to function. This minimum amount is 106 bytes, and is referred of as LEEWAY, which is a globally defined symbol in the operating system equate file (see file TI&EQU in Volume III of this document).

Whenever an operation system activity must consume available memory, a check is performed according to the following conventions:

* If the memory allocation is permanent (that is, after the activity is completed, the memory will remain allocated), then available memory must not dip below LEEWAY. Examples of permanent allocations are creating a system buffer, creating a variable, adding to the GOSUB Stack, FOR/NEXT Stack, or the CALL Stack.

* If the memory allocation is temporary (that is, after the activity is completed, the memory will be released), then available memory may dip below LEEWAY. Examples of temporary allocations are: parsing or decompiling into the output buffer, expression evaluation using the Math Stack, preparing messages for display, or issuing a poll (which saves 31 bytes on the SAVSTK).

When an insufficient memory condition has been detected and reported, the user must be able to perform certain commands, such as CAT, PURGE, COPY or END, in order to release memory in a safe manner so that the system is again usable.

To allow these activies to occur during low memory, the following special cases of LEEWAY checking have been implemented:

* When a command is added to the Command Stack that causes a dip below LEEWAY, previous commands will be crushed to null, starting with the oldest, until LEEWAY is reached or only 1 command remains.
* When the statement buffer is expanded to accept the tokenized statement, LEEWAY is not checked.
* Leeway in not checked when COPY saves its file info on the Save Stack.
* The poll routine does not check LEEWAY when saving poll info on the Save Stack.

The value of of LEEWAY has been set to allow a file to be copied to an external device. This requires the following amount of memory:

Command Stack to enter COPY command	25 bytes
To move tokenized COPY statement into statement buffer	25 bytes
Save COPY file info on the Save Stack	25 bytes
Issue COPY poll to external device	31 bytes

LEEWAY =	106 bytes

If a LEX file or other user-supplied code causes the memory available to the operating system to shrink below this minimum, catastrophic failure may occur. For example, if available memory has shrunk so far below LEEWAY that the error message handling routines do not have enough room to build the "Insufficient Memory" error message, the system will loop infinitely attempting to process the message.

See the "Message Handling" chapter for a discussion of the chapter discusses the MEMCKL utility which checks available memory with or without LEEWAY.

3.8 Handling Memory Movement

Whenever file memory is moved due to adding data to or deleting data from the MAIN file chain or an IRAM file chain, the various system pointers which reference the file system and neighboring areas of memory may need to be adjusted. RFADJ is the utility called after such a memory move, to examine these pointers and make the necessary adjustments. There are two major routines which make up RFADJ: RFADJ- (used when memory moves to lower addresses, as with a PURGE of a file [MOVEMU called] and RFADJ+ (used when memory moves to higher addresses [MOVEMD called]).

Entry conditions parallel requirements for calling MOVEUx and MOVEDx (move memory routines): Begin Source, Begin Destination, and End Source, are referred to in this context. Note that the End Source address is the address of the nibble that immediately follows the last nibble in the source block. Therefore, the source block is null when Begin Source equals End Source.

B(A) is assumed to be an offset: Begin source - Begin destination.

Algorithms:

```
  RFADJ- :  Save begin source in R0
  RFAD-- :  Position D1 at AVMEMS ram location
```

The following entry point can be used by memory movement on plug-ins. It assumes D1 is positioned at a ram location which contains 'AVMEMS' of that plug-in, i.e., the address after the last file in the chain.

```
  RFAD-I :  Save begin destination in R1 (R0+B)
            D(S) <-- 1          (flags which way mem is moving)
            Call RFAD58         (Updates addresses on FOR and
                                 GOSUB Stacks)
            Call RFAD97         (Updates addresses in RAM locations
                                 PCADDR-->TMRAD3 - zeroes out those
                                 referencing purged address space)
            Goto PCUPD+

  RFADJ+ :  Save begin source in R0
  RFAD++ :  Position D1 at AVMEMS ram location

  RFAD+I :  D(S) <-- 0          (flags which way mem is moving)
            Call RFAD58         (Updates addresses on FOR and
                                 GOSUB Stacks)
            Call RFAD86         (Updates addresses in RAM locations
                                 PCADDR-->TMRAD3)
  PCUPD+ :  Updates CURRST-->AVMEMS
            .
            .
            .
  PCUPDT :  .
```

Address updating:

```
    If address < End Source
       THEN If address >= Begin Source
         THEN update (add offset).

 Address zeroing:  (Done only if D(S)#0)
    If address < Begin Source
       THEN If address >= Begin Destination
         THEN zero it.
```

The following references are NEVER zeroed:

1) Addresses on FOR/NEXT Stack
2) CURRST-->AVMEMS

3.8.1 In Configuration Buffer Area

Configuration buffers are only manipulated during execution of the configuration code. Following is a summary of the effects of configuration buffer manipulation on various system pointers.

HP-71 REFERENCE ADJUSTMENTS -- CONFIGURATION BUFFERS

B ::= Updated only if Begin Source <= address < Begin Dest

A ::= Updated only if Begin Source < address < Begin Dest

U ::= Unconditionally updated (offset always added to pointer)

Z ::= Address set to 0 if Begin Dest <= address < Begin Source

* ::= Not updated

Actions:

- Create ::= Item created
- Expand ::= Buffer expands
- Contract ::= Buffer shrinks

System Pointers	ACTION ON CONFIGURATION BUFFERS: Create	Expand	Contract
MAINST - MAIN File Chain Start	U	U	U
CURRST - Current File Start	B	B	B
PRGMST - Current Program Start	B	B	B
PRGMEN - Current Program End	B	B	B
CURREN - Current File End	B	B	B
MAINEN - MAIN File Chain End	B	B	B
CLCBFR - CALC Mode Buffer Start	B	B	B
RFNBFR - CALC Mode Refined Buffer	B	B	B
RAWBFR - CALC Mode Raw Buffer	B	B	B
CLCSTK - CALC Mode Token Stack	B	B	B
SYSEN - System RAM End	B	B	B
OUTBS - Output Buffer Start	B	B	B
AVMEMS - Available Memory Start	B	B	B
AVMEME - Top Math Stack	*	*	*
FORSTK - Top FOR/NEXT Stack	*	*	*
GSBSTK - Top GOSUB Stack	*	*	*
ACTIVE - Active Variable Pointer	*	*	*
CALSTK - CALL Stack	*	*	*
RAMEND - User RAM End	*	*	*

Pointers in System Buffers :			
LEX BUFFER Pointers	B	B	B
FIB: File Begin Field	B	B	B
FIB: Data Start Field	B	B	B
Pointers Within Environments:			
FOR/NEXT Stack Addresses	B	B	B
GOSUB Stack Update Addresses	B	B	B
Miscellaneous Pointers:			
PCADDR - Program Ctr at Stmt len	B	B	B
CNTADR - Continue Address	B	B	B
ERRSUB - ON ERROR-GOSUB Rtn Addr	B	B	B
ERRADR - ON ERROR Statement Addr	B	B	B
ONINTR - ON INTRPT Statement Addr	B	B	B
DATPTR - DATA Statement Pointer	B	B	B
TMRAD1 - ON TIMER#1 Statement Addr	B	B	B
TMRAD2 - ON TIMER#2 Statement Addr	B	B	B
TMRAD3 - ON TIMER#3 Statement Addr	B	B	B
Note that these are NEVER UPDATED:			
INBS - Input buffer start	*	*	*
SNAPBF - Snapshot Buffer Addresses	*	*	*
RSTKBF - Rtn Stack Save Buf Addrs	*	*	*

3.8.2 In a File Chain

When file memory moves, system pointers such as CURRST may need to be adjusted. In this case the routine RFADJ (Reference Adjust) must be called to handle the updating of all of these pointers. This routine examines each pointer to determine whether or not it was affected by the memory move; all affected pointers are updated.

RFADJ examines pointers DSPCHX through TMRAD3, CURRST through AVMEMS, all pointers in FIB's, and pointers on the FOR/NEXT Stack, GOSUB Stack, and CALL Stack. Pointers which reference purged address space are zeroed out (this does not include any pointer which pointed at the begin destination of the memory move - For example, if the file following the current file was purged, CURREN would NOT be zeroed out).

When files move to a lower address (as when a file is purged), RFADJ- is called; if files are on a plug-in, RFAD-I is the entry point to use. When files move to a higher address (as when a file expands), RFADJ+ is called; if files are on a plug-in, RFAD+I is the entry point to use.

HP-71 REFERENCE ADJUSTMENTS -- FILE MEMORY MOVES

B ::= Updated only if Begin Source <= address < Begin Dest

A ::= Updated only if Begin Source < address < Begin Dest

U ::= Unconditionally updated (offset always added to pointer)

Z ::= Address set to 0 if Begin Dest <= address < Begin Source

* ::= Not updated

Actions:

Create ::= Item created

Purge ::= Item purged

At end ::= Movement at end

Within ::= Item grows/shrinks in the middle

System Pointers:	ACTION ON FILE IN MAINFRAME				ACTION ON FILE IN IRAM			
	Create	Purge	At end	Within	Create	Purge	At end	Within
MAINST - MAIN File Chain Start	*	*	*	*	*	*	*	*
CURRST - Current File Start	*	B	B	B	*	B	B	B
PRGMST - Current Program Start	*	B	B	B	*	B	B	B
PRGMEN - Current Program End	*	B	B	B	*	B	B	B
CURREN - Current File End	*	B	B	B	*	B	B	B
MAINEN - MAIN File Chain End	U	U	U	U	*	*	*	*
CLCBFR - CALC Mode Buffer Start	U	U	U	U	*	*	*	*
RFNBFR - CALC Mode Refined Buff	U	U	U	U	*	*	*	*
RAWBFR - CALC Mode Raw Buffer	U	U	U	U	*	*	*	*

CLCSTK - CALC Mode Token Stack	U	U	U	U	*	*	*	*
SYSEN - System RAM End	U	U	U	U	*	*	*	*
OUTBS - Output Buffer Start	U	U	U	U	*	*	*	*
AVMEMS - Available Memory Start	U	U	U	U	*	*	*	*
AVMEME - Top Math Stack	*	*	*	*	*	*	*	*
FORSTK - Top FOR/NEXT Stack	*	*	*	*	*	*	*	*
GSBSTK - Top GOSUB Stack	*	*	*	*	*	*	*	*
ACTIVE - Active Variable Pointer	*	*	*	*	*	*	*	*
CALSTK - CALL Stack	*	*	*	*	*	*	*	*
RAMEND - User RAM End	*	*	*	*	*	*	*	*
Pointers in System Buffers :								
LEX BUFFER Pointers	B	BZ	B	B	B	BZ	B	B
FIB: File Begin Field	B	BZ	B	B	B	BZ	B	B
FIB: Data Start Field	A	A	A	A	B	AZ	A	A
Pointers Within Environments:								
FOR/NEXT Stack Addresses	B	B	B	B	B	B	B	B
GOSUB/RETURN Addresses	B	BZ	B	B	B	B	B	B
Miscellaneous Pointers:								
PCADDR - Program Ctr at Stmt len	B	BZ	B	B	B	B	B	B
CNTADR - Continue Address	B	BZ	B	B	B	B	B	B
ERRSUB - ON ERROR-GOSUB Rtn Addr	B	BZ	B	B	B	B	B	B
ERRADR - ON ERROR Stmnt Addr	B	BZ	B	B	B	B	B	B
ONINTR - ON INTRPT Stmnt Addr	B	BZ	B	B	B	B	B	B
DATPTR - DATA Statement Pointer	B	BZ	B	B	B	B	B	B
TMRAD1 - ON TIMER#1 Stmnt Addr	B	BZ	B	B	B	B	B	B
TMRAD2 - ON TIMER#2 Stmnt Addr	B	BZ	B	B	B	B	B	B
TMRAD3 - ON TIMER#3 Stmnt Addr	B	BZ	B	B	B	B	B	B
NOTE -- THESE ARE NEVER UPDATED:								
INBS - Input buffer start	*	*	*	*	*	*	*	*
SNAPBF - Snapshot Buffer Address	*	*	*	*	*	*	*	*
RSTKBF - Rtn Stack Save Buf Addr	*	*	*	*	*	*	*	*

3.8.3 In System Buffer Area

When an buffer is created or deallocated, or when an existing buffer is expanded or contracted, pointers are updated to reflect this. All pointers in the RAM map between IOBFEN and AVMEMS, inclusive, are updated by a call to PTRAD2 from within the System buffer code.

HP-71 REFERENCE ADJUSTMENTS -- BUFFERS

B ::= Updated only if Begin Source <= address < Begin Dest

A ::= Updated only if Begin Source < address < Begin Dest

U ::= Unconditionally updated (offset always added to pointer)

Z ::= Address set to 0 if Begin Dest <= address < Begin Source

* ::= Not updated

Actions:

- Create ::= Item created
- Purge ::= Item purged
- At end ::= Movement at end
- Within ::= Item grows/shrinks in the middle

System Pointers:	ACTION ON I/O BUFFERS			
	Create	Purge	At end	Within
MAINST - MAIN File Chain Start	*	*	*	*
CURRST - Current File Start	*	*	*	*
PRGMST - Current Program Start	*	*	*	*
PRGMEN - Current Program End	*	*	*	*
CURREN - Current File End	*	*	*	*
MAINEN - MAIN File Chain End	*	*	*	*
CLCBFR - CALC Mode Buffer Start	U	U	U	U
RFNBFR - CALC Mode Refined Buffer	U	U	U	U
RAWBFR - CALC Mode Raw Buffer	U	U	U	U
CLCSTK - CALC Mode Token Stack	U	U	U	U
SYSEN - System RAM End	U	U	U	U
OUTBS - Output Buffer Start	U	U	U	U

AVMEMS - Available Memory Start	U	U	U	U
AVMEME - Top Math Stack	*	*	*	*
FORSTK - Top FOR/NEXT Stack	*	*	*	*
GSBSTK - Top GOSUB Stack	*	*	*	*
ACTIVE - Active Variable Pointer	*	*	*	*
CALSTK - CALL Stack	*	*	*	*
RAMEND - User RAM End	*	*	*	*
Pointers in System Buffers :				
LEX BUFFER Pointers	*	*	*	*
FIB: File Begin Field	*	*	*	*
FIB: Data Start Field	*	*	*	*
Pointers Within Environments:				
FOR/NEXT Stack Addresses	*	*	*	*
GOSUB/RETURN Addresses	*	*	*	*
Miscellaneous Pointers:				
PCADDR - Program Ctr at Stmt len	*	*	*	*
CNTADR - Continue Address	*	*	*	*
ERRSUB - ON ERROR-GOSUB Rtn Addr	*	*	*	*
ERRADR - ON ERROR Statement Addr	*	*	*	*
ONINTR - ON INTRPT Statement Addr	*	*	*	*
DATPTR - DATA Statement Pointer	*	*	*	*
TMRAD1 - ON TIMER#1 Statement Addr	*	*	*	*
TMRAD2 - ON TIMER#2 Statement Addr	*	*	*	*
TMRAD3 - ON TIMER#3 Statement Addr	*	*	*	*
Note that these are NEVER UPDATED:				
INBS - Input buffer start	*	*	*	*
SNAPBF - Snapshot Buffer Addresses	*	*	*	*
RSTKBF - Rtn Stack Save Buf Addrs	*	*	*	*

SYSTEM CONTROL	CHAPTER 4

This chapter describes the fundamental algorithms which control the behavior of the operating system. The over-all process by which the system repeatedly waits for and then processes the next command, is generally referred to as the "main loop."

The following diagrams and detailed algorithms describe the main loop and its related processes.

4.1 Main Loop Flow Diagram

```
                           +-------------+
                           |Cold Start   |
                           |Initialization|
                           +------+------+
                                  |
      +---------------------->|<----------------------+
      |                       v                       |
      |                    +-------------+            |
      |                    | Collapse    |            |
      |                    | Statement   |            |
      |                    | Buffer      |            |
      |                    +------+------+            |
      |                           |                   |
      |                           |                   |
      |                           v                   |
      |                    +-------------+            |
      |                    | Character   |            |
      |                    | Editor      |            |
      |                    +------+------+            |
      |                           |                   |
      |                           |                   |
      |                           v                   |
      |                    +-------------+            |
      |                    | Edit Line   |            |
      |                    | Into Command|            |
      |                    | Stack       |            |
      |                    +------+------+            |
      |                           |                   |
      |                           |                   |
      |                           v                   |
      |                    +-------------+            |
      |                    |             |            |
+-----+------+             | Parse Line  |            |
| Execute    |             |             |            |
| Statement  |             +------+------+            |
| Buffer     |                    |                   |
+------------+                    |                   |
      ^                           v                   |
      |                           ^                   |
      |                          / \                  |
+-----+------+                  /   \            +----+--------+
| Expand     |     no          /     \     yes   | Edit into   |
| Statement  |<---------------/ Program\-------->| Current     |
| Buffer     |                \ Line?  /         | Program     |
+------------+                 \     /           +-------------+
                                \   /
                                 \ /
                                  v
```

4.2 Algorithm

4.2.1 Cold Start

```
Enables interrupt system
Initialize CMOS test word
Initialize system RAM to zeroes
Reset display
Turn display on
Set display row drivers
Set display contrast nibble
Initialize DELAY parameters
Perform ColdStart configure
Create Statement Buffer
Initialize clock system
Check for low battery
Initialize flags and traps
Zero RAM between AVMEMS and RAMEND
Clear AUTO mode
Clear program running flag
Clear don't continue flag
Initialize IS-TBL table
Initialize PRINT and DISP position and width
Initialize ENDLINE string
Put Coldstart message in display
Create Workfile
Create file information buffer
Initialize random number seed
Perform coldstart fast poll
```

4.2.2 Main Loop, Wakeup, Power Off, Deep Sleep

```
MAINLP:  If MakeOff (flMKOF) is set then
            Set TurnOff (flTNOF)
            Clear MakeOff (flMKOF)
            Go to PWROFF
         If TurnOff (flTNOF) is set then
            Go to PWROFF
         If CALC mode (flCALC) is set then
            Give control back to CALC mode w/error
         Fast Poll (pMNLP)
         If in AUTO mode then
            Display Line; goto Wakeup
MAIN05:  If CALC mode (flCALC) is set then
```

```
            Give control back to CALC mode w/error
         Clear program annunciator & status bit
         Set Dormant flag (flDORM)
         If ATTN key has been pressed then
            Go to ATTNTN
         If Don't Prompt flag (flNOPR) is set then
            Go to WAKEUP
         If scrolling needed (NEEDSC) then
            Allow user to scroll
         If ATTN key has been pressed then
            Go to ATTNTN
         Send prompt string consisting of
            Cursor off, prompt character(">"),
            Cursor on
WAKEUP:  If ATTN key has been pressed then
            Go to ATTNTN
         Clear Don't Continue flag (NoCont)
         Collapse math stack
         Collapse AVMEMS,OUTBS,SYSEN to CLCSTK
         Clear Don't Prompt flag (flNOPR)
         Collapse statement buffer (bSTMT)
         Delete Immediate Execute Key buffer (bIEXKY)
         Set "Dormant" flag (flDORM)
         Call Character Editor
         If Immediate Execute Key then
            Go to IEXKEY
         If its not a cursor up or down key then
            Turn off command stack mode (flCMDS)
         Clear "Dormant" flag (flDORM)
         Clear Attention Flag so HPIL won't abort
         Move cursor to far right of display
         Go to appropriate place to process key
            Endline          (LINEP)
            Attention        (ATTNTN)
            RUN key          (RUNK)
            CONT key         (CONTK)
            SST key          (SST)
            Cursor Up        (CURSUj)
            Cursor Down      (CURSDj)
            Cursor Top       (CURSTj)
            Cursor Bottom    (CURSBj)
            G-Attention      (ATTNTN)
            CALC Mode key    (CALC)
            Off key          (PWROFF)
            Command Stack    (CMDSTK)


ATTNTN:  Flush key buffer
         If line feed (LF) wasn't last character sent to display
            then Call FINLIN to terminate previous display line
         Clear "need to scroll" flag (NOSCRL)
```

```
         Clear AUTO mode
         Go to  MAINLP


PWROFF:  Set flPWDN
         Call DPS010 to go to DSLEEP
         If there is an external command buffer
            Go to LINEP+ to process it
         If there is an STARTUP buffer
            Go to LINEP+ to process it
         Go to MAINLP


DSLEEP: Clear =flPWDN flag (indicate that we were not
          called from PWROFF).
DPS010: (Entry point for PWROFF).
        If ON key down
           Set ATTN flag and goto DSP040
        If display-clear flag clear then goto DPS030
           Send <cursor on>/CR/LF.
DPS030  Send <cursor off>
DPS035: Perform power-down poll.
        Set TURNOFF (flTNOF) flag.
        Clear MAKEOFF (flMKOF) flag.
        Turn off display.
        Clear f-g shift status bits.
        Clear ATNFLG and ATNDIS.
        Turn off timer #3 (Low battery check).
        Activate KB row with ATTN key.
        SHUTDN.

DPS040: Configure.
        Deallocate external command buffer (to give poll
          handlers a chance to create one if we were
          called by PWROFF).
        Check clock system
        If ATTN key woke us up, goto DPS200.
        If program running and ON TIMER pending
           Clear =flTNOF; goto DPS200.
        Perform pDSWNK poll (who woke us up?!?).
        If turnoff flag set and ATNFLG clear then
           goto DSP035
DPS200: Flush key buffer.
        Clear flALRM flag.
        =pDSWKY poll
        Password processing (does not require password if
          password=null or =flTNOF is clear).
        If failed to unlock machine (password required but
          not correctly given), goto DPS035.
        AC/BAT check
        RETURN
```

4.3 Interrupt Handling

The HP-71 CPU has a limited interrupt structure.

4.3.1 Causes of Interrupts

4.3.1.1 Keyboard Interrupts

An interrupt occurs whenever there have been no keys down and a key goes down. If there is already a key down then another key going down will not cause another interrupt. This type of interrupt is maskable. Only key rows activated by the lower 4 bits of the output register cause this type of interrupt. The ON-key does not cause this type of interrupt.

4.3.1.2 ON-Key Interrupt

This type of interrupt occurs when the ON (Attention) key is pressed. This interrupt is non-maskable. The ON-key receives special treatement by the hardware and is scanned during each instruction to check whether this key is down. The content of the output register is unimportant.

4.3.1.3 Module Pulled Interrupts

As a module is being plugged in or pulled out it will briefly complete a connection which signals the CPU that this is happening. The CPU latchs a status bit the indicates that a module has been pulled. This type of interrupt is non-maskable.

4.3.1.4 Other Interrupts

The CPU input register bit 14 is available to all ports. An interrupt occurs if some module pulls on this line. This type of interrupt is closely related to keyboard interrupts. The system interrupt routine has no provisions for processing this type of interrupt except to allow interrupts to be vectored to a specified address. This type of interrupt is maskable.

4.3.2 Interrupt Handling Algorithm

The system interrupt routine starts at address 0000F. The interrupt routine saves the A,B,C,D0,Carry,Hex/Dec Mode and P registers. It then checks for a module pulled interrupt. It then checks if the CMOS test word is intact and performs a COLDSTART if not. If the interrupt vector address is non-zero it jumps to it. Otherwise it waits approximately 16 milliseconds to debounce the keyboard and performs a keyscan. When the keyscan is completed, all the registers are restored and a return from interrupt is done.

```
        Save C(W) in R4
        Save R4(5-15) and D0 in INTR4
        Save A(W) in INTA
        Save B(W) in INTB
        Save 1 stack level, Pointer, Carry, and Mode in INTM
        If this is a module pulled interrupt
           goto MPI
        If Interrupt Ignore Flag is set
           Clear it and goto RESTORE
        If CMOS test word is invalid
           Perform Cold start
        If VECTOR is non-zero
           Jump to that address
        Wait 8/512ths second to debounce keyboard
        Call KEYSCN
    RESTORE:
      Restore Mode, Carry, Pointer and 1 Stack level
      Restore B(W)
      Restore A(W)
      Restore D0
      Restore C and R4
      Return from interrupt
```

4.4 Statement Parse

4.4.1 Initiation

Statement parse is initiated in one of four ways.

Statement parse usually begins when endline is entered from the keyboard. The display buffer moves to the command stack, which becomes the input buffer for parse (i.e., (INBS) is set to point to the entry in the command stack).

Statement parse also begins when the computer turns on and an external command buffer or a startup buffer exists; (INBS) is set to point into that buffer.

Statement parse is also initiated when a direct execute key is pressed; (INBS) is set to point at the key definition in the keys file.

TRANSFORM also initiates statement parse.

In all cases, the output buffer is the destination of the internal token stream as it is generated.

If the input line is a legal program line, the contents of the output buffer is edited into the current program. Memory associated with the output buffer is released.

If the input line is a Calculator BASIC statement (including implied DISP) and computer is not performing a TRANSFORM, the compiled line is moved into the statement buffer and executed. If the computer is performing a TRANSFORM, an input line without a line number will cause a transform failure.

4.4.1.1 External Invoking of Parse

The entry point, LNPEXT, allows parse to be called externally and have control returned to the caller. This entry point will set a flag, flRTN, to indicate external entry. Line parse will alter status bits S0 thru S11 and S13; these status bits should be saved by the caller if necessary. The pointer INBS should point at the start of the line to parse, and OUTBS should point to the start of the output line. The input line must be terminated by a CR (ASCII 13) and be preceded by a 3 nibble line length (similar to buffer format).

If the parser takes an abnormal exit, due to a parse error or insufficient memory, control returns to the caller, with the error in C(A) and the carry set. If the parse was successful, carry is clear.

On return, flRTN should be cleared by the caller. See the LINEP routine for further information.

4.4.2 Statement Parse Algorithm

Algorithm:

Entry point for externally invoking parse (LNPEXT) saves the caller's return stack level in S-R0-2 and sets the system flag flRTN. flRTN flags that all error exits (including MEMERR) will return to the caller with carry set and the error number in C(A). Goto A.

NOTE: Anyone using LNPEXT entry point MUST clear flRTN as soon as it returns to them!

```
   LINEP: (normal statement parse entry point)
      Copy Display Buffer to Command Stack (MAKEBF)
      Set INBS to start of input line in command stack
      Send Carriage Return & Line Feed  (CRLFOF)
       (so next character will clear display buffer)

  A:  Set OUTBS to AVMEMS (Collapses Output buffer)
      Point D1 to start of input line, using INBS
      Clear S0-S11, S13
      Set D(A) = End of Available Memory, using AVMEME
            D0 = OUTBS (Output buffer start)
      Call Block 1

      Retokenize lexeme
      If line#
        Set S5; Decrement D0 (delete statement
        length byte at buffer start); Output line#
        Call Block 2
        If tEOL
          If externally invoked (f1RTN set)
            THEN error
            ELSE clear AUTO flag; delete line
   B:   Decrement D0
        Call Block 1.
        Retokenize.

   C:   If Begin BASIC command (S3=1)
          THEN goto I.
          ELSE If System Command (S3=0,S0=1)
                 THEN error
   D:          If !
                 THEN parse remark; goto M
                 ELSE error.
       If externally invoked (f1RTN set)
         THEN error;
       Clear AUTO flag
       If tEOL (null line)
         THEN exit parse
         ELSE goto F.

      BLOCK 1:
      --------
        Save D0 (statement length byte) in INADDR;
        Increment D0; Clear RESTART flag (S-R1-3);
        Clear Err# (S-R1-0); Call NTOKEN;
        Set RESTART flag if XWORD or XFN &
        save RESTART address (S-R1-2).
        Save contents of LEXPTR (position of D1
        before NTOKEN call) in STMTD0 - will be
        needed to restore input pointer for RESTART.
        Clear Middle of IF flag (S9).
```

Entry point for variable or tFN after THEN/ELSE:

```
    E:  If variable or FN:
          set implied LET error flag.
          If no line# on line
            Clear AUTO flag
    F:    If implied LET errors (S10 set)
            Restore D1,D0 from R3; Clear S10
            If not in Middle of IF (S9=0=>try Implied DISP)
            THEN try implied DISP
            ELSE Decrement D0 4 nibbles (tEXTIF & stmt len byte);
                 Recover old INADDR from S-R0-0; Call GOSUBP;
          Goto K
        If looking at first lexeme on line
          If line# followed by !
            set S5; output line#; save D0 (location of
            statement length byte) in INADDR; increment
            D0; Parse remark; goto M
        If not a terminator (eg not tEOL,@,!,tELSE)
          If legal implied DISP statement followed by
          a terminator
            If no line number on line
                 Clear AUTO flag; goto K:
        Restore D1,D0; return
      END OF BLOCK 1

   ***Block 2 only returns if a label is not found***

      BLOCK 2:
      --------
        Save D0 (position of statement length byte) in
        INADDR; increment D0
        If quote
          Set appropriate flag(s);
          Step over it; Call FILEP+
          If legal
            THEN If matching closing quote
    G:             THEN if colon follows
                          THEN LEGAL LABEL;
                               Output tLBLST & label
                               If tEOL follows
                                 THEN goto N
                                 ELSE goto L  (parse as @)
                          ELSE RESPTR; Return
                   ELSE RESPTR; Return
            ELSE RESPTR; Return
        If 1st character is letter
          RESPTR; GNXTCR; FILEP1; Goto G
      END OF BLOCK 2
```

```
H:  If not Calculator BASIC (S0=0)
      THEN If begin BASIC (S3=1)
             THEN error
             ELSE goto D.
I:  If in IF statement (S-R0-3 nonzero)
J:    If not legal after THEN/ELSE (S2=0)
        THEN error
      If pending THEN (S6=1)
        If token is IF token
          THEN error

    If XWORD
      THEN Output 3-byte token
      ELSE Output 1-byte token
    Calculate Parse address
    Clear flags (S0,S8)
    Gosub to Parse routine   (CRGJMP)
    If Middle of IF return  (Carry Set)
      THEN Extended IF token already output;
           INADDR points to following byte;
           D0 is pointing past that byte
           S9 is set (middle of IF flag)
           S-R0-3 is nonzero (IF in progress)
           If S5=1
             THEN goto C
             ELSE goto H

 K: Normal stmt return (carry clr)
    Get Next Token
    If ELSE
      If no pending THEN (S6=0)
        THEN error
        ELSE Clear S6; Decr D0; Output t@;
             Call STMTLN, UPDIN+; Output tELSE
             Call ELSEP; goto K
    Check legal stmt terminators (@,!,EOL)
     Clear S7
     If @  (Multi-statement line)
 L:    THEN Set S7, Output t@
       ELSE If !   (Remark)
              THEN Output t!, Remark; goto M
              ELSE If EOL
 M:                  THEN Output tEOL
                     ELSE Error Exit --> Excessive Chars
 N:  Output terminator
     Clear S10 (Implied LET error flag)
     Calculate & write out statement length
     If multi-statement line
       If S5=1
         THEN Call Block 2; Goto B
         ELSE Call Block 1; Goto H
```

```
Set AVMEMS to D0
If line# found (S5=1)
  If externally invoked (f1RTN set)
   THEN exit with carry clear
   ELSE Edit line into program memory (PEDIT)
        Return to Main Loop
Calculate output buffer length, move to I/O buffer
area; call SYCOLL (Resets AVMEMS,OUTBS to SYSEN)
Execute calculator BASIC Stmt (RUNX+)
```

See the portion of the algorithm handled in IFP in JP&PR3

NOTES: Line parse only special checks for external invoking in 4 distinct places.
1) eol, 2) line# followed by eol, 3) parse error,
4) correctly parsed line about to be edited into program memory.

Implied DISP is not legal immediately after THEN/ELSE.
Implied DISP is not legal during TRANSFORM.

4.4.3 Errors and Restart

Often when a keyword parse fails, it is because the keyword was not initially recognized. For example: Assume there is a FORM keyword on a plug-in LEX file; FORM takes a single string expression as a mandatory parameter. Further assume the user types in: >10 FORM=1 TO 5

FORM parse fails; a mechanism exists wherein the lexical analyzer is restarted to find FOR parse. This capability is set up in the main parse driver, and implemented in the parse error handler.

4.4.4 Restart Algorithm

Algorithm:
```
If S4=0
  THEN RESPTR
If RESTART flag (S-R1-3) set
  THEN goto RESTAR;
  ELSE If previously restarted (S-R1-0 [err#] #0)
         THEN Restore D1 to original error position
              using S-R1-1; Set D0 from S-R1-0;
       If Implied LET error (S10=1)
         Restore D1,D0 from R3; Clear S10;
         If not in middle of IF (S9=0)
```

```
                THEN try implied DISP
                ELSE Decrement D0 4 nibbles
                     (over tEXTIF & stmt length byte);
                     Recover old INADDR from S-R0-0;
                     Call GOSUBP;
            Handle as error.
```

4.4.5 Parse Routines

For further details on parse routines and writing parse routines see the

4.5 Statement Decompile

4.5.1 Initiation

Statement decompile is called as a subroutine by DCPLIN whenever a BASIC program line is to be displayed for editing. DCPLIN is called by AUTO, FETCH, cursor up, cursor down, cursor top and cursor bottom. LIST and single step (SST) invoke statement decompile directly. The two "standard" entry points are: 1) LDCOMP, which updates CURRL (Current Line) and decompiles the entire line, and 2) LDCM10 (used by LIST), which decompiles the entire line without updating CURRL. The "single step" entry (LDSST1/LDSST2) decompiles only one statement.

4.5.1.1 External Invoking of Decompile

Decompile can be externally invoked, using the LDCEXT entry. This entry sets the flRTN flag, so control returns to the caller in all cases, even if an error occurs. if this error occured. The flag, flRTN, MUST be cleared by the caller on return.

TRANSFORM utilizes this entry point.

4.5.2 Algorithm

```
LDCEXT entry: (external invoking of decompile - used by TRANSFORM)
     Saves caller's return address in S-R0-2; Sets flRTN so in case
     of MEMERR will still return. Goto LDCM10.

LDCOMP entry: (cursor up/cursor down)
     Update Current Line;
LDCM10 entry: (LIST)
```

```
     Clear SST (S1) flag;
LIST/SST entry:
     D(A)<--AVMEME; D0<--OUTBS; Decompile Line#;
     Save desired cursor position in LDCSPC (pointed to
     by D0);
A:   Save address of line length byte (pointed to by D1)
     in INADDR;

SST entry for multi-statement line:
     Step D1 over statement length byte; Clear S8, S9;
     If label declaration (tLBLST)
       Step D1 over tLBLST and 5 nibble chain length;
       Output quote; Call ASCICK; Output quote & colon;
       If at tEOL
         THEN goto OUTEOL;
         ELSE goto A.
     If variable (<6A)
       THEN goto LETDC.
     If user defined function (tFN)
       THEN goto FNDC.
     If remark (t!)
       THEN goto !DC.
     Call GTEXTI;
     If text not found
       THEN output 'XWORD', followed by ID#;
            Use INADDR to get to end of statement;
            Goto OUTELA;
     Output text; Read in 1st 6 nibbles of tokenized
     line into A; Copy A into C; Jump to decompile address.
```

4.5.3 Decompile Routines

For further details on decompile routines and writing decompile routines, see the "Statement Parse, Decompile, and Execute" chapter.

4.6 Program Edit

At edit time, all program execution stacks are collapsed. The FOR/NEXT and GOSUB/RETURN stacks are collapsed. The CALL stack is also collapsed. Only one set of variables exists.

```
     LOW    +---------------------+
            |       System        |
            |        RAM          |
            +---------------------+
            |  Variable Pointers  |
            +---------------------+
            |   Display Buffer    |
            +---------------------+
            |   Configuration     |
  MAINST--> +---------------------+
            |       Files         |
  CURRST--> +---------------------+
            |                     |
  PRGMST--> |                     |
            |                     |
  PRGMEN--> |                     |
  CURREN--> +---------------------+
            |         |           |
  (IOBFST)  |         v           |
  MAINEN--> +---------------------+
            |    Buffer List      |
  INBS ---> +---------------------+
            |    Input Buffer     |
  OUTBS --> +---------------------+
            |   Output Buffer     |
  CLCBFR--> +---------------------+
            |   Command Stack     |
  CLCSTK--> |                     |
  AVMEMS--> +---------------------+
            |  Available Memory | |
            |                  v  |
            |                     |
  AVMEME    |                     |
  ACTIVE--> +---------------------+
            |     Variables       |
  RAMEND--> +---------------------+
     HIGH
```

4.6.1 Global Assumptions

If PEDITD entry, S8 set indicates the line to PEDIT is null, i.e., the line number followed by EOL.

4.6.2 Program Edit Algorithm

```
 PEDIT:   Clear null line flag (S8);
 PEDITD:  If current file not BASIC or if protected
            THEN error;
```

```
PEDITM:  Zero out all GOTO/GOSUB links;
         Update current line;
         Collapse stacks;
         If null line
           THEN collapse output buffer;
         If line exists
           THEN set R3 to line length
           ELSE set R3 to 0;
         Call RPLLIN
```

```
+--------------------------------------+----------------+
|                                      |                |
|  THE BASIC INTERPRETER               |  CHAPTER  5    |
|                                      |                |
+--------------------------------------+----------------+
```

5.1 BASIC Interpreter

5.2 Entering the BASIC Interpreter

The BASIC interpreter is entered through two entry points: BSCEXC and BSCEX2. The first entry point is used when executing from the Keyboard. The second entry point allows the "Don't Continue" (NoCont) flag to be set, indicating that execution will halt after the next statement is executed. This entry point is used for Single Step execution, RUN, CONT and CHAIN.

The global flag, PgmRun (S13), is set before entry if a program is executing.

A Fast Poll (pBSCEN) is sent out when entering the BASIC interpreter.

The BASIC interpreter executes a statement at a time, not an entire line. The current BASIC program counter (PCADDR) is updated to the statement length byte of the statement to be executed. Status (S0-11) are cleared. If the begin token of the statement is a BASIC statement token, the execution address is computed and jumped to. Otherwise, the statement is assumed to be an Implied LET statement and Assignment Execute is called as a subroutine.

5.3 Reentering the BASIC Interpreter

Most statements return to the BASIC Interpreter through a direct jump to NXTSTM. This routine computes the address of the next statement, using the current program counter address (PCADDR) and the corresponding statement length. NXTSTM jumps directly to back into the BASIC Loop (at RUNRTN), with the data pointer (D0) positioned at the next statement to execute. This mechanism was developed to allow execution routines an additional subroutine level, rather than using a hardware return stack level to jump to each routine and having them do a machine code 'RTN'.

Statements that change program flow, such as GOTO, GOSUB, CALL, END SUB, and FN, jump directly back to the BASIC Loop with the data pointer (D0) set at the appropriate "next statement" address.

Error Exits from BASIC (through MFERR or BSERR), return to RUNRT1 with the data pointer (D0) at the statement in error.

RUNRT1 explicitly clears sENDx, a status to indicate an END statement execute, allowing execution routines to use this status internally. NXTSTM explicitly clears this flag.

The Math Stack is collapsed at the end of every statement execute. Since Expression Execute (EXPEXC) does not collapse the Math Stack, this clean up is necessary between statements and eliminates the need for individual execution routines to do it.

Exceptions are checked at the end of every statement. See the section below on Exception Handling.

5.4 Exiting the BASIC Interpreter

A global flag, NoCont (S14), indicates if program or statement execution is Not to Continue. This flag is set several ways: Single Step sets NoCont before the "continue" statement is executed; PAUSE, Ending or Stopping a Program, Error Exit, hitting the ATTN Key, GOTO from the keyboard, also set NoCont. RETURN,END,ENDSUB,ENDDEF executed from the keyboard and returning to program execution set NoCont.

The ERROR exit flag (sERROR) is set when the error message handler jumps to ERRRTN. In all other returns, this flag is cleared.

If execution is to continue, the BASIC Interpreter continues by executing the next statement. If execution is to stop several things are done. The program annunciator is cleared. The filetype of the current file is checked. If the file is non BASIC or a program is not running, all open file buffers are flushed, unless an error ocurred (sERROR). The Fast Poll: pBSCex is issued.

Non BASIC file execution that is interrupted due to an error exit are not "SUSPended" like a BASIC program. Responding to the pBSCex poll can change this.

If the current program is BASIC and the current statement is not an END or STOP statement (sENDx=0), the continue address (CNTADR) is set at the current D0 and the SUSP annunciator is lit.

The current D0 is the "next statement" to execute if execution is continued. In the case of Errors, the "next statement" is the statement generating the error. IF/THEN execution could pause with the "next statement" at the ELSE clause. If the next statement execution token is "ELSE", a statement skip is done to position the next statement execution past the ELSE clause. For END statement execution, there is no next statement to execute. The continue address has been zeroed and must not be updated.

The current line is computed and updated, to reflect where the program halted.

Statement execution (from the keyboard/statement buffer) halts when End of Line is reached. When beginning to execute the "next statement" of a program, if the next statement address is past the current program end, an END statement is executed.

Except for errors, all exits from the BASIC Interpreter flush open file buffers. This can not be done for an error because an error generated from attempting to flush file buffers would cause an infinite loop. All exits from the BASIC Interpreter issue a Fast Poll (pBSCex) when exiting the BASIC Interpreter and clear the NoCont flag. Control jumps to the Main Loop.

5.5 Exception Handling

Except in the case of an error, execution exceptions are checked at the end of every statement. Exception checking is skipped for errors so timer expiration execution will not continue after an error message is generated.

A global status flag, Except (S12), indicates an exception has occurred. This flag can be set at various times during statement execution, to indicate an exception has occured and service may be required at the end of statement execution.

An exception is a software interrupt--a condition which will be serviced after execution of the current statement. An exception is ALWAYS set by software, although the software may be setting it because of a hardware condition. The computer's procedure for checking exceptions is as follows:

> If no exceptions have occured (Except is clear), a hardware service request is issued (SREQ?). If no hardware service request results, timers are checked for expiration. If no timers expired, there are no exceptions to service.

If an exception (Except is set) or hardware service request occurs, CKSREQ is called. This routine, explained elsewhere, checks for hardware service requests which can be handled by the mainframe: expiration of any of the three countdown timers. Then, if a hardware service request is still pending OR the software exception flag (Except--S12) is set, a pSREQ poll is issued. This is the opportunity for other device-handling software (HPIL, for example) to do whatever it needs to do. This is also an important spot for any external clock system (pocket secretary, instrument controller, etc.) to schedule alarms.

After CKSREQ, if the exception flag is set, it is cleared and a pEXCPT poll is issued. Unlike pSREQ, which may occur between or during statements, pEXCPT occurs at a well-defined spot, and therefore allows more latitude in what can be done during poll handling. See the poll documentation header for more information. The ATTN/ON Key is checked after this poll.
If a program is running when exceptions are checked, Pending Alarm RAM is checked by calling ALMSRV to see if one of the three BASIC timers has expired. If a timer has expired and the associated ON TIMER address is within the current program scope, the ON TIMER code is executed. Control returns to the BASIC Loop through normal statement execution return at RUNRTN/RUNRT1.

5.5.1 Servicing Clock System Exceptions

Exception handling is one of the prime times to service the clock system. The system provides an external alarm "slot" for use by all applications which need to schedule alarms. The pSREQ and pEXCPT polls provide an opportunity to schedule alarms and to SET UP to process alarms. Although alarms cannot actually be processed during these polls (except for non-disruptive events, like beeping), it is possible to set up a command buffer or some such mechanism for later processing.

See the "Clock System" chapter for details about the clock system.

5.5.2 Algorithm

```
BSCEXC:  Clear No Continue of Program flag       (NoCont)
BSCEX2:  Place current D0 into R0
         Fast poll on entering BASIC interpreter(pBSCen)
         If not running                          (not PgmRun)
            goto BSCX+
BSCXLP:  Read & Move past EOL |
         If EOL and not running
            go exit BASIC                        (goto BSCEXT)
```

```
         If   (multi-statement line)
            go Update PC address               (goto BSCX+)
         If End of current program
            go execute END statement
         Skip line
  BSCX+: Save addr  statement length byte        (PCADDR)
         Skip statement length byte
         Clear status  (S0-S11)
         Read Begin BASIC token
         If not Begin BASIC token range          (BASICs)
            Call Assignment Execute
            Skip to next statement               (NXTSTM)
         else
            Move past BASIC token
            Calculate Execution addr             (EXCADR)
            Jump to Execution routine
```

Statement Execute Return: (from NXTSTM or directly)

```
 RUNRT1: Clear END execute flag                   (sENDx)
 RUNRTN: Clear Error flag                         (sERROR)
 ERRRTN: Collaspe Math Stack
         If ERROR
            Skip exception checking              (goto 6)
         If no exceptions                      (Except=0)
            If no hardware service request
               If any pending alarm set          (PNDALM)
                  Save D0 on stack
                  go Process timers              (goto 3)
               go continue                       (goto 6)
         Save D0 on stack
         Check Service requests                  (CKSREQ)
         If no exceptions                      (Except=0)
            go Restore D0 and continue           (goto 5)
         Clear Exception Flag                    (Except)
         Fast Poll on Exception                  (pExcpt)
         Restore low status from DSPSTA          (USRSTA)
     3:  If ATTN Key hit                           (CKON)
            Set NoCont flag                         (S14)
         If Program running
            Load mask to check Timer bits
            Read Pending Alarm field
     4:     If Timer expired         (Bit 0|1|2 of PNDALM)
               Get Timer Address
               If non-zero Timer address
                  Verify address in prgm scope   (SCOPCK)
                  If within scope
                     Clear timer bit in PNDALM
                     Enable another Timer to be serviced
                     C <-- ON TIMER address
                     Set ONTIMER statement flag
```

```
                      go process ON TIMER statement
                go Check if any other Timers off     (goto 4)
        5: Restore D0 from R0
           Clear Error occured flag                  (sERROR)
        6: If Continue
              go process next of statement           (BSCXLP)
           else
  BSCEXT:     Clear PRGM annunciator                 (SflgCp)
              Read filetype                          (RDCHD+)
              If non-BASIC file                      (BASCHK)
                 go exit BASIC                  (goto BSCEX+)
              If not running
                 go exit BASIC                  (goto BSCEX+)
              else
                 If not END/STOP execute             (sENDx)
                    If ELSE
                       Skip to End of Line
                    Update Continue Address
                    Set SUSP Annunc/Flag
                 Compute & update current line
 BSCEX+:
           If not an error
              Flush all open files
           Fast Poll on Exiting BASIC interp         (pBSCex)
           Clear Don't Continue flag
           golong MAIN Loop                          (MAINLP)
```

5.6 Immediate Mode

Whenever a line without a line number preceding it is legally parsed, that line is executed immediately.

The BASIC Interpreter is entered at BSCEXC. The program running flag (PgmRun) is clear.

5.6.1 Statement Buffer

An immediate execute line is moved from the output buffer into the statement buffer before being executed. The statement buffer is always the first buffer in the Buffer chain, ensuing that only movement of mainframe files affects the value of the BASIC program counter.

5.7 Program Execution

Program Execution begins through the RUN Key, RUN statement, CHAIN statement, CONT Key, CONT statement and the SST Key.

Before running a program, several things are done. If a filename is specified in the RUN statement, the Current File pointers are changed to point to the file. In the case of CHAIN, the current file is purged.

If the filetype is neither BASIC nor binary, a poll is issued (pRUNft) allowing a Lex File to take over the RUN/CONT/CHAIN statement.

Except for continuing or single stepping at a valid continue address, program scope is recomputed and reset. All labels and user defined functions are chained. In case any of the direct execute keys (RUN, CONT, SST) were hit within Auto Mode, AUTO Mode is cleared.

If the program file is empty, control returns to the Main Loop.

In the case of RUN or CHAIN, all BASIC stacks are collapsed. For RUN, the Assign Table and all FIB entries are deleted.

For CONT and SST, if the continue address (CNTADR) is non-zero, execution is continued at this address. Otherwise, CONT and SST begin execution at the first statement of the program, after collapsing stacks, deleting the Assign Table, and deleting all FIB entries (acts as a RUN). A CONT execution collapses the Statement Buffer to prevent a subsequent "Return to Keyboard" in a paused program from returning incorrectly to the Statement Buffer containing "CONT".

The suspend annunciator is cleared, the program running flag is set, along with the PRGM annunciator.

If a binary program is to be run, a poll is issued (pRUNnB), indicating beginning execution of a non-BASIC file. The binary file type is passed. On return from the poll, the binary code is branched to by pushing its address on the hardware return stack and doing a machine code 'RTN'. The binary program exits by branching to the EXITRN entry point in the RUN statement code; this clears flags and exits through BASIC.

If a BASIC program is to be run, the BASIC interpreter is entered at BSCEX2.

```
NIBHEX 0
CON(3) <aaa>
CON(3) <bbb>
CON(3) <ccc>
CON(3) <ddd>
CON(3) <eee>
CON(3) <fff>
CON(3) <ggg>
CON(3) <hhh>
CON(3) <iii>
CON(3) <jjj>
CON(3) <kkk>
CON(3) <lll>
CON(3) <mmm>
CON(3) <nnn>
CON(3) <ooo>
CON(3) <ppp>
CON(3) <qqq>
CON(3) <rrr>
CON(3) <sss>
CON(3) <ttt>
CON(3) <uuu>
CON(3) <vvv>
CON(3) <www>
CON(3) <xxx>
CON(3) <yyy>
CON(3) <zzz>
NIBHEX 0
```

The 0-nibble at either end serves to identify the presence of the speed table whether the code is looking for it from above or below. (Similarly, the single F-nibble identifies the absence of the speed table whether the code is looking for it from above or below.)

The quantities <aaa>, <bbb>, <ccc>, et cetera are offsets into the text table. The text table is maintained in approximately alphabetized form (see TEXT TABLE below for more detail), and the 3-nibble quantities in the speed table identify the position of each alphabetic-character's first entry RELATIVE to the start of the text table.

EXAMPLE:
If the first entry starting with the letter "P" is at address 126 (decimal) relative to the start of the text table, the line appearing as "CON(3) <ppp>" above would actually be "CON(3) 126".

If there are no keywords beginning with a particular letter, the 3-nibble offset for that letter should be the size of the entire text table.

EXAMPLE: If the text table is 459 (decimal) nibbles long and there are no keywords beginning with Q, the line appearing as "CON(3) <qqq>" above would actually be "CON(3) 459".

TEXT TABLE OFFSET: 4 nibbles Offset from current location to the second nibble of the text table (start of first text string). If the beginning of the text table is labeled "TxTbSt", an assembly-language pseudo-op to properly fill this location would be:

```
CON(4) (TxTbSt)+1-(*)
```

MESSAGE TABLE OFFSET: 4 nibbles Offset from current location to the beginning of the message tables. The message table must be structured to work with the message-handling system described in the "Message Handling" chapter. If there is no message table, the value should be zero.

POLL HANDLER OFFSET: 5 nibbles Offset from current location to the poll handler for this LEX file. If there is no poll handler, this should point to an RTNSXM instruction. Since the RTNSXM instruction is a "00", setting this field to "00000" will point it at itself, which will conveniently turn out to be an RTNSXM instruction.

MAIN TABLE: 9 * (# of keywords) nibbles The Main Table contains information needed to run or to decompile every token in the LEX file. The entries are in token number order. The first table entry corresponds to the lowest token # in the LEX file, the second table entry corresponds to the next token #, et cetera.

Each main table entry takes 9 nibbles and is formatted as follows:

TEXT TABLE OFFSET: 3 nibbles This is the position of the corresponding text in the text table for this keyword, relative to the start of the text table. This points at the START of the text table entry--the nibble count, which is one nibble before the start of the actual text (see description of TEXT table below).

EXECUTION ADDRESS: 5 nibbles Offset relative to current location of start of execution code for this keyword. The corrsponding parse address for the token is 5 nibbles above the start of the execution code. The corresponding decompile address for the token is 10 nibbles above the start of the execution code.

CHARACTERIZATION NIBBLE: 1 nibble The characterization nibble categorizes a token during statement parse. If the keyword is a function (string or numeric), this nibble is always a hex "F". Otherwise, the four bits of this nibble mean the following:

bit 0: Calculator BASIC (Legal from the keyboard)
bit 1: 0 (unused)
bit 2: Legal after THEN/ELSE
bit 3: Begin BASIC (Programmable)

Some examples follow:

For keywords which are programmable, legal after THEN/ELSE, and legally executed from the keyboard, the characterization nibble is "D"; an example is the DISP keyword.

For keywords which are used strictly as intermediate keywords (such as PORT in the mainframe), the characterization nibble is "0".

Non-programmable commands (like FREE and EDIT) which are legal after THEN/ELSE should have a characterization nibble of "5"; note that a keyword which is Calculator BASIC, but not Begin BASIC, is interpreted as non-programmable.

On the other hand, a keyword which is Begin BASIC but not Calculator BASIC, is not executable from the keyboard, but only makes sense within the context of a program; the DATA keyword, which has a characterization of "8", is an example of such a keyword.

In all cases, bit 1 of the characterization is unused.

TEXT TABLE: 3*(# of keywords) +2 * (total # chars) + 3 nibbles
Strictly speaking, the text table does not have to reside immediately after the main table. It can reside anywhere since its address is specified in the header. The text table contains the text representation of all keywords in the LEX file, and is used by the parse and the decompile drivers.

Entries in the Text Table are in alphabetical order with one important difference: a shorter keyword which comprises the first part of a longer keyword, occurs AFTER the longer keyword. In other words, the keyword "ABC" must appear after the keyword "ABCD". If this is not done, the parse driver (which scans the text table linearly from beginning to end) will never find the keyword "ABCD" because it will match on the keyword "ABC" first. (Equivalently, for purposes of sorting the keyword list, the keywords can be considered to be padded with "FF"s out to eight bytes.)

The entry for each keyword in the text table has the following format:

(SIZE OF TEXT - 1) IN NIBS: 1 nibble If the text is 2 chars (4 nibs), this field = "3". If the text is 3 chars (6 nibs), this field = "5". And so on. Needless to say, the maximum value for this field is "F", implying that the maximum length of a keyword is 8 characters.

TEXT: 2-8 bytes (as specified above) Text of keyword in ASCII. Note that keywords must be at least two characters long, since one character keywords would conflict with variable names.

TOKEN #: 1 byte
Token # of this keyword.

The Text Table is terminated with the nibbles "1FF".

EVERYTHING ELSE: This ends the list of required components of a LEX file. All that is needed now is the following:

1) MESSAGE TABLE If there is a message table for this LEX file, its address is specified in the header. The message table must conform to the standard message table format; the first byte contains the lowest message#, and the second byte contains the highest message#.

When calling the mainframe message routines (BSERR and MFWRN), a message within this table is specified by the LEX ID# in C[3-2] and the message number in C[B].

See the "Message Handling" chapter for further details.

2) POLL HANDLER Offset to the poll handler's address is specified in the header. See the section on polling for further details.

3) STATEMENT/COMMAND/FUNCTION EXECUTION CODE The execution code of the statement, command, or function. Statement execution entry points are preceded by decompile and parse addresses; non-programmable statement execution entry points are preceded by a parse address only; function execution entry points are preceded by a parameter count and description.

6.1.1 How it All Works

The SPEED Table, MAIN table and TEXT table are the tools with which the BASIC language is extended. The mainframe keeps a directory of all the LEX files in the machine, and refers to this directory at parse, decompile, and execution time. See the LEX Entry Buffer section under the "Table Formats" chapter for details.

6.1.1.1 Parsing

When the lexical analyzer (NTOKEN) is trying to tokenize text, it searches the LEX file text tables for a matching string. If there are a lot of keywords in the LEX file, the presence of an optional speed table speeds this searching.

Once a matching string has been found, the lexical analyzer reads the token number associated with the keyword. This token number serves as an index into the main table. The main table provides the execution address.

For a statement, the code at the execution address is immediately preceded by a 5 nibble offset to the corresponding parse routine, so that the parse driver is able to find the parse routine for a particular statement.

For a function, the execution code is immediately preceded by the parameter count and parameter descriptors; these are used by the expression parser to parse the function.

6.1.1.2 Decompiling

When decompiling, the decompile driver has a token number and a LEX ID number. The LEX ID number and token number locate the proper LEX file; the relative token number serves as an index into the main table. From the main table the decompile driver fetches the following:

1) The location of the text table entry for the text of the keyword, and

2) The execution address. For a statement, ten nibbles prior to the execution address is the five nibble offset to the corresponding decompile routine; this is used by the decompile driver to invoke the decompile routine for a particular statement.

For a function, the expression decompiler uses the parameter count and parameter descriptors which immediately precede the execution address to decompile the function.

6.1.1.3 Execution

When executing an external statement or function, the LEX ID and token number are used to locate the proper LEX file. The relative token numbers serves as an index into the Main table. The execution address is calculated and jumped to, beginning execution of the keyword.

6.1.2 How to Create a LEX File

The HP-71 provides no mechanism to create a LEX file other than to copy it from an external device or to POKE it into a file chain. A number of tools have been used by the HP-71 software development team to assist in creating LEX files. They are described below.

6.1.2.1 HP-71 Assembler

An assembler is obviously the most important tool. The HP-71 assembler is available both in the HP-71 Assembler/FORTH ROM, as well as in a special set of programs which run on the HP200 series machines.

Note that assembly language examples given in this section are in the proper format for the assembler which was used by the HP-71 mainframe software development team.

6.1.3 Symbolic Referencing

Following are copies of the mainframe and built-in XWORD tables which comprise every keyword token in the mainframe; these files were used to generate all the necessary tables. Note that in the first table all the token names are given as starting with 't', indicating one-byte tokens. In the second table (as with all LEX files), all the token names begin with 'x', indicating these are not complete tokens, but only the first byte of a three-byte token. We discuss later how to build the symbolic for the complete three-byte token.

6.1.3.1 Mainframe Tokens

```
                      RTNSXM          MAINTS         00
******** ****** ****** ****** ****** ****** **
*    ^      ^      ^      ^      ^      ^     ^
*   File   Msg    Poll   EOF   TblNam TblLnk ROM#
*
*
** ******** **** ****** ****** ****************************
** ******** **** ****** ****** ****************************
*^      ^     ^^^^    ^      ^         ^
*T      T     BLSC    E      T         C
*o      e     eeya    x      o         o
*k      x     ggsl    e      k         m
*e      t     iatc    c      e         m
*n            nle     u      n         e
*              mB     t                n
*             BA A    i      n         t
*             AfCS    o      a
*             StmI    n      m
*             ledC           e
*             Cr      A
*                     d
*                     d
*                     r
*
*
00 FN             FN-GO         FN (lex only)
00 GO                           GO (lex only)
01                TRMNTR        Dummy Fill
02                BLDNUM tINT12 12-Digit Integer
03                BLDNUM tINT11 11-Digit Integer
04                BLDNUM tINT10 10-Digit Integer
05                BLDNUM tINT9  9-Digit Integer
06                BLDNUM tINT8  8-Digit Integer
07                BLDNUM tINT7  7-Digit Integer
08                BLDNUM tINT6  6-Digit Integer
09                BLDNUM tINT5  5-Digit Integer
0A                BLDNUM tINT4  4-Digit Integer
0B                BLDNUM tINT3  3-Digit Integer
0C                BLDNUM tINT2  2-Digit Integer
0D                TRMNTR        [Unused]
0E                TRMNTR tLBLRF Label Reference
0F                TRMNTR tLINE  Line Number
10                TRMNTR tBIG   Constant Too Big
11                TRMNTR tSMALL Constant Too Small
12                BLDNUM tFLT12 12-Digit Float
13                BLDNUM tFLT11 11-Digit Float
14                BLDNUM tFLT10 10-Digit Float
15                BLDNUM tFLT9  9-Digit Float
16                BLDNUM tFLT8  8-Digit Float
17                BLDNUM tFLT7  7-Digit Float
18                BLDNUM tFLT6  6-Digit Float
```

19	BLDNUM	tFLT5	5-Digit Float
1A	BLDNUM	tFLT4	4-Digit Float
1B	BLDNUM	tFLT3	3-Digit Float
1C	BLDNUM	tFLT2	2-Digit Float
1D	BLDNUM	tFLT1	1-Digit Float
1E	TRMNTR		[Unused]
1F	TRMNTR		[Unused]
20	TRMNTR		[Unused]
21	TRMNTR	a!	(!)
22	STRLIT	a"	(") (String Delimiter)
23	TRMNTR		(#)
24	TRMNTR	a$	($)
25	TRMNTR		(%)
26	TRMNTR		(&)
27	STRLIT	a'	(') (String Delimiter)
28	TRMNTR		(
29	TRMNTR		)
2A	TRMNTR		(*)
2B	TRMNTR		(+)
2C	TRMNTR		(,)
2D	STRING	tSVAR	String Variable (-)
2E	TRMNTR	a.	(.)
2F	TRMNTR		(/)
30	ONEDGT	a0	0 (Digit)
31	ONEDGT	a1	1 (Digit)
32	ONEDGT	a2	2 (Digit)
33	ONEDGT	a3	3 (Digit)
34	ONEDGT	a4	4 (Digit)
35	ONEDGT	a5	5 (Digit)
36	ONEDGT	a6	6 (Digit)
37	ONEDGT	a7	7 (Digit)
38	ONEDGT	a8	8 (Digit)
39	ONEDGT	a9	9 (Digit)
3A	TRMNTR		(:)
3B	TRMNTR		(;)
3C	TRMNTR		(<)
3D	TRMNTR		CALC MODE ASNMNT OPRTR (=)
3E	TRMNTR		(>)
3F	TRMNTR		(?)
40	TRMNTR		()
41	STATIC		A (Static Variable)
42	STATIC		B (Static Variable)
43	STATIC		C (Static Variable)
44	STATIC		D (Static Variable)
45	STATIC		E (Static Variable)
46	STATIC		F (Static Variable)
47	STATIC		G (Static Variable)
48	STATIC		H (Static Variable)
49	STATIC		I (Static Variable)
4A	STATIC		J (Static Variable)
4B	STATIC		K (Static Variable)

4C			STATIC		L (Static Variable)
4D			STATIC		M (Static Variable)
4E			STATIC		N (Static Variable)
4F			STATIC		O (Static Variable)
50			STATIC		P (Static Variable)
51			STATIC		Q (Static Variable)
52			STATIC		R (Static Variable)
53			STATIC		S (Static Variable)
54			STATIC		T (Static Variable)
55			STATIC		U (Static Variable)
56			STATIC		V (Static Variable)
57			STATIC		W (Static Variable)
58			STATIC		X (Static Variable)
59			STATIC		Y (Static Variable)
5A			STATIC	tZ	Z (Static Variable)
5B			TRMNTR		([)
5C			TRMNTR		(\)
5D			TRMNTR		(])
5E			TRMNTR		(^)
5F			TRMNTR		()
60			DYNAMC	tADIG0	Dynamic Variable 0
61			DYNAMC	tADIG1	Dynamic Variable 1
62			DYNAMC	tADIG2	Dynamic Variable 2
63			DYNAMC	tADIG3	Dynamic Variable 3
64			DYNAMC	tADIG4	Dynamic Variable 4
65			DYNAMC	tADIG5	Dynamic Variable 5
66			DYNAMC	tADIG6	Dynamic Variable 6
67			DYNAMC	tADIG7	Dynamic Variable 7
68			DYNAMC	tADIG8	Dynamic Variable 8
69			DYNAMC	tADIG9	Dynamic Variable 9
6A	IP	1111	IP	tIP	IP
6B	FP	1111	FP	tFP	FP
6C	MAXREAL	1111	MAXRL	tMAXRL	MAXREAL
6D	RMD	1111	RMD	tRMD	RMD
6E	RAD	1111	RAD	tRAD	RAD
6F	DEG	1111	DEG	tDEG	DEG
70	INF	1111	INF	tINF	INF
71	EPS	1111	EPS	tEPS	EPS
72	CEIL	1111	CEIL	tCEIL	CEIL
73	KEY$	1111	KEY$	tKEY$	KEY$
74	MOD	1111	MOD	tMOD	MOD
75	ERRL	1111	ERRL	tERRL	ERRL
76	ERRN	1111	ERRN	tERRN	ERRN
77	DATE	1111	DATE	tDATE	DATE
78	DATE$	1111	DATE$	tDATE$	DATE$
79	PI	1111	PI	tPI	PI
7A		1111	CMPLX	tCMPLX	CMPLX
7B	TIME	1111	TIME	tTIME	TIME
7C			FN	tFN	FN
7D			ARRAY	tARRAY	ARRAY
7E			DMARRY	tDMYAR	Dummy array

7F	RES	1111	RES	tRES	RES
80		0000	INVLUT	t^	^ (INVOLUTION)
81	NOT	0000	NOT	tNOT	NOT
82		0000	MINUS	t-	- (Unary)
83		0000	MULTPY	t*	*
84		0000	DIVIDE	t/	/
85		0000	PERCNT	t%	%
86	DIV	0000	DIV	tDIV	DIV
87		0000	PLUS	t+	+
88					[Unused]
89		0000	CONCAT	t&	& (CONCATENATE)
8A		0000	COMPAR	tRELOP	Relational operators
8B	AND	0000	AND	tAND	AND
8C	EXOR	0000	EXOR	tEXOR	EXOR
8D	OR	0000	OR	tOR	OR
8E			EXPR		[Unused]
8F			EXPR		[Unused]
90	LOG	1111	LOG	tLOG	LOG
91	LN	1111	LOG	tLN	LN
92	SQR	1111	SQR	tSQR	SQR
93	LOG10	1111	LOG10	tLOG10	LOG10
94	EXP	1111	EXP	tEXP	EXP
95	TIME$	1111	TIME$	tTIME$	TIME$
96	SIN	1111	SIN	tSIN	SIN
97	COS	1111	COS	tCOS	COS
98	TAN	1111	TAN	tTAN	TAN
99	ASIN	1111	ASIN	tASIN	ASIN
9A	ACOS	1111	ACOS	tACOS	ACOS
9B	ATAN	1111	ATAN	tATAN	ATAN
9C	INT	1111	INT	tINT	INT
9D	MEAN	1111	MEAN	tMEAN	MEAN
9E	SDEV	1111	SDEV	tSDEV	SDEV
9F	PREDV	1111	PREDV	tPREDV	PREDV
A0	RND	1111	RND	tRND	RND
A1	SGN	1111	SGN	tSGN	SGN
A2	ABS	1111	ABS	tABS	ABS
A3	NUM	1111	NUM	tNUM	NUM
A4	CHR$	1111	CHR$	tCHR$	CHR$
A5	VAL	1111	VAL	tVAL	VAL
A6	STR$	1111	STR$	tSTR$	STR$ (formerly VAL$)
A7		1111	SUB$	tISUB$	SUB$ (implied)
A8	FACT	1111	FACT	tFACT	FACT
A9	LEN	1111	LEN	tLEN	LEN
AA		1111	LPRP	tLPRP	LPRP ()
AB	UPRC$	1111	UPRC$	tUPRC$	UPRC$
AC	MIN	1111	MIN	tMIN	MIN
AD	MAX	1111	MAX	tMAX	MAX
AE	IVL	1111	IVL	tIVL	IVL
AF	OVF	1111	OVF	tOVF	OVF
B0	UNF	1111	UNF	tUNF	UNF
B1	DVZ	1111	DVZ	tDVZ	DVZ

B2 INX	1111 INX	tINX	INX
B3	1111 XFN	tXFN	XFN
B4	1111 XFN	tFFN	Funny Function
B4		LASTFN	Last Function
B5 COPY	1101 COPY	tCOPY	COPY
B6 LR	1101 LR	tLR	LR
B7 DELETE	0111 D'LTE	tDELET	DELETE
B8 EDIT	0111 EDIT	tEDIT	EDIT
B9 DEF	1101 DEF	tDEF	DEF
BA	0000 ENDDEF	tENDDF	END DEF (parsed by ENDP)
BB LIST	1101 LIST	tLIST	LIST
BC REAL	1101 REAL	tREAL	REAL
BD NAME	1101 NAME	tNAME	NAME
BE DESTROY	1101 DSTROY	tDSTRY	DESTROY
BF LINPUT	1101 LINPUT	tLINPT	LINPUT
C0 LET	1101 LET	tLET	LET
C1 SUB	1000 SUB	tSUB	SUB
C2	0000 ENDSUB	tENDSB	END SUB (parsed by ENDP)
C3 FOR	1001 FOR	tFOR	FOR
C4 NEXT	1001 NEXT	tNEXT	NEXT
C4		tLITRL	LITERAL (Literal label or file name)
C5 DISP	1101 DISP	tDISP	DISP
C6 DATA	1000 DATA	tDATA	DATA
C7 READ	1101 READ	tREAD	READ
C8 FETCH	0111 FETCH	tFETCH	FETCH
C9 INPUT	1101 INPUT	tINPUT	INPUT
CA INTEGER	1101 INTEGR	tINTEG	INTEGER
CB SHORT	1101 SHORT	tSHORT	SHORT
CC DIM	1101 DIM	tDIM	DIM
CD PRINT	1101 PRINT	tPRINT	PRINT
CE STAT	1101 STAT	tSTAT	STAT
CF KEYS	0000	tKEYS	KEYS
D0 CARD	0000	tCARD	CARD
D1 PORT	0000	tPORT	PORT
D2 MAIN	0000	tMAIN	MAIN
D3 DEGREES	1101 DEGREE	tDEGRE	DEGREES
D4 RADIANS	1101 RADIAN	tRDIAN	RADIANS
D5 ADD	1101 ADD	tADD	ADD
D6 DELAY	1101 DELAY	tDELAY	DELAY
D7 PAUSE	1100 PAUSE	tPAUSE	PAUSE
D8 WAIT	1101 WAIT	tWAIT	WAIT
D9 STOP	1101 STOP	tSTOP	STOP
DA END	1101 END	tEND	END
DB RETURN	1101 RETURN	tRETRN	RETURN
DC GOSUB	1101 GOSUB	tGOSUB	GOSUB
DD GOTO	1101 GOTO	tGOTO	GOTO
DE RESTORE	1101 RESTOR	tRESTR	RESTORE
DE		tRFILE	Run file specified in RUNP
DF IF	1101 IF	tIF	IF
E0 ON	1101 ON	tON	ON
E0		tCREF	Call by reference separator

```
E1 OFF      1101 OFF    tOFF    OFF
E1                      tCVAL   Call by value separator
E2 USER     1101 USER   tUSER   USER
E2                      tCOLON  HPIL colon token
E3 ERROR    0000 NXTSTM tERROR  ERROR
E4 TIMER    0000 NXTSTM tTIMER  TIMER
E5 KEY      1101 KEY    tKEY    KEY
E6 REM      1101 REM    tREM    REM
E7 IS       0000 NXTSTM tIS     IS
E8 BEEP     1101 BEEP   tBEEP   BEEP
E9 BASE          NXTSTM tBASE   BASE
EA TRACE    1101 TRACE  tTRACE  TRACE
EB PURGE    1101 PURGE  tPURGE  PURGE
EC CAT      1101 CAT    tCAT    CAT
ED OPTION   1101 OPTION tOPT'N  OPTION
EE AUTO     0111 AUTO   tAUTO   AUTO
EF          1101 XWORD  tXWORD  XWORD
F0          0000 TRMNTR tEOL    <eol>
F1          0000 TRMNTR tCOMMA  COMMA
F2          0000 TRMNTR tSEMIC  SEMICOLON
F2                      tIN     tIN      (for CALL)
F3 TO       0000        tTO     TO
F3                      tPRMST  PRMST    (Start of Parm list-SUB,CALL)
F4 THEN     0000 TRMNTR tTHEN   THEN
F4                      tEXTIF  Extended If
F4                      t       (Continuation)
F5 ELSE     0000 ELSE   tELSE   ELSE
F6 STEP     0000 LABEL  tSTEP   STEP
F6                      tLBLST  Label Statement
F7 TAB      0000 NXTSTM tTAB    TAB
F8 ALL      0000 NXTSTM tALL    ALL
F8                      tPRMEN  PRMEN    (End of Parm list-SUB)
F9 CALL     1101 CALL   tCALL   CALL
FA CFLAG    1101 CFLAG  tCFLAG  CFLAG
FB SFLAG    1101 SFLAG  tSFLAG  SFLAG
FC               BANG   t!      Comment
FD USING    0000 NXTSTM tUSING  USING
FE RUN      1101 RUN    tRUN    RUN
FF IMAGE    1000 IMAGE  tIMAGE  IMAGE
```

The following is the "built-in XWORD" table (LEX ID 01):

```
                 RTNSXM        xrm01s MAINTS 01
******** ****** ****** ****** ****** ****** **
*   ^     ^      ^      ^      ^      ^     ^
*  File  Msg    Poll   EOF    TblNam TblLnk ROM#
*
*
** ******** **** ****** ****** ****************************
** ******** **** ****** ****** ****************************
```

```
*^         ^     ^^^^     ^        ^           ^
*T         T     BLSC     E        T           C
*o         e     eeya     x        o           o
*k         x     ggsl     e        k           m
*e         t     iatc     c        e           m
*n               nle      u        n           e
*                  mB     t                    n
*                BA A     i        n           t
*                AfCS     o        a
*                StmI     n        m
*                IedC              e
*                Cr       A
*                         d
*                         d
*                         r
*
*
01 ACS           1111 ACOS             ACS
02 ADDR$         1111 ADDR$            ADDR$
03 ADJABS        1101 ADJAAA           ADJABS
04 ADJUST        1101 ADJNNN           ADJUST
05 AF            1111 AF               AF
06 ANGLE         1111 ANGLE   xANGLE   ANGLE  (function and middle word)
07 ASN           1111 ASIN             ASN
08 ASSIGN        1101 ASSIGN           ASSIGN
09 ATN           1111 ATAN             ATN
0A BYE           1101 BYE              BYE
0B CAT$          1111 CAT$             CAT$
0C STD           1101 STD              STD
0D FIX           1101 DSPF             FIX
0E SCI           1101 DSPF             SCI
0F ENG           1101 DSPF             ENG
10 CHARSET       1101 CHARST           CHARSET
11 CHAIN         1101 CHAIN            CHAIN
12 CHARSET$      1111 CHRST$           CHARSET$
13 CLAIM         0111 NASSAU           CLAIM (PORT)
14 CLASS         1111 CLASS            CLASS
15 CLOCK         0000         xCLOCK   (RESET) CLOCK
16 CLSTAT        1101 CLSTAT           CLSTAT
17 CONTRAST      1101 CNTRST           CONTRAST
18 CONT          0111 CONT             CONT
19 CORR          1111 CORR             CORR
1A PLIST         1101 PLIST            PLIST
1B CREATE        1101 CREATE           CREATE
1C ZERO          0000         xZERO    ZERO
1D DEFAULT       1101 DEFALT           DEFAULT
1E DROP          1101 DROP             DROP
1F DTH$          1111 HEX$             DTH$
20 ENDLINE       1101 ENDLIN           ENDLINE
21 ERRM$         1111 ERRM$            ERRM$
22 VER$          1111 VER$             VER$
```

23	EXACT	1101	EXACTT		EXACT
24	EXPM1	1111	EXPM1		EXPM1
25	EXPONENT	1111	EXPON		EXPONENT
26	EXTEND	0000		xEXTND	EXTEND
27	FLAG	1111	FLAG		FLAG
28	FLOOR	1111	INT		FLOOR (Same as INT)
29	FLOW	0000		xFLOW	(TRACE) FLOW
2A	FREE	0111	FRPORT		FREE (PORT)
2B	GDISP	1101	GDISP		GDISP
2C	GDISP$	1111	GDISP$		GDISP$
2D	HTD	1111	HXDEC		HTD
2E	INTO	0000		xINTO	INTO
2F	KEYDEF$	1111	KEYDEF		KEYDEF$
30	KEYDOWN	1111	KEYDWN		KEYDOWN
31	LC	1101	FLIP		LC
32	LGT	1111	LOG10		LGT
33	LOCK	1101	LOCK		LOCK
34	LOGP1	1111	LOGP1		LOGP1
35	WIDTH	1101	WIDTH		WIDTH
36	MATH	0000		xMATH	MATH
37	MEAN	1111	MEAN		MEAN (Duplicate of Built-in)
38	MEM	1111	MEM		MEM
39	MERGE	1101	MERGE		MERGE
3A	MINREAL	1111	MINRL		MINREAL
3B	NAN	1111	NAN		NAN
3C	NEAR	0000		xNEAR	NEAR
3D	NEG	0000		xNEG	NEG
3E	PCRD	0000		xPCRD	PCRD
3F	PEEK$	1111	PEEK$		PEEK$
40	POKE	1101	POKE		POKE
41	POP	1101	POP		POP
42	POS	1111	POS	xPOS	POS
43	PRIVATE	1101	PRIVAT		PRIVATE
44	PROTECT	1101	PROTCT		PROTECT
45	PUT	1101	PUT		PUT
46	PWIDTH	1101	PWIDTX		PWIDTH
47	RANDOMIZ	1101	RANDOM		RANDOMIZ(E)
48	RED	1111	RED		RED
49	RENAME	1101	RENAME		RENAME
4A	RENUMBER	1101	RENUM		RENUMBER
4B	RESET	1101	RESET		RESET [CLOCK]
4C	ROUND	0000		xROUND	ROUND
4D	SDEV	1111	SDEV		SDEV (Duplicate of Built-in)
4E	WINDOW	1101	WINDOW		WINDOW
4F	SECURE	1101	SECURE		SECURE
50	DISP$	1111	DSP$		DISP$
51	SETDATE	1101	SETDAT		SETDATE
52	SETTIME	1101	SETTIM		SETTIME
53	SHOW	0111	SHOW		SHOW (PORT)
54	SQRT	1111	SQR		SQRT
55	STARTUP	1101	STRTUP		STARTUP

```
56 TOTAL    1111 TOTAL           TOTAL
57 TRANSFOR 1101 TRSFMX          TRANSFORM
58 TRAP     1111 TRAP            TRAP
59 UNPROTEC 1101 UNPROT          UNPROTEC(T)
5A UNSECURE 1101 UNSECR          UNSECURE
5B VARS     0000         xVARS   (TRACE) VARS
```

6.1.3.2 Other Mainframe Symbolics

Existing symbolics for all the mainframe operators are defined as follows:

```
 t%     EQU   #85
 t&     EQU   #89
 t*     EQU   #83
 t+     EQU   #87
 t-     EQU   #82
 t/     EQU   #84
 tAND   EQU   #8B
 tDIV   EQU   #86
 tEXOR  EQU   #8C
 tNOT   EQU   #81
 tOR    EQU   #8D
 t^     EQU   #80
```

There are no existing symbolics for the relational operators, which are 3 nibbles long. However, each relational operator has for its first byte tRELOP (8A). The third nibble is a bit map:

```
        Relop         Bit#
        -----         ----
          <            1
          =            2
          >            3
          ?            4
```

Symbolics could be defined as follows:

```
  t<     EQU    (#1)~(tRELOP)
  t=     EQU    (#2)~(tRELOP)
  t>     EQU    (#4)~(tRELOP)
  t?     EQU    (#8)~(tRELOP)
  t<=    EQU    (#3)~(tRELOP)
  t>=    EQU    (#6)~(tRELOP)
  t#     EQU    (#5)~(tRELOP)
```

The following symbolics are available for loading up single characters of ascii. Symbolics for ascii are certainly not necessary, since:

```
            LC(2)  =a!       is equivalent to    LCASC  \!\
```

But here they are anyway:

```
a!      EQU    #21
a"      EQU    #22
a$      EQU    #24
a'      EQU    #27
a.      EQU    #2E
a0      EQU    #30
a1      EQU    #31
a2      EQU    #32
a3      EQU    #33
a4      EQU    #34
a5      EQU    #35
a6      EQU    #36
a7      EQU    #37
a8      EQU    #38
a9      EQU    #39
```

Note that if a symbolic is defined to be N nibbles long, and N+X nibbles are referenced, then the upper X nibbles are zeroes. For example:

```
                          LC(5)  =t<
  is equivalent to:       LCHEX  0018A
```

6.1.3.3 Building Symbolic Tokens For a LEX File

Given a one-byte token, xTOKEN, in a LEX whose ID# is FE, you could do the following to build the symbolic representation for the complete three-byte token:

```
tTOKEN EQU  (xTOKEN)~(#FE)~(tXWORD)
```

This builds tTOKEN by concatenating three bytes of information. The low byte is the XWORD token, the middle byte is the LEX ID, and the high byte is the token number in the table.

If xTOKEN were a function name, you would replace tXWORD above with tXFN. Analogously, if xTOKEN were a funny function, you would use tFFN.

6.2 Lexical Analysis, Parse, Execute

A language extension file contains tables used by the parse, decompile, and execution routines to recognize and execute external statements and functions. The TEXT table holds the ASCII string and associated token for each new or extended keyword. The optional SPEED table allows rapid searching of the TEXT table when a large number of keywords exists within the LEX file.

The message table within a language extension file contains messages related to routines and functions within the file. These messages may be error, warning, or system messages. See the "Message Handling" chapter for details.

The parse, decompile, and execution routines for external keywords and functions reside in the language extension file.

When searching for keywords, LEX files are searched first. This allows a BASIC statement to be extended beyond its definition in the mainframe. Correspondingly, LEX file functions can override main machine functions. New statements and functions can also be added in a LEX file.

As long as it contains all the necessary elements in the header, a LEX file may omit certain tables described here if its purpose does not require them. In particular, a LEX file may omit the message table if it's not needed. Or, as in the case of a foreign language translator, it may consist entirely of a message table which overrides mainframe messages (together with a poll handler which intercepts the pERR poll to do this). For details of foreign language message tables, see the chapter on "Message Handling."

6.3 LEX IDs and Entry #s

The token associated with an external keyword indicates that the keyword is either an XWORD (external BASIC keyword) or an XFN (external function). The lexical analyzer returns this token, along with the LEX ID (0-255) and the Entry # (0-255).

The LEX ID and entry# are stored in HEX. The LEX ID is used to locate the LEX file independently of what port it is plugged into. The entry# is the keyword# or function# used as the offset into the LEX file's main table and text table. For an external statement, the offset into the main table is used to obtain the parse, decompile, and execution addresses for the keyword; for an external function, the offset is used to obtain the number and type of parameters and the execution address. The relative offset into the text table is used tp obtain the ASCII text associated with the statement or function stored in the text table; this text is used to decompile the external keyword.

254 external LEX IDs are allowed. LEX ID 0 and 1 are reserved for the mainframe. 255 internal keywords and functions are allowed per LEX file. If a language extension requires more than 255 keywords,

then more than one language extension file must be used.

6.3.1 LEX ID Allocation

LEX IDs and entry# ranges are allocated by Hewlett-Packard. See the chapter "HP-71 Resource Allocation" for information on current resource allocations and the procedure for getting a token range officially allocated.

LEX IDs 92, 93 and 94 have been allocated as temporary/scratch IDs that can be used by LEX file developers who want a safe ID to experiment with without fear of interfering with LEX files written and distributed by Hewlett-Packard or other software developers.

6.3.2 Range of Entry Numbers

A LEX file may contain a contiguous range of entry numbers, allowing libraries of keywords to be distributed in logical groups. The format of the LEX file allows the range of entry numbers to be specified during creation.

6.3.3 Merging LEX Files

LEX files may be merged together for single file distribution of several LEX files. An internal LEX file chain exists within the LEX file structure.

6.4 Referencing Mainframe Entry Points

If HP's internally developed HP-71 linker is to be used after a file is assembled, entry points which are referenced external to the LEX file must always be preceded by '='. For example, GOSBVL =OUTBYT. Note that this is not true when using the FORTH/Assembler ROM, which does not use a linker.

In either case, all references to mainframe entry points must be absolute (GOVLNG or GOSBVL or LC(5)) since a LEX file may move in memory, thus prohibiting relative references.

In the interest of saving code, if a mainframe entry point is to be referenced several times from a LEX file, it is shorter to have only one external reference in the module to that entry point, with shorter relative jumps within the module to the point of external reference:

```
        GOSUB   outbyt
          .       .
          .       .
        GOSUB   outbyt
          .       .
          .       .
        GOTO    outbyt
          .       .
          .       .
outbyt GOVLNG =OUTBYT
```

6.4.1 LEX Files and Memory Movement

Any LEX file which is likely to reside in RAM (system or IRAM) faces a problem when invoking certain mainframe utilities which can cause files to move. For example, the utility to purge a file (PRGFMF) causes all subsequent files in a file chain to move to a lower address. In general, utilities which cause files to move are those which call some entry point in either the MOVEDM or MOVEUM routines; the other entry points in these routines are MOVED0, MOVEDA, MOVED1, MOVED2, MOVED3, MOVEDD, MOVEU0, MOVEUA, MOVEU1, MOVEU2, MOVEU3, and MOVEU4. Therefore, a given utility can be identified as one which causes memory to move by looking at its documentation header in Volume II of the IDS, and examining which routines it calls.

The danger of executing code in RAM, such as in a LEX file, is that it may invoke a system utility which moves the code, invalidating the return address on the CPU return stack and sending the machine to never-never land. To remedy the problem, a system utility has been created to allow calling mainframe utilities from movable code. The utility, MGOSUB, places the return address on the system GOSUB stack, where it will be updated if memory moves.

Because any unprotected LEX file in ROM can be copied to RAM, the above also applies to LEX files in ROM. However, if a LEX file in ROM is protected against being copied to RAM, then it does not need to be concerned with memory movement. There are two ways to guard against this: 1) Make the file Private, or 2) Give the LEX file a name with at least one lower case character. Of these two options, the first is probably preferrable.

6.4.2 MGOSUB Utility

This utility allows movable code (code running in RAM) to call utilities which may move it (such as the utility to purge a file). Rather than leaving the return address of the calling code on the CPU return stack, it places the return address on the BASIC GOSUB stack, where it is updated whenever memory is moved.

The MGOSUB utility is invoked as follows:

```
GOSBVL =MGOSUB
CON(5) <addr of target subroutine>
.. <code continues here> ..
```

The call to MGOSUB is transparent with regard to all registers, carry, SB, XM, and status bits. That is, entry conditions will be faithfully transmitted from caller to subroutine, and exit conditions will be faithfully transmitted from subroutine to caller. There is a price for this, however: the MGOSUB code uses SCRTCH RAM for temporary storage before and after the call to the target subroutine. This means that SCRTCH is not a safe place to keep things during the MGOSUB call, and that it cannot be used to pass data to or from the subroutine. Obviously, subroutines called via MGOSUB also pay an overhead in execution time.

6.5 Referencing Addresses in a LEX File

All references within a LEX file must be relative. If a table contained in a LEX file must be referenced, a way to get the current absolute address of the table is as follows:

```
              .
              .
       GOSUB  GTADDR        Push address of table onto stack
TABLE  NIBASC \HELLO\
       NIBHEX FF
GTADDR C=RSTK               Recall address of table
              .             Code continues
              .
```

6.6 External Lexical Analysis

Entry #0 in the Main Table of a LEX file contains the execution address of an external lexical analyzer or a system override.

An external lexical analyzer can be used to handle cases that cannot be handled by standard mainframe scanning techniques. If the token associated with a text item in the TEXT table is #00, an external lexical analyzer will be invoked. The external lexical analyzer will interpret the text using non-standard techniques and return a non-zero token to the mainframe lexical analyzer. Care must be taken to jump back to an appropriate reentry in the mainframe.

6.7 Entry and Display of External Keywords

When an external keyword is keyed in, the LEX file containing the keyword should exist. If the LEX file is in the machine during decompile, then upon decompiling the keyword the corresponding ASCII name is displayed. If the LEX file is not present during decompile, then one of the following is displayed:

XWORD lllee
XFNllleee

XFN indicates an external function; XWORD indicates some other external keyword. The first 3 digits (lll) are the LEX ID in decimal. Leading zeroes are suppressed. The last 3 digits (eee) are the keyword entry # in decimal. Three digits are always displayed. The LEX ID and entry# are stored in hexadecimal and displayed in decimal. The decimal display of LEX IDs corresponds to those displayed in error messages.

When an external statement is decompiled without the corresponding LEX file plugged in, only the XWORD text itself is decompiled; any text which would normally follow the XWORD is not displayed. An expression with an XFN from a missing LEX file is displayed normally, except that the ASCII function name is replaced with the XFNllleee notation; all parameters are displayed normally. Funny functions are an exception to this rule; their parameters are not displayed.

When a missing LEX file has added a new device type, the device type is decompiled as "external".

Note that in all cases, once the missing LEX file is plugged back in, decompiling resumes normally.

6.8 Short Keywords

If a short keyword in a LEX file is wholly contained within the first characters of a longer keyword in the same LEX file, special attention is required. The longer keyword should always precede the shorter keyword in the table, otherwise the longer keyword will NEVER be found.

Also, if a keyword exists in a LEX file that is wholly contained in the first characters of a longer keyword in the main machine or another LEX file, then the longer keyword will not be found unless the parse of the shorter keyword fails. To illustrate the two points made above:

FORM in LEX File
FO in LEX File

FOR in Main

If FO had preceded FORM in the LEX file above, then the FORM keyword would never be found.

Also note that only if FO parse fails, will the machine ever try FOR parse; this capability to try another parse routine once the parse of an external statement fails is provided through the RESTART mechanism.

Finally, assume the user types in the following:
>10 FORM=1 TO 5
Assume that FORM parse requires a string expression. FORM parse will fail; through the RESTART mechanism the FOR keyword in the mainframe table will be found next, and that parse will be successful. The Restart portion of line parse continues searching for a keyword if a LEX file returns an error condition from one of its parse routines. This ensures that longer keywords in other LEX files and in the mainframe are found.

The last example above illustrates that the RESTART mechanism continues the search in another LEX file, or if there aren't any more, into the mainframe. RESTART does not continue in the same table; this is why it's so important to put a longer keyword (FORM) prior to a shorter keyword (FO) when they occur in the same LEX file.

Parse routines that look for a particular keyword may have trouble using the lexical analyzer (NTOKEN) if a LEX file is present containing a shorter keyword than the one being searched for. For example, if a given parse routine requires the FOR keyword as an intermediate keyword, but FO is present in a LEX file, then NTOKEN will return tFO, not tFOR.

Using the WRDSCN utility gets you around this problem. WRDSCN was designed especially for searching all possible LEX files until a keyword that YOU specify is found. WRDSCN calls NTOKEN to find a lexeme. When NTOKEN returns a lexeme, then WRDSCN checks if it is one of the keywords that you have designated. If it is, WRDSCN returns that keyword; otherwise, it restarts the lexical analyzer, so that NTOKEN continues searching LEX files. Ultimately, WRDSCN either returns one of the keywords you have designated or indicates that the ascii pointed to by D1 does not contain any of the keywords you have specified (as indicated by LEX files present in the machine). See IDS Volume II for further details of WRDSCN.

6.9 Line Number References Within a Statement

Any statement which controls program flow using line number references, has a 5 nibble relative address field following tLINE#, so that the address can be compiled; note that commands such as LIST, which may contain line number references, would not have such relative offset fields, since LIST has nothing to do with controlling program flow. External statements containing line number references must exercise care when executing a line number reference.

A program can be edited or renumbered without a LEX file being present. But, if the LEX file is missing at the time the program is modified, any compiled addresses in the XWORD statements of that LEX file will not be cleared. Subsequent execution of such XWORD statements using this compiled address could result in an invalid branch.

There is an external entry in the Mainframe GOTO/GOSUB execution code. If the sXWORD status is set, the compiled line number address will be ignored and the line number will always be searched for, guaranteeing correct statement branching. See the GOTO documentation in Volume II of the IDS for details.

6.9.1 References Within an "Interrupt" Statement

A statement that branches to a line number due to an interrupt must execute special code to handle TRACE FLOW. Examples of interrupt statements are ON TIMER, ON ERROR and ON INTR.

Since the "TRACE FROM" address is not the preceding statement in sequential statement execution, the ONTIMR code must be duplicated to compute and trace the FROM address. The sXWORD flag must be set prior to the GOTO+ jump to guarantee all line number references are recomputed.

See the ONTMR documentation in Volume II of the IDS for details.

6.10 Polling

Polling is performed from many places in the HP-71 operating system to allow a LEX file to perform special processing when appropriate. During a poll, a one byte process number is passed to each LEX file; this identifies the reason the system is performing a poll.

Each LEX file has an opportunity to respond to a poll. The location of the poll handling code is identified by an offset-to-poll-handler which exists in each LEX file header. When a LEX file poll handler is polled (given control) it determines if it wants to respond to the process based on the process number. Response comes in several flavors:

1 - LEX file "handles" poll. The LEX file performs some processing and then returns with XM=0 and carry clear, indicating that the polling process should terminate.

2 - LEX file detects error (Slow Poll ONLY). The LEX file detects an error condition and returns with carry set, which terminates polling. An error identification is passed back in the C-register.

3 - None of the above. Many polls are NOT looking for a specific "handler", but are simply offering an opportunity for a LEX file to do some processing. For example, the pSREQ poll should never be "handled", but it allows an opportunity for a LEX file to handle whatever service requests it knows how to handle.

There are two kinds of polling: Fast and Slow. Their entry points are FPOLL and POLL, respectively. In both cases, the process number must immediately follow the call.

```
GOSBVL =FPOLL              GOSBVL =POLL
CON(2) =pPOLL#      or     CON(2) =pPOLL#
```

For both types of polling, XM can be set by the responding LEX file to indicate whether or not the poll was 'handled'. This is desirable if only one LEX file can respond to a particular poll; XM=0 on return to the system terminates the polling operation. In some cases it will be desirable for multiple LEX files to respond to a single poll; in this case responding LEX files should NOT set XM to 0.

The return requirements for a poll are indicated in the documentation for each separate poll, and can be found in the IDS Volume II under the individual poll name - pXXXXX.

6.10.1 Fast Poll

A fast poll is relatively fast and uses no extra memory. It is used when:

1) Execution speed is important, and/or
2) Little information is to be passed to the handler, and/or
3) There is little available memory or the memory may be in a strange state (e.g., pointers not valid).

The carry is set at entry to the LEX file poll handler, so fast polls are easy to detect. Typically, fast polls are used for low-level system polls, indicating a state within the machine, with no specific information to pass.

The process number is passed in B(A). D(A) should not be destroyed by a LEX file, since it is used as a pointer into the LEX file entry buffer during the polling process. However, if a LEX file is going to handle the process and exit with XM=0 (ensuring polling will stop), it is acceptable to destroy D(A). The poll handler is executing two stack levels deeper than the calling code.

Fast poll does nothing with R0-R4 and the status bits. Depending on the application, any or all of the above may be used to pass data to or from the handler. Information cannot be passed to or from poll handlers in A-D, D0, D1 or P. For specifics on register usage and availability, see the individual poll documention.

6.10.1.1 Fast Poll Example

A typical fast póll may look like the following:

```
        GOSBVL =FPOLL
        CON(2) =pPOLL#        Process #
               .
               .
```

Often, when a fast poll is issued, no distinction is made as to whether or not the poll was handled; in such cases it is not necessary to check XM.

6.10.2 Slow Poll

A slow poll allows passing of more information to poll handlers then does a fast poll. In addition, it saves stack levels and the contents of some registers in RAM, allowing recursive polling (a poll handler may perform a poll).

The advantages of slow poll over fast poll are:

1) Allows passing data to poll handlers in A,D,D0 and D1.
2) Handler can perform an error exit which will terminate the poll.
3) Stack levels are saved in RAM, so handler can
   a) Use more stack levels, and
   b) Call POLL itself.
4) Address of caller is saved on the GOSUB stack where it will be updated if memory moves.

The disadvantages of slow poll compared to fast poll are:

1) It's slower.

2) It requires enough memory and valid pointers to establish a save area in RAM.

As with fast poll, slow poll does nothing with R0-R4 and the status bits. Unlike fast poll, A, D, D0 and D1 can be used to pass data to the handlers. The contents of these registers are restored to their original entry values upon entry to each poll handler.

If a LEX file responds by "handling" the poll or performing an error return, most of the registers are returned to the caller as they were left by the handler. If no LEX file handles the poll, A,D,D0 and D1 are restored to their entry values upon return to the calling code.

6.10.2.1 Slow Poll Example

A typical slow poll may look like the following:

```
        GOSBVL =POLL
        CON(2) =pPOLL#         Process#
        GOC    Err             Error occured during handling?
        ?XM=0
        GOYES  OKAY            Process handled without error?
 * Process not handled at all
        LC(4)  =eXXYY          Load up appropriate err#
 Err    GOVLNG =BSERR          Error# loaded up
 OKAY
               .
               .
```

6.10.2.2 Save Stack Slow Poll Information

The save stack resides between the math stack and the FOR-NEXT stack. The SAVSTK pointer (same as FORSTK) points to the bottom of the save stack area. The following information is kept on the save stack during a slow poll:

```
        Register A             16 nibbles    Low Memory
        Register D             16
        Data Pointer D1         5
        Data Pointer D0         5
        Poll#                   5
        Return Level 2          5
        Return Level 3          5
        Rel Pos in LEX Buffer   5            High
SAVSTK-->
```

In addition to this save information, the calling return address is pushed on the BASIC GOSUB stack. This adds 6 nibbles to the stack pointed to by GSBSTK.

The total memory used by POLL is 68 nibbles (44 hex).

If a responder to a slow poll "takes-over" the poll and does not return to the caller, the POLL save information must be deleted. The math stack pointer should be collapsed to the FOR stack pointer. The mainframe routine =COLLAP will do this.

6.10.3 POLL Subroutine Level Usage

A handler for a fast poll is two subroutine levels deeper than the caller of the poll.

Because of subroutine level saving, a handler for a slow poll is one level shallower than the caller.

6.10.4 How to Answer a Poll

Each LEX file determines which poll process numbers it will respond to. As mentioned earlier, response may consist of handling, not handling, or returning an error. In each case, the availability of registers is clearly spelled out in the documentation for the individual poll.

The type of response is indicated by the poll handler in the state of the carry and the XM bit:

Handled: XM=0, carry clear.

Not handled: XM=1 (RTNSXM instruction), carry clear.

Error exit: (meaningful for POLL only, FPOLL ignores this):
Carry set.
Error number in C(3-0).

Each poll issued from the mainframe is documented to indicate entry and exit conditions for the poll. It is important that a responding LEX file follow the conventions indicated by the documentation.

6.10.5 Responding to a Poll from Binary

If a binary routine responds to a slow poll and does both of the following:

1. Indicates "no response" (XM=1) so the poll information is restored

2. Calls a BASIC subprogram during the poll

then the poll information and poll return address must be preserved during the CALL to BASIC. The return address to poll must be saved on the GOSUB stack, and the FORSTK pointer must be set over the poll save area. See the subsection on "Responding to POLL and Invoking BASIC" below for code examples.

6.10.6 Take-over Poll

If the handler of a slow poll "takes-over" by not returning to the operating system POLL routine, it should collapse the math stack to the FOR Stack to delete the saved poll information. The mainframe routine COLLAP will do this. In addition, the mainframe routine POPUPD should be called to pop the poll issuer's return address off the GOSUB stack.

6.10.7 Polling during Parse or Decompile

Any LEX file issuing a slow poll during parse or decompile must use the POLLD+ entry point. This entry adjusts the end of available memory value in D(A) to reflect the memory used by POLL.

AVMEMS (available memory start) must be set to the value in D0 to save data already written to the output buffer; this can be accomplished by calling AVS=D0. On return from poll, D(A) must be reset to the new available memory end. The routine D=AVME will do this.

Sample code:

```
        GOSBVL =AVS=D0       Set AVMEMS  D0
        GOSBVL =POLLD+       Issue Poll
        CON(2) =pPOLL
        GOSBVL =D=AVME       Set D = AVMEME
```

6.10.8 Polling from a LEX File in RAM

Polling from code which is executing in RAM can be tricky, since a poll handler may cause memory to move. If a poll handler can cause memory to move, a slow poll must be performed. Slow poll saves the address of the caller in a place where it will be updated if memory moves. Fast poll does not.

Poll (slow or fast) must be invoked DIRECTLY from a LEX file. The utility, MGOSUB cannot be used.

6.10.9 Summary of Poll Function Codes

The list of process numbers (poll function codes) and their meanings is maintained in the "HP-71 Resource Allocation" chapter. All polls issued by the mainframe are grouped within common categories (e.g., filetype polls, parse polls, card reader polls, etc.). System polls (those which identify a state of the system, such as going-to-sleep, waking-up, etc.) are assigned numbers in the upper range of possible process numbers (from 255 downward). Other polls are assigned process numbers upward from zero. As new process numbers are added for non-mainframe use, they will be assigned sequentially from the highest existing assigned process number.

It is this process number which is passed in the B-register to poll handlers in all LEX files.

See the "HP-71 Resource Allocation" chapter for a one line description of all system polls. See the POLL category in Volume II of the IDS for detailed information about individual polls.

6.10.10 Special Mainframe Polls

6.10.10.1 Pointer and Buffer "Clean-Up"

Whenever execution stacks are collapsed, the mainframe issues a fast poll, referred to as the zero program poll (pZERPG), to collapse any buffers and zero any pointers associated with program information. This happens whenever RUN, EDIT, or END are executed, or whenever the current file is modified or purged (any time the mainframe entry points CLRSTK, CLPSTK, or ZERPGM are called, this poll goes out).

A LEX file which uses a system buffer for a given application may want to answer the poll so that it can collapse or deallocate its buffer. The Math ROM, for example, keeps a copy of the math stack in its system buffer, so when the Zero Program poll (pZERPG) goes out, it responds by deallocating the buffer since the math stack no longer exists.

6.11 BIN Main Programs

A binary main program is a program written in HP-71 assembler language and invoked through the RUN statement. A binary main program can also be CALLed as a subprogram with no parameters.

Execution begins two nibbles past "20" (the equivalent to the EOL byte preceding the first statement of a BASIC program). Since common statements and utilities are used for both BASIC and BIN files, this "20" guarantees the same "start of file" length for both file types.

6.11.1 Ending a Binary Program

When execution of a binary program is complete, the code should GOVLNG =ENDBIN. This mainframe system entry point will "END" a binary program invoked through RUN/CALL. This entry point is part of the BASIC END statement execution. Stacks are collapsed, all open files are closed, the program running flag (PgmRun), the PRGM annunciator, and the status bits 0-11 are cleared, and pBSCex poll is issued. Control returns to the calling program or, to MAINLP if the binary program was invoked from the keyboard.

6.12 BIN Subprograms

A BIN subprogram is a subprogram written in HP-71 assembly code, with the tokenized BASIC SUB statement at the start of the code. The SUB statement is tokenized exactly like it is in a BASIC statement, except no line number is required. This tokenization allows binary subprograms to be CALLed just like BASIC subprograms.

Binary subprograms are used instead of BASIC subprograms to gain execution speed or system access not available to BASIC.

A BIN file containing only subprograms must have as its first command (preceding the first SUB statement): GOVLNG =ENDBIN. This guarantees standard handling of invoking RUN on a file containing nothing but subprograms - a NOP occurs.

For information on chaining of subprograms in a BIN file, see section on BIN files in the "File System" chapter.

See the section on SUB tokenization in the "Statement Parse, Decompile, and Execution" chapter.

6.13 BIN Error Exit

Invoking some mainframe routines from binary may result in a non-returning error exit through the mainframe message handler. The message driver jumps directly to ERRRTN at the end of the BASIC interpreter loop.

When an error occurs, BASIC program execution suspends. If the current program file type is not BASIC, the program is halted, but not suspended (the SUSP annunciator is not on so the program cannot be continued). The assumption made for suspending a BASIC program is that from the current D0 setting, the error line# can be found. For an error exit within a binary program, the D0 setting is meaningless; this is why the line# reported on an error within a BIN file is "~~".

If you want to cause a binary program or subprogram to suspend, respond to the pBSCex poll, which goes out each time the BASIC interpreter is exited; If the current file type is BIN and an error occurred (sERROR set), then you may want to set the SUSP annunciator and update CNTADR to point to the binary code to CONTinue at. See the pBSCex and PRUNnB poll documentation for further information.

6.14 Invoking BASIC from Binary

Binary programs and subprograms can be invoked through the RUN and CALL statements of BASIC. Provided the binary program or subprogram is formatted properly, invoking it is transparent to the user.

Likewise, it is possible to invoke BASIC from HP-71 assembly code. The entry point CALBIN is called. The PgmRun (S13) must be set before the call. Following the GOSBVL =CALBIN is the tokenized form of the BASIC CALL statement to the subprogram. The line length of the CALL statement starts the tokenization. See the section on CALL tokenization in the "Statement Parse, Decompile, and Execution" chapter.

Following the tokenized CALL statement is the next assembler instruction to be executed after the subprogram is ended.

6.14.1 Responding to POLL and Invoking BASIC

If a binary routine responds to a slow poll and does both the following:

1. Indicates "no response" (XM=1), so the poll information is restored and the poll continues

2. Calls a BASIC subprogram from within the poll handler

then the POLL information and poll return address must be preserved during the CALL to BASIC. The return address to POLL must be saved on the GOSUB stack, the FORSTK pointer must be set over the poll save area.

```
C=RSTK
```

```
A=C    A
GOSBVL =PSHUPD      Push return address on GOSUB stack
C=0    A
LC(2)  =1POLSV      Length of POLL Save area
D1=(5) =FORSTK
A=DAT1 A            Current FORSTK position
A=A-C  A            Move FORSTK over Poll save area
DAT1=A A
ST=1   PgmRun       Set prog running flag
GOSBVL =CALBIN      CALL BASIC
.....
```

On return from the BASIC subprogram, FORSTK must be readjusted and the POLL return address restored:

```
C=0    A
LC(2)  =1POLSV
D1=(5) =FORSTK
A=DAT1 A            Current FORSTK value
A=A+C  A            Adjust back
DAT1=A A
GOSBVL =POPUPD      Pop return address off stack
C=D    A
RSTK=C              Restore to stack
C=-C-1 A            Clear carry
RTNSXM              Return "not handled"
```

```
+------------------------------------------------+------------------+
|                                                |                  |
|   STATEMENT PARSE, DECOMPILE, AND EXECUTION    |   CHAPTER  7     |
|                                                |                  |
+------------------------------------------------+------------------+
```

7.1 Writing a Parse Routine

7.1.1 Statement Tokenization

Statement tokenization involves the calling of parse utilities to interpret the incoming ASCII stream as BASIC, and to convert and output it as a token stream. A BASIC program line begins with a line number and terminates with an End of Line token (tEOL). A program line may contain multiple statements. Subsequent statements in a multi-statement line are preceded by an @ (t@) token. Following each line number or @ token is a statement length byte. This statement length is a relative offset to the next terminating token (tEOL or t@). Statements within a BASIC file are chained together using these relative offsets.

In the following examples, assume that low memory is on the left and higher memory on the right.

7.1.1.1 Program Line

```
         +----------+    +-------------+
         |          |    |             |
         |          V    |             V
+-----+--+--+------+--+-----+----------+--+
|line#|StLen| Stmt |4F|StLen| Stmt     |0F|
+-----+--+--+------+--+-----+----------+--+
```

line# = Line number of program line
4 nibble BCD encoding

StLen = Statement length
1 byte offset to the end of the statement
Adding the address of the byte to the contents of the byte yields a pointer to @ (4F) or Endline (0F)

Stmt = Tokenized statement

Note that encoding of immediate execute lines is exactly as above, EXCEPT no line number is tokenized.

7.1.1.2 Program Line with Comment

Tokenization of a comment following a statement, using !, is included within the tokenization of the last statement. Therefore, the Statement Length byte preceding that last statement is an offset to Endline (0F):

```
        +---------+    +-----------------------+
        |         |    |                       |
        |         V    |                       V
+------+------+----+--+------+----+--+-------+--+--+
|line#|StLen|Stmt|4F|StLen|Stmt|t!|Comment|D0|0F|
+------+------+----+--+------+----+--+-------+--+--+
```

Note that ! is tokenized as CF, and that the comment itself is always followed by D0, then 0F (tEOL).

The tokenization for a comment at the beginning of a line (using REM or !) is analogous to that shown above; the comment is always immediately followed by D0. REM is tokenized as follows:

```
             +-------------------+
             |                   |
             |                   V
     +------+-----+----+-------+--+--+
     |line#|StLen|tREM|Comment|D0|0F|
     +------+-----+----+-------+--+--+
```

The tokenization for ! at the beginning of a line is the same as above, only substitute t! for tREM.

7.1.1.3 Program Line Containing Labels

Label identifiers are allowed within program lines. A label identifier is tokenized as a separate statement within the line. The Statement Length byte is an offset past the label tokenization, pointing to either @ (4F) or Endline (0F). A label is up to 8 characters of uppercase letters and digits, starting with a letter. A label token (tLBLST = 6F) precedes the ASCII label name.

For example, the following is the tokenization of a single statement line, with two preceding labels:

100 "ABC":'TEST1': GOSUB 525

```
          +-------------+    +-------------+    +---------+
          |             |    |             |    |         |
          |             V    |             V    |         V
+-----+-----+--+------+--+-----+--+------+--+-----+----+--+
|line#|StLen|6F|label |4F|StLen|6F|label |4F|StLen|Stmt|0F|
+-----+-----+--+------+--+-----+--+------+--+-----+----+--+
```

7.1.1.4 Multi-statement Line with Label

Tokenization of a multi-statement line, with a single label name following the first statement:

225 A=FNB(X) @ "ASSIGNA": KEY "A", A$;

```
        +---------+    +-------------+    +---------+
        |         |    |             |    |         |
        |         V    |             V    |         V
+-----+-----+----+--+-----+--+-----+--+-----+----+--+
|line#|StLen|Stmt|4F|StLen|6F|label|4F|StLen|Stmt|0F|
+-----+-----+----+--+-----+--+-----+--+-----+----+--+
```

7.1.2 Statements with Special Tokenization

7.1.2.1 IF...THEN...ELSE

Statements which immediately follow THEN or ELSE are in one of two categories: 1) Implied GOTO and 2) Extended IF. An implied GOTO does not contain 'GOTO', just the label or line number, as in:

```
      IF A THEN 100 ELSE LABEL1
```

Any statement immediately following THEN or ELSE which is not an implied GOTO is classified as an Extended IF Statement. There is a difference in the way these two classes of statements are tokenized. Note that the Extended IF token (tEXTIF) is simply the multi-statement token (t@ - 4F); the label reference token (tLBLRF) is E0; the line# token (tLINE#) is F0.

IF <expr> THEN PURGE

```
   +--------------------+        +---------+
   |                    |        |         |
   |                    V        |         V
+-----+---+------+-----+------+-----+----+--+
|StLen|tIF| expr |tTHEN|tEXTIF|StLen|Stmt|0F|
+-----+---+------+-----+------+-----+----+--+
```

IF <expr> THEN 100

```
    +-------------------------------+
    |                               |
    |                               V
 +-----+---+------+-----+------+----+--+
 |StLen|tIF| expr |tTHEN|tLINE#|0010|0F|
 +-----+---+------+-----+------+----+--+
```

IF <expr> THEN <string expression>

```
    +--------------------------------------+
    |                                      |
    |                                      V
 +-----+---+------+-----+------+-----------+--+
 |StLen|tIF| expr |tTHEN|tLBLRF|string expr|0F|
 +-----+---+------+-----+------+-----------+--+
```

IF <expr> THEN PURGE ELSE "ABC"

```
    +------------------+         +--------+
    |                  |         |        |
    |                  V         |        V
 +-----+---+----+-----+------+-----+----+--+
 |StLen|tIF|expr|tTHEN|tEXTIF|StLen|Stmt|4F|  (continued below)
 +-----+---+----+-----+------+-----+----+--+

             +-------------------------+
             |                         |
             |                         V
          +-----+-----+------+---------+--+
          |StLen|tELSE|tLBLRF|strg expr|0F|
          +-----+-----+------+---------+--+
```

So far only label references which are string expressions have been shown; also legal are 'literal' label references. They are tokenized with a tLITRL (4C) preceding them.

IF <expr> THEN ABC

```
    +-------------------------------------------+
    |                                           |
    |                                           V
 +-----+---+----+-----+------+------+-----------+--+
 |StLen|tIF|expr|tTHEN|tLBLRF|tLITRL|ascii label|0F|
 +-----+---+----+-----+------+------+-----------+--+
```

```
IF <expr> THEN A=B @ RETURN ELSE 10
    +------------------+          +---------+     +---------+
    |                  |          |         |     |         |
    |                  V          |         V     |         V
+-----+---+----+-----+------+-----+----+--+-----+----+--+---
|StLen|tIF|expr|tTHEN|tEXTIF|StLen|Stmt|4F|StLen|Stmt|4F...
+-----+---+----+-----+------+-----+----+--+-----+----+--+---

          +---------------------+
          |                     |
          |                     V
     -----+-----+------+-----+--+
  ...StLen|tELSE|tLINE#|0100|0F|
     -----+-----+------+-----+--+
```

7.1.2.2 CALL

The simplified tokenization of CALL is as follows:
tCALL [<name> [tPRMST<parm list>] tPRMEN [tIN<file name>]]

The simplest form of the CALL statement takes no parameters. The multi-statement line:

CALL @ CALL <subprogram name>

would be tokenized as follows:

```
 +---------+  +---------------------+
 |         |  |                     |
 |         V  |                     V
+--+-----+--+-----+-----+----+------+--+
|40|tCALL|4F|StLen|tCALL|name|tPRMEN|0F|
+--+-----+--+-----+-----+----+------+--+
```

Note that the statement length of the first statement is only 4 nibbles.

Next, look at the tokenization of the CALL statement with parameter passing.

```
        CALL <name>(PV,PR,#5)
```

would be tokenized as follows (assuming PV is a pass by value & PR represents a variable which will be passed by reference):

```
   +----------------------------------------------------------+
   |                                                          |
   |                                                          V
+-----+-----+----+------+--+-----+--+-----+--+--+-----+------+--+
|StLen|tCALL|name|tPRMST|PV|tCVAL|PR|tCREF|t#|53|tCREF|tPRMEN|0F|
+-----+-----+----+------+--+-----+--+-----+--+--+-----+------+--+
```

Note in this example that each parameter is followed by a 1-byte token, indicating whether it is a pass by value (tCVAL) or a pass by reference (tCREF). Channel numbers are encoded somewhat non-intuitively as a pass by reference. Any parameter list of a CALL statement is preceded by tPRMST (Parameter Start); the list is terminated by tPRMEN (Parameter End). Every CALL statement (except the one with no subprogram name or parameters given) is terminated by tPRMEN.

This example illustrates the tokenization of a CALL which specifies a file.

```
      CALL <name> IN <file name> @ CALL <name>(PV) IN <file name>

        +-----------------------------------+
        |                                   |
        |                                   v
     +-----+-----+----+------+---+---------+--+
     |StLen|tCALL|name|tPRMEN|tIN|file name|4F|...
     +-----+-----+----+------+---+---------+--+

        +---------------------------------------------------------+
        |                                                         |
        |                                                         V
     +-----+-----+----+------+--+-----+------+---+---------+--+
 ... |StLen|tCALL|name|tPRMST|PV|tCVAL|tPRMEN|tIN|file name|0F|
     +-----+-----+----+------+--+-----+------+---+---------+--+
```

When the subprogram name is specified as a string variable or quoted string, it is tokenized either as the variable or in ascii (quotes included). However, when the subprogram name is given as an unquoted string it is tokenized with a preceding byte: tLITRL. For example:

```
     CALL "AB" @ CALL AB

   +-------------------------+    +----------------------------+
   |                         |    |                            |
   |                         V    |                            V
+-----+-----+--------+------+--+-----+-----+------+----+------+--+
|StLen|tCALL|22142422|tPRMEN|4F|StLen|tCALL|tLITRL|1424|tPRMEN|0F|
+-----+-----+--------+------+--+-----+-----+------+----+------+--+
```

7.1.2.3 SUB

The tokenization of the SUB statement is similar in many ways to that of CALL; however, CALL does not output comma tokens between parameters, whereas SUB does. Also, the SUB statement has two 5-nibble fields which are used for chaining. The first field immediately follows tSUB, and the second field immediately precedes either t@ or tEOL (depending on which token follows the SUB statement).

If the SUB statement is followed by !, then the second field immediately FOLLOWS the tokenization of the comment.

The tokenization is as follows:
tSUB<xxxxx><name> [tPRMST <parm list>] tPRMEN [t! comment] <xxxxx>

Note that in all cases, the subprogram name in a SUB statement is preceded by tLITRL. Following are some examples of the tokenization of SUB.

SUB <name> ! comment

```
    +------------------------------------------------------+
    |                                                      |
    |                                                      V
+-----+----+-----+------+----+-----+--+-------+--+--+-----+--+
|StLen|tSUB|xxxxx|tLITRL|name|tPRMEN|t!|comment|DO|OF|xxxxx|OF|
+-----+----+-----+------+----+-----+--+-------+--+--+-----+--+
```

SUB <name> (PV,#5) @ BEEP

```
    +------------------------------------------------------->...
    |
    |
+-----+----+-----+------+----+-----+--+------+--+--+------+
|StLen|tSUB|xxxxx|tLITRL|name|tPRMST|PV|tCOMMA|t#|53|tPRMEN|...
+-----+----+-----+------+----+-----+--+------+--+--+------+

    ...----+   +----------+
           |   |          |
           V   |          V
    +-----+--+-----+-----+--+
 ...|xxxxx|4F|StLen|tBEEP|OF|
    +-----+--+-----+-----+--+
```

7.1.2.4 IMAGE

Parsing of an image string is performed at the time the USING statement is executed.

There are no special considerations for parsing the IMAGE keyword, on the level of the BASIC interpreter. An IMAGE statement is tokenized as follows:

```
              +-------------------------+
              |                         |
              |                         V
    +-----+-----+------+------------+--+--+
    |line#|StLen|tIMAGE|image string|DO|OF|
    +-----+-----+------+------------+--+--+
```

Similarly, a USING statement (for example, DISP USING "<image string>", or DISP USING <line#>), is tokenized with the image string as an expression, or a tLINE# token to reference the IMAGE statement.

Parsing of the image string must be performed at the time the USING statement is executed, since the image string expression can be changed during execution (consider DISP USING S$; <output list>). IMAGE syntax is of a peculiar design -- that is, its rules are not governed by the BASIC interpreter. In addition, image parsing is inextricably linked to its execution. For these reasons, image parsing is entirely separate from BASIC interpreting. For a detailed description of the tokenization of image strings, see IDS Volume III, module MB&IMG.

7.1.3 Global Assumptions

Status bits:

S4 - No Restore of Input Pointer
Used by error handler to determine if RESPTR should be called for correct cursor position.

S5 - Line Number on Line
Program line, as opposed to immediate execute line.

S6 - Pending THEN
Within the scope of an IF-THEN clause, and ELSE has not yet been encountered (ELSE is a legal terminator at this point; IF is not legal).

S10 - Implied LET Error
Used by error handler to determine if statement parsed was being interpreted as an Implied LET (If S9 is set, then attempt to parse as label, else attempt to parse as implied DISP).

flRTN- System flag indicating that parse is externally invoked.

Registers
D(A) End of Available Memory; used to check against when outputting tokens.

Statement Scratch Ram:

S-R0-2 When flRTN is set (indicates parse is externally invoked), this RAM location contains the address to return to.

S-R0-3 IF Statement in progress. All statements following THEN and preceding Endline are in this realm. Set if nonzero.

STMTD0 RESTART Input Pointer
When the RESTART flag is set, the position of D1 prior to the call to the lexical analyzer (contents of LEXPTR) is saved. D1 is restored from this ram location prior to restarting the lexical analyzer.

S-R1-0 Original Error Number
If a keyword is to be restarted and has not been previously restarted, then this is where the error number is saved. When a keyword has not been restarted previously, this location is zero.

S-R1-1 Original Error Position
At the same time Original Error Number is saved, the error position address is saved.

S-R1-2 RESTART Address
Each time the lexical analyzer is called to evaluate a lexeme at the beginning of a statement (or immediately after THEN or ELSE), its restart address is saved. If the RESTART flag is set, then the error handler restarts the lexical analyzer with this address.

S-R1-3 RESTART Flag
If the lexeme at the beginning of a statement is an XWORD or XFN, this flag is set; otherwise it is cleared. Set if nonzero.

7.1.4 Entry Conditions from Line Parse Driver

D1 points to the first character following the keyword. D0 points into the output buffer, past the statement length byte and the keyword token. Status bits 0, 8, 9, and 10 are clear.

7.1.5 Exit Conditions

All parse routines which do not error exit, must return with carry clear. Carry set is reserved for 'middle of IF' return.

D1 should be pointing past the last legal character or keyword accepted as part of the legal parse, but no farther. In many cases this requires a RESPTR to be done before returning - this can be accomplished by ending a parse routine with: GOVLNG =RESPTR. For example, if an optional keyword is searched for with NTOKEN but not found, D1 must be backed up. Note that if GNXTCR had been called instead of NTOKEN, this wouldn't be necessary since GNXTCR does not move D1 past any non-blank character.

D(A) should still hold the End of Available Memory.

Whenever information is output to the Output Buffer at the D0 pointer) through the OUTxxx utilities, available memory is checked to make sure there is enough memory to write out the information. If there is not enough memory, an "Insufficient Memory" error is generated.

If the Parser was invoked externally, the Message Driver returns to the caller, instead of taking a hardwired exit.

7.1.6 Parse Errors

The following entry points already exist for parse errors. If S4=1, D1 is expected to be pointing at the input in error; otherwise RESPTR will be called to position D1 at the previous input, assumed to be the error.

SYNTXe	Syntax	
IVEXPe	Invalid Expression	
IVPARe	Invalid Parameter	*
MSPARe	Missing Parameter	*
IVVARe	Invalid Variable	
ILCNTe	Illegal Context	
EXCHRe	Excess Characters	
QUOEXe	Quote Expected	
PRNEXe	) Expected	
FSPECe	Invalid Filespec	

* If IVPARe is used, and there is no remaining input in statement (after optional RESPTR, D1 points at @, !, ELSE, or EOL), then MSPARe is issued.

If it is necessary to generate a parse error other than one listed above, load the low 4 nibbles of D0 with the error number and GOVLNG =PARERR.

NOTE: For MOST parse error exits, S10 should be clear; S10 is the Implied LET error flag.

If more details are needed to generate specific parse errors, see the chapter, "Message Handling", or the header for the MFERR* routine.

7.1.6.1 Relinquishing Error Handling

In some cases it is desirable for a LEX file parse routine to not report its error message and position, but to give control BACK to the mainframe and let the mainframe report the error. An example of such a case is as follows:

Consider the mainframe routine ON TIMER; further consider what happens when the user has HPIL plugged in, and incorrect syntax is used with this statement. For example:

```
ON TIMER %1,1 GOSUB 50
```

Here's the scenario: ON INTR (an HPIL statement) errors out in the normal way (causing its error information to be saved); the parse is restarted, ON TIMER also errors, and the error information generated by HPIL is restored and reported to the user, resulting in some obscure message like

HPIL ERR: Invalid Parm,

with the cursor flashing on TIMER. Obviously, this is less than desirable.

By using the REST* entry point, the LEX file error is forever forgotten, and the mainframe-generated error is the one reported (or any parse error previously or subsequently reported in the 'normal' way).

In short, this entry point enables language extensions to suppress their particular error message/error position, providing it is KNOWN that a parse routine exists in the mainframe which will gain control when the parse is restarted and which has the capability of giving a more coherent error message.

To use this feature when a parse error is detected, simply do a GOVLNG =REST*.

7.1.7 Expression Tokenization

Expressions specified in statements are converted to RPN (postfix notation) by the expression parser and are stored in this format. In this form, the expression is a series of tokens. The tokens are described next.

7.1.7.1 Constants

Single-digit constants are tokenized as the ASCII character code for that digit. ("0" thru "9")

Integer constants (2-12 digits) are tokenized by a byte which identifies the number of digits in the constant followed by a nibble for each of the digits. The digits are stored least significant digit first.

Floating point constants (1-12 digits) are tokenized by a byte which identifies the number of digits in the mantissa of the constant followed by a nibble for each of the digits. The digits are stored least significant digit first. Following this is a 3 nibble 9's complement exponent.

String constants (single or double quoted strings) are tokenized as the opening quote with the enclosed characters following and are terminated with a matching closing quote.

7.1.7.2 Variables

Variables are tokenized in one to three bytes as follows:

[t$] [tADIGx] Alpha

Where the t$ token is present if its a string variable, the tADIGx token is present if the variable has a digit character after the letter and alpha is always present and encoded as the ASCII code for that letter. There are ten possible tADIGx tokens (tADIG0 - tADIG9) corresponding to the ten possible digits.

7.1.7.3 Operators

Operators (monadic and dyadic) are tokenized with a single byte except for the relational operators which have a nibble following the first byte to identify the specific relation.

7.1.7.4 Functions

Functions are divided into four groups:

Mainframe functions -- These are tokenized as a single byte.

XFN's -- These are tokenized as an tXFN token followed by a byte identifying the LEX ID and another byte specifying the entry number within that ID. Following these three bytes is a nibble which says how many parameters this function reference actually has.

Arrays -- The tokenization of arrays is a hybrid of variable and XFN tokenization. A tARRAY token is followed by one to three bytes that describe the name of the array (same as for variables) and this is followed by a nibble describing the number of subscripts.

Funny Functions -- This type is used for functions which defy normal rules for parse or execution. The tokenization is described in the next section.

Following any parameterless function a tLPRP token may be present to preserve a "()" which followed the function.

Any token other than those above signals the end of the expression.

7.1.8 Funny Function Parse

The lexical analyzer (NTOKEN) finds the keyword corresponding to the FFN in a lex table. It detects that its token number is 00. It jumps to the "execution address" of token 00. This routine figures out what token should be returned by looking at the letters of the text (or maybe some pointer the lexical analyzer passes to it) and leaves that in A(5-0) in the form:

```
                       5 4  3 2  1 0
      +-----------------+----+----+----+
   A: |                 |Fn# | Id |tFFN|
```

+------------------+----+----+----+

It also loads B(A) with the address of a routine (in that lex file) which knows how to parse that FFN. This will be called by the expression parser if indeed the expression parser was the one who called NTOKEN. It should set status bits to look like a parameterless function (S0-S3 clear).

It then returns. This is actually the return from NTOKEN.

If it wasn't the expression parser who called NTOKEN then the entry returned simply looks like a function and the parse routine can give the same parse error that it would give if any other function was found. CALC mode has a specific trap for the tFFN token and disallows it.

The expression parser eventually sees the tFFN token and jumps to the address returned in B(A). Before jumping, it compiles the 8 nibbles in A(7-0). This leaves room for the length byte to be filled in. D0 (the output pointer) points past these eight nibbles, ready for the FFN parse to take over. D1 (the input pointer) points wherever it was left by the lexical analyzer override routine described above. D(A) points to the parse stack. This stack must be preserved. It extends from D(A) to AVMEME. The FFN parse routine must respect the register usage of the expression parser.

If the expression parser must be reentered to parse an expression within the FFN, AVMEME must be moved up to "protect" the parse stack. This implies that the stack length must be saved so that AVMEME can be set back to its original value. In order to be able to fill in the FFN length when it is done parsing it, D0 should be saved also. One subroutine level should also be saved to prevent overflowing the stack. If these three items (parse stack length, D0 pointing past the length byte and one return stack level) are saved on the parse stack before moving AVMEME to protect the stack, then unlimited nesting of FFNs is possible.

The net effect of the FFN parser is to parse a "parameterless" function. This implies that no parameters precede the function in the RPN stream of tokens. Once the FFN has been completely parsed, control should passed back to the expression parser in the state where an operand has just been found (P1-10). It should return to SE1-10 if the FFN returns a string result. This pushes a "Primary" on the parse stack and scans for another token. In either case it should do a RTNSXM to indicate that this is a value expression. The expression parser continues, trying to work this primary into the expression.

The CALL statement expects the expression parser to set the RAM nibble at PRMCNT to a non-zero value if the expression contains any function that can possibly cause another call statement to be executed. In the mainframe, only user-defined functions can cause this to happen. It is conceivable that a funny function could perform a CALL on its own. In this case, the PRMCNT nibble should be set to prevent a problem with call. There is no problem if the expression parser is used recursively since if the expression which is a parameter to the funny function contains a user-defined function, that "copy" of the expression parser will set the PRMCNT nibble and it will remain set for the duration.

The only acceptable error exit in the process described above is the case of insufficient memory to continue normally; the routines must return in all other cases.

7.1.8.1 Funny Function Tokenization

The "Funny Function" token (FFN) lies just within the range of built-in functions. This token (tFFN) is encoded as follows:

```
+----+----+----+----+---------------------+
|tFFN| Id |Fn# |Len |  Funny code         |
+----+----+----+----+---------------------+
                 |                         ^
                 |                         |
                 +-------------------------+
```

First comes the tFFN token followed by the Lex Id and the function number, just as in XFN. Following this, there is a length byte. This byte, when added to its own address points to the first nibble of code not contained in the FFN.

7.1.9 Polling during Parse

A statement issuing a poll (slow poll) during parse must use the POLLD+ entry point. This adjusts the end of available memory value in D(A) to reflect the save area and GOSUB stack level used by poll.

AVMEMS (available memory start) must be set to the value in D0 in order to preserve data already written to the output buffer; this can be done by calling AVS=D0. On return from the poll, the calling routine must reset D(A) to available memory end. The routine D=AVME does this.

Sample code:

```
        GOSBVL =AVS=D0      Set AVMEMS  D0
        GOSBVL =POLLD+
        CON(2) =pPOLL#      Issue poll
        GOSBVL =D=AVME      Set D(A) = AVMEME
```

7.2 Writing a Decompile Routine

7.2.1 Global Assumptions

INADDR - Contains pointer to statement length byte of statement currently being decompiled.

LDCSPC - Contains pointer to desired cursor position in decompiled line (immediately following line number).

sSSTdc - SST Flag (S1) - Set ONLY by Single Step to decompile only a statement not the entire line.

S12-S15 - Global System Flags - Except (S12), PgmRun (S13), NoCont (S14), Trace (S15)

f1RTN - System flag which indicates that decompile was externally invoked.

S-R0-2 - When f1RTN is set, this RAM location contains the address to return to.

R3 Used by LIST; not available to decompile routines

7.2.2 Entry Conditions from Line Decompile

D1 points into the token stream. D1 is past the keyword token; A and C contain the next token.

D0 points into the output buffer, past the decompiled line number, keyword, and a blank.

D(A) contains the End of Available memory; used to check against by the output routines. This value should remain untouched.

7.2.3 Decompile Utilities

For output utilities, see "How To Write a Parse Routine."
GTEXT1 - Given a token, outputs the corresponding text. Includes numerous entry conditions and entry points which provides for outputting leading and/or trailing blanks.

EOLDC - Checks for statement terminators: t@, t!, tEOL

EOLXC* - Calls EOLDC above; if statement terminator found, does not return - handles rest of statement by going to OUTELA. If no statement terminator found, returns to caller with carry clear.

VARDC - Decompiles variables

LIN#DC - Decompiles and outputs a line number, suppressing leading zeros.

ASCICK - Copies ascii characters from input stream to output buffer, until encountering a non-ascii character.

EXPRDC - Decompiles expression pointed to by D1.

FILDC - Decompiles file specifier

ARYDC - Decompiles array which was compiled by ARRYCK.

LABLDC - Assumes D1 is at tLBLRF (label reference token), steps over tLBLRF. If label is a literal, outputs it within quotes; otherwise, the string expression is decompiled. Returns with carry clear.

SKIPDC - Useful if an unrecognized XWORD is encountered; skips D1 to the end of the statement and goes to OUTELA (see below).

7.2.4 Exit Conditions

When the token stream has been exhausted, exit through either OUTEL1 (D1 points to statement terminator) or OUTELA (D1 points to statement terminator and A(B) contains it).

D(A) points to the end of available memory.

7.2.5 Existing Multi-use Decompile Routines

Any keywords which have no parse to speak of (STOP and RETURN are good examples), can use OUTELA as their decompile routine.

Any keywords which have an optional expression list, delimited with compiled commas and/or semi-colons may use DROPDC as their decompile routine. Note that this can be used even if no delimiters are compiled between expressions: the expression list is still output with comma delimiters.

Any keywords which have a mandatory expression list may use FIXDC as their decompile routine. Again, delimiters need not be compiled between expressions; comma and semi-colon delimiters are acceptable and will be decompiled.

7.2.6 Funny Function Decompile

When expression decompile sees a tFFN token, it outputs a nullop and it looks up the execution address of the FFN. If the FFN can't be found (ie the lex file is missing) it pretends the token is a tXFN and outputs XFNllleee, where lll is the LEX ID (leading zeroes suppressed), and eee is the entry#. It skips over the FFN by adding the FFN length. No attempt is made to decompile the FFNs parameters.

The decompile handler for this FFN is pointed to by a relative address immediately above the execution address. The FFN decompile handler should decompile the FFN as only it knows how. This decompile cannot leave unquoted characters greater than 127 in the buffer since this would mess up the decompiler when it is resumed.

If the FFN contains an expression, it will have to preserve some information to be able to call expression decompile; it will have to steal some available memory at AVMEME to preserve the pointers which are critical to the expression decompile which is in progress. It will also have to save one stack level.

Once the entire FFN has been decompiled, control should be passed back to the main expression decompile loop (via a GOVLNG =EXDCLP). The expression decompile should continue normally looking at the rest of the expression. The text that has been generated will be treated as a parameterless function with a very long name.

7.2.7 Polling during Decompile

A statement issuing a poll (slow poll) during decompile must use the POLLD+ entry point. This adjusts the end of available memory value in D(A) to reflect the save area and GOSUB stack level used by poll.

AVMEMS (available memory start) must be set to the value in D0 in order to preserve data already written to the output buffer; this can be done by calling AVS=D0. On return from the poll, the calling routine must reset D(A) to the current value of available memory end. The routine D=AVME will do this.

Sample code:

```
        GOSBVL =AVS=D0     Set AVMEM at D0
        GOSBVL =POLLD+     Issue poll
        CON(2) =pPOLL#
```

```
        GOSBVL =D=AVME    Set D  AVMEM
```

7.3 Statement Execution

7.3.1 Entry Conditions

The program counter (D0) is positioned past the begin BASIC token. PCADDR has been updated and points at the statement length byte for the statement.

7.3.2 Global Assumptions

Several flags have global meaning during statement execution:

Except	(S12)	Exception has occured
PgmRun	(S13)	Program Running
NoCont	(S14)	No Continue of execution
Trace	(S15)	TRACE Mode active

PgmRun (S13) is set if a program is executing. NoCont (S14) is set if execution is to halt after the next statement is executed. Single step execution sets this flag.

7.3.3 Exit Conditions

When the execution associated with a given statement is complete, control must be turned over to the run loop. This is done by exiting through NXTSTM or RUNRTN.

NXTSTM - Skips over statement preceded by current PCADDR. The statement following will be the next one to execute.

NXTST2 - D0 points to statement length byte of statement to skip over.

RUNRTN - D0 points to statement terminator (t@,tEOL,tELSE) preceding next statement to execute. Be sure sENDx (S1) is clear.

RUNRT1 - D0 points to statement terminator (t@,tEOL,tELSE) preceding the next statement to execute. sENDx is explicitly cleared.

7.3.4 Error Exits through MFERR/BSERR

Error exits from statements and functions require only four things:
1) S13 is set when appropriate (indicates program running)
2) PCADDR is accurate
3) The error number is loaded in C.
4) P is set appropriately to select options (set ERRN, display error prefix, etc.). See MFERR* documentation or the "Message Handling" chapter for details.

Entry points MFERR and BSERR are used for processing errors generated in the BASIC operating system. MFERR requires that the error number is loaded in C(B); this error exit can be used for mainframe generated errors (LEX file #00). However, BSERR requires that the error number is loaded in C(3-0), specifying both the LEX ID number and the message number. It is acceptable to use BSERR for mainframe- generated errors, as long as C(3-2) is filled with zeros.

7.3.5 Use of Available Memory by Statements

The execution of statements often requires the usurping of available memory. There are some restrictions on how much of available memory may be allocated and for how long. Refer to the section Available Memory Management in the "Memory Structure" chapter for details.

7.3.6 Statement Execution Utilities

FSPECx - Evaluates file specifiers; will poll for any not recognized by mainframe.

FILXQ^ - Evaluates mainframe file specifiers and dedicated device specifiers. Currently accepted device names are PORT, MAIN, CARD, and PCRD.

EXPEXC - Evaluates expression pointed to by D0. Evaluated expression on stack. See EXPEXC documentation for details.

FINDF - Given a file specifier returned from FSPECx or FILXQ^, searches for the given file. Indicates upon exit, whether or not file found. If file found, provides information on where. Numerous entry points.

EOLXCK - Given a token in A(B), returns with carry set if it is a statement terminator: tEOL, t@, t!, tELSE.

7.4 Expression Execution

7.4.1 Entry Conditions to Expression Execute

D0 is the interpreter's program counter; it must point to the first token of the expression when expression execution is called. D1 is the active stack pointer for the operand stack during execution.

Several entry points are available:

EXPEX- collapses the math stack, but leaves status bits alone.

EXPEX+ saves the caller's status bits, and reads MTHSTK to position the stack pointer.

EXPEXC leaves status bits alone, and reads MTHSTK to position the stack pointer. EXPEX1 is another name for this entry point.

EXPR assumes the stack pointer is already positioned.

7.4.2 Math Stack Usage and Format

The math stack grows from high addresses to low. The stack item at the lowest address is said to be on top of the math stack. MTHSTK is updated only upon termination of expression execution, or for special cases such as user-defined function execution.

7.4.3 Data Types on the Stack

There are four kinds of objects that exist on the math stack under normal circumstances:

- Real numbers
- Complex numbers
- Strings
- Array descriptors

Real numbers exist on the math stack in standard floating-point form. They can be identified by a legal BCD digit on top of the stack.

```
        Real number on stack
        +-+-------------+---+
 High   |S| Mantissa    |Exp|   Low
 mem    +-+-------------+---+   mem
         1      12        3
```

Complex numbers consist of an E-digit on top of the stack, with a zero-digit just below it. This is the complex stack signature. Below the stack signature are two standard floating-point numbers: the imaginary part on top of the real part.

```
                Complex number on stack
          Real part       .  Imaginary part
        +-+----------+---+-+-------------+---+--+
 High   |S| Mantissa |Exp|S| Mantissa    |Exp|0E|   Low
 mem    +-+----------+---+-+-------------+---+--+   mem
```

Strings have an F0 stack signature. Below the signature is a five-nibble field giving the length of the string in nibbles. Then come nine nibbles which can normally be ignored; they contain destination information for string assignment if they contain anything useful at all. This information includes the maximum string length and the address of the destination. Hence, a string stack header consists of 16 nibbles; the ASCII text of the string lies under the header, with the first character of the string toward the bottom of the stack, and the last character next to the header.

```
                    STRING on stack
        +-------------+-----+-------+------+--+
 High   |String ...   |MaxLn|Address|Length|0F|   Low
 mem    +-------------+-----+-------+------+--+   mem
                         4      5       5    2
```

A string may have another representation on the stack if it was created by pushing an element of a nonexistant string array. In this case, the tag is a F8 instead of F0. The length field will indicate a null string. The name of the variable referenced and the element number will be filled in. This is treated as a null string by system routines. This item is 16 nibbles in length with the following format:

```
        Nonexistent string array element on stack
       +-----+--+----+------+--+
 High  |Ele# |00|Name|Length|8F|  Low
  mem  +-----+--+----+------+--+  mem
          4   2   3    5    2
```

Any other object on the stack must be an array descriptor, with its offset field changed to the absolute address of the array's data area.

```
       Array descriptor on stack
       +-------+------------+-+-+-+
 High  |Address|Dim lengths|b|#|t|  Low
  mem  +-------+------------+-+-+-+  mem
          5          8      1 1 1

    b=Option base , t=Type
```

7.4.4 Expression Execution Utilities

Utilities exist for popping and type-checking arguments, along with reentry points for pushing results.

POP1N and POP2N are used for popping numeric arguments. Attempting to pop a string or an array descriptor with these routines causes an error to occur. If the carry is set upon return from these routines, the arguments are complex.

MPOP1N and MPOP2N establish the math modes, pop arguments, and test for exceptional inputs before returning. These utilities all leave the stack pointer (D1) positioned for placing a standard floating-point number back on the stack.

POP1S tests for a string on the stack. Attempting to pop a number or array descriptor with this routine causes an error to occur. Upon return, the string length is left in the lower 5 nibbles of the A-register, with the stack pointer (D1) at the topmost character of the string text.

REVPOP has the same exit conditions as POP1S, but the string is reversed before returning. REV$ is a string reversal routine, which returns with the stack pointer unaltered.

7.4.5 Function Returns

Reentry points are called function returns. The mainframe code has established function returns for real numbers only. FNRTN1 assumes the PC is still in D0, and the result is in C. FNRTN2 assumes the PC has been moved to A, with the result in C. These two function returns are for placing new items, such as constants, on the stack; a stack collision check is performed. These returns are generally NOT used for functions which have arguments, since the stack pointer is usually already where it needs to be upon return. FNRTN3 assumes the PC is in A, and the result is in C. FNRTN4 assumes the PC is back in D0, and the result is in C. A typical numeric function will be implemented with a call to a POP routine, calls to appropriate math routines, and a jump to an appropriate FNRTN (usually FNRTN4). If a function places its result on the stack itself (as do most string functions), EXPR is the appropriate return; this begins processing of the next token.

7.5 Implementation of Function Execution

7.5.1 Entry Point

The execution address should be marked as an entry point to allow the loader to fill in lex tables. Immediately above the entry point is the range of valid argument counts.

Above this is a string of nibbles describing each parameter. Each nibble should have the 8's bit set if a numeric parameter is allowed. The 4's bit should be set if a string parameter is allowed. The 2's bit should be set if an array parameter is required. The 1's bit is not defined but should be zero. One nibble is required for each possible parameter.

The minimum argument count (0-F) is specified first, followed by the maximum argument count (0-F).

For example:

```
        NIBHEX 8       3rd parameter numeric (if present)
        NIBHEX 8       2nd parameter numeric
        NIBHEX 4       1st parameter string
        NIBHEX 23      Argument count range (min=2,max=3)
=SUBST$ P=C    15      Load actual number of parms in P
        ?P=    2       Check if only 2 parms
        GOYES  SUBST2
```

where 2 is the minimum argument count and 3 is the maximum argument count. All XFNs have a 7 nibble tokenization, the last nibble of which is the actual number of parameters passed to the function. If the function has a variable number of parameters, the execution code for the function can find the actual number of parameters by looking at the sign field of the C register. If a function has a fixed number of parameters, it may assume that the proper number of parameters are on the stack.

Four hardware stack levels are available for function execution. A complete list of RAM that is available to and restricted from function execute is in the "Memory Structure" chapter.

7.5.2 Entry Conditions

The current program counter location is contained in D0 and has been updated past the tokens that specify the function. The 'B' field of the B register contains the table entry number to the function execution code.

The arithmetic stack expands from the end of available memory (AVMEME) toward lower memory making use of available memory. At the time of the function call, D1 points to the "top" of the stack.

If the stack grows as a result of the function call, a check should be made to prevent the stack from exceeding available memory, by comparing the stack pointer with AVMEMS. No LEEWAY need be maintained during expression execution, ie. all of available memory is truely available.

7.5.3 Exit Conditions

The program counter is stored in D0. The stack pointer is stored in D1. Other than these data pointers, the function need not preserve any CPU registers (working, scratch, or status). See the section on function returns under expression execution for information on how to resume the expression interpreter once the function's execution has completed.

7.5.4 Error Exits through MFERR/BSERR

Error exits from statements and functions require only four things:
1) S13 is set when appropriate (indicates program running)
2) PCADDR is accurate
3) The error number is loaded in C.
4) P is set appropriately to select options (set ERRN, display error prefix, etc.). See MFERR* documentation or the "Message

Handling" chapter for details.
Entry points MFERR and BSERR are used for processing errors generated in the BASIC operating system. MFERR requires that the error number is loaded in C(B); this error exit can be used for mainframe generated errors (LEX file #00). However, BSERR requires that the error number is loaded in C(3-0), specifying both the LEX ID number and the message number. It is acceptable to use BSERR for mainframe- generated errors, as long as C(3-2) is filled with zeros.

7.5.5 "Funny" Functions

The execution address of tFFN is exactly the same as tXFN. This will cause the execution address of that particular FFN to be called. One peculiar side effect of using XFN execute to get to the execution address is that the program counter (D0) will have been moved to point past the first nibble of the length byte. The first nibble of the length byte will have been read into C(S) since XFN thought it was reading a parameter count. The FFN execute should merely move D0 one nibble farther to finish skipping the length byte.

The FFN should do what it has to in order to leave exactly one item on the stack. It should not alter what was already on the stack--this is the nature of parameterless functions.

Once it has its value pushed on the stack, it should jump to EXPR, or use any of the normal entry points.

BASIC FILE CONSIDERATIONS	CHAPTER 8

When extending system capability through BASIC, there are several items to keep in mind.

8.1 ROM Generation

Before a BASIC file is RUN, the system chains together all its labels, subprograms, and user-defined functions. Also, any line number references are compiled as they are encountered in the running program. If a file in ROM is not already chained or does not have its line number references compiled, then at the time it's invoked an error will result as the system attempts to write to ROM. There are several ways to avoid this unpleasant situation.

8.1.1 Chaining a BASIC File

There are three ways to chain a file:

1) COPY it. The destination file will be chained.
2) RUN it.
3) TRANSFORM it into TEXT, and then back into BASIC.

Keep in mind that any time a file is modified, it is no longer chained.

8.1.2 Compiling Line Number References

To compile all line number references in the current file, simply execute:

```
RENUMBER 1,1,1,1
```

This statement acts as a NOP, except for the fact that it compiles line number references (No line numbers are changed).

8.2 BASIC Application Standards

8.2.1 Preserving The Main Environment

When the user runs a BASIC file to perform a given application, every effort should be made to preserve as much of the user's environment as possible. This includes variables, user flags, display format, etc. To further this end, we suggest that any BASIC application program which may destroy or use BASIC variables, should save the user environment via a CALL statement.

For example, say there is an application program, PLOT, which by necessity must use BASIC variables. If the first line of PLOT is as follows, then the user's variables will remain intact:

```
10 CALL PLOT @ SUB PLOT
```

Now the user can safely invoke PLOT by simply saying:
```
> RUN PLOT
```

8.3 BASIC Packing Techniques

With some forethought, you can use features of the HP-71 BASIC interpreter to minimize the amount of memory that your BASIC programs require. Listed below are our suggestions, along with the actual memory savings.

1) Don't use GOTO immediately after THEN or ELSE:
   Change:
   ```
   10 IF FLAG(X) THEN GOTO 100
   ```
   To:
   ```
   10 IF FLAG(X) THEN 100
   ```
   This saves three bytes.

2) Check for a null string using the LEN function:
   Change:
   ```
   10 IF A$#"" THEN .....
   ```
   To:
   ```
   20 IF LEN(A$) THEN ....
   ```
   This saves three bytes.

3) Instead of using THEN and ELSE to make one of two assignments

to a variable, do one assignment followed by a conditional to determine if the other assignment should be done:

Change:
```
10 IF X THEN K=L ELSE K=P
```
To:
```
10 K=P @ IF X THEN K=L
```
This saves three bytes.

4) Instead of testing a flag to determine if a variable (or value) should be incremented, just add the flag value:

Change:
```
10 IF FLAG(X) THEN K=K+1
```
To:
```
10 K=K+FLAG(X)
```
This saves five bytes.

Change:
```
20 IF NOT FLAG(X) THEN K=K+1
```
To:
```
20 K=K+NOT FLAG(X)
```
This also saves five bytes

5) Use single character alpha variables, instead of alpha-digit variables. There is a one byte savings for each occurence.

6) Concatenate a statement to the previous one, instead of using a new line number. There is a two byte savings for each concatenated statement.

8.4 Version Number

It is strongly recommended that each BASIC software application respond to the VER$ poll to indicate the version of the software.

This requires a LEX file to be included with no keywords, but the appropriate code to indicate the proper VER$. The last LEX ID for Custom Products - Special (244) is used as the LEX ID for VER$ response of BASIC applications. This LEX ID may be used for words by a particular custom application, without conflict.

The VER$ string should indicate the application name, using 3 or less characters, followed by a colon and a single letter. The single letter indicates the specific version number. The letter "A" is the first released version.

The following examples show VER$ strings for three HP71 BASIC applications:

VER$ String   Application Pac

----------	--------------
CIR:A	HP71 Circuit Analysis Pac - Version A
FIN:A	HP71 Finance Pac - Version A
SUR:A	HP71 Surveying Pac - Version A

The LEX file containing the VER$ should be the first file in the ROM and have a name representing the application. It is suggested that the file be protected from being copied. This can be accomplished either by designating the file as Private, or by ensuring that the file name has some lower case characters. The latter can be done by poking into the file name field of the file header:

```
10 DIM F$,N$,A$[5]  INTEGER N
20 DISP "Set name to lower case"
30 INPUT "Old filename:";F$
40 INPUT "New name:",F$;N$
50 A$=ADDR$(F$)
60 N$=(N$&"        ")[1,8]
70 FOR I=1 TO 8
80 N=NUM(N$[I])
90 POKE A$,DTH$(N)[5]&DTH$(N)[4,4]
100 A$=DTH$(HTD(A$)+2)
110 NEXT I
120 DISP "Done with name change"
```

The following examples show the names of the LEX files containing the VER$ for three BASIC application pacs:

VER$ LEX File Name	Application Pac
Circuit	HP71 Circuit Analysis
Finance	HP71 Finance
SurveyV	HP71 Surveying

UTILITIES	CHAPTER 9

This chapter provides a brief overview of some operating system entry points which are useful for external software development.

9.1 Decompile Utilities

Entry	Description
GTEXT1	Given a token, outputs the corresponding text. Includes numerous entry conditions and entry points which provides for outputting leading and/or trailing blanks.
EOLDC	Checks for statement terminators: t@, t!, tEOL.
EOLXC*	Calls EOLDC above; if statement terminator found, does not return, but handles rest of statement by going to OUTELA. If no statement terminator found, returns to caller with carry clear.
VARDC	Decompiles variables.
LIN#DC	Decompiles and outputs a line number, suppressing leading zeros.
ASCICK	Copies characters from input stream to output buffer, until encountering a character with high bit set.
EXPRDC	Decompiles expression pointed to by D1.
FILDC*	Decompiles file specifier.
ARYDC	Decompiles array which was compiled by ARRYCK.
LABLDC	Assuming D1 is at tLBLRF (label reference token), steps over tLBLRF. If label is a literal, outputs it within quotes; otherwise decompiles string expression. Returns with carry clear.
SKIPDC	Useful if an unrecognized XWORD is encountered; skips D1 to the end of the statement and goes to OUTELA (see below).

9.2 Display and Keyboard Control Utilities

9.2.1 Display Control

The LCD display and all associated HP-IL "DISPLAY IS" devices may be controlled by sending characters to the DSPCHA/DSPCHC routine. In general, these characters are processed as if they are being passed on to some external display device but the processing is actually performed by the HP-71 CPU. This includes insert mode, processing escape sequences and in general all necessary maintenence of the display buffer and status information.

The display buffer is controlled by sending characters as described above but the actual LCD is generally not affected by these characters. It is only updated when the BLDDSP routine is called. At that time the display buffer and status information is used to decide which bits of the LCD should be on. It also controls the left and right arrows that indicate whether the buffer extends past either end of the window.

9.2.1.1 Carriage Return and Line Feed

When a carriage return is sent to the display (via DSPCHA) it will cause BLDDSP to be called automatically. If the display needs to be updated to reflect the current display then BLDDSP must be called explicitly. In general calling BLDDSP doesn't take long if the LCD already reflects the display buffer since a status bit (Exact) is cleared whenever anything is done to the display buffer that might alter the LCD bit pattern that would be built. BLDDSP returns immediately if that bit indicates that the display is already built correctly.

When a carriage return is sent to the display, the cursor (and FIRSTC) should be reset to zero. What actually happens is that a flag is set so that when the next character is sent to the display these values will be reset before the character is processed. This allows the information needed to properly build and scroll the display to be preserved until it is no longer needed.

The character scroll rate is checked when a carriage return is received. If it is zero, then the first character in the display is moved so that the last character in the buffer will fit in the display. If the scroll rate is infinite, then the display is built starting at the first character in the buffer (pointed to by FIRSTC). In all other cases, the display is built starting where

FIRSTC points (usually zero) and then the character scroll delay is performed, then the FIRSTC is incremented and the display rebuilt. This is repeated until all characters in the display buffer have been viewed.

When a line feed is sent to the display, the buffer should be cleared. What actually happens is that a flag is set so that when the next character is sent to the display the buffer is cleared before then character is processed. This allows the characters in the display buffer to be scrolled through the display even though the display has technically been cleared.

The display delay is triggered whenever a line feed character is sent to the display unless the cursor is on (CurOff clear) or the delay suppress bit (XDelay) is set.

9.2.1.2 Display Escape Code Sequences

The HP-71 display accepts the following escape sequences:

Esc Q -- Insert cursor
Esc N -- Insert cursor (with wrap)
Esc R -- Replace cursor
Esc C -- Cursor right
Esc D -- Cursor left
Esc H -- Home cursor
Esc J -- Clear Display (Treated same as ESC K)
Esc K -- Delete through end of line
Esc > -- Cursor on
Esc < -- Cursor off
Esc E -- Reset display
Esc P -- Delete char
Esc O -- Delete char (with wrap)
Esc % <col> <row> -- Set cursor position absolute
Esc Ctrl-C -- Cursor far right
Esc Ctrl-D -- Cursor far left

9.2.1.3 Scrolling The Display

Once characters have been sent to the display buffer it is frequently necessary to allow the user to scroll the contents of the buffer using the cursor keys. The SCRLLR routine does this. It will watch the keyboard and cause the display to scroll whenever one of the scrolling keys is hit. It will return when the user presses a key other than a scrolling key. It will also time out after ten minutes if no key has been pressed.

9.2.1.4 Setting The Bit Pattern In The Display

The actual bit pattern in the display is normally set by BLDDSP to reflect the display buffer. However, at a lower level, the BLDBIT routine may be used to set the bit pattern according to some other buffer. This is used to implement the "VIEW" and "ERRM" keys.

9.2.2 Keyboard Interface

Keyboard scanning is performed by KEYSCN. This routine is called by the interrupt routine but may be called from anywhere. If it is called too often key bouncing may result. To prevent this, the entry point DEBNCE can be used to cause a specified wait before performing the keyscan.

When KEYSCN finds keys newly down it adds to the queue of keys in the keyboard buffer. This buffer holds up to 15 keys. If the buffer is full then the new keys are discarded.

The POPBUF routine should be used to remove keys from the buffer. This routine sets up the buffer so that repeating keys can work.

9.2.3 Summary

Entry	Description
BLDDSP	Build LCD pattern from display buffer.
BLDBIT	Build LCD from specified buffer.
DEBNCE	Debounce key before keyscan.
DSPCHA	Send character in A(B) to display buffer.
DSPCHC	Send character in C(B) to display buffer.
DSPRST	Reset display.
KEYSCN	Keyboard scanning.
POPBUF	Remove keys from buffer.

9.3 Expression Execution Utilities

9.3.1 Utilities for Pushing Items Onto Math Stack

Entry	Description
EXPEXC	The normal entry point for expression execution. Evaluates an expression by processing the tokenized stream. The value(s) are left on the stack when done.
FNRTN1	Resumes expression execution after pushing a value onto the stack. Related entry points are FNRTN2, FNRTN3, and FNRTN4. Further described in the "Statement Parse, Decompile, and Execution" chapter.
BF2STK	Adds a string to the stack from a string of characters in memory.
STKCHR	Creates a string on the stack one character at a time. It works with ADHEAD to build a proper stack item.
ADHEAD	Adds the proper string header to a string that has been placed on the the stack by STKCHR.

9.3.2 Utilities for Popping Items Off Math Stack

The following utilities are used for popping numeric or string arguments off the MATH Stack, and for checking their type.

Entry	Description
POP1N	Pops a numeric argument. If item is a string or a dope vector, a fatal error occurs. If the carry is set upon return, the argument is complex.
POP2N	Pops two numeric arguments. If either item is a string or a dope vector, a fatal error occurs. If the carry is set upon return, the arguments are complex (coerced to match each other if necessary).
MPOP1N	Similar to POP1N, but establishes the math modes, pops an argument, and tests for an exceptional value before returning. Leaves the stack pointer (D1) positioned for placing a standard floating-point number back on the stack.
MPOP2N	Similar to POP2N, but establishes the math modes, pops arguments, and tests for an exceptional values before returning. Leaves the stack pointer (D1) positioned for placing a standard floating-point number back on

	the stack.
POP1S	Tests for a string on the stack. Attempting to pop a number or dope vector with this routine results in a fatal error. Upon return, the string length is left in the CPU, with the stack pointer at the topmost (lowest address) character of the string text.
REVPOP	Has the same exit conditions as POP1S, but the string is reversed before returning.
REV$	Reverses character order of a string on the stack. Returns with the MATH Stack pointer unaltered.
POPMTH	Moves the stack pointer past one item on the stack. This item may be string, real, complex, etc.

9.4 File I/O Utilities

The following utilities are used to create files, open files, read or write arbitrary data to or from files, and to close files. For further information on file access, see the "File System" chapter in this volume and the "File Utilities" chapter in Volume II of this document.

Entry	Description
CLOSEF	Close an open file.
CRTF	Create a file of arbitrary type, in mainframe or on mass medium. Does not open file.
FIBADR	Fetches the address of an open file's FIB into register D0.
FINDF	Find a file in memory given its name and memory device type.
FSPECx	Evalute (execute) a tokenized file specification to determine the file name and device type.
MVMEM+	Expand or contract the contents of a file in memory. May be used to delete a file from the file chain.
OPENF	Open a file given its name and device type.
PRGFMF	Purge a file in memory.

PURGEF	Purge a file in memory or on mass medium.
RDBYTA	Read a byte from an opened byte-oriented file. See also WRBYTC.
READNB	Read an arbitrary number of nibbles from an opened file of any file type. See alse WRITNB.
RPLLIN	Replace, delete, or insert a line or stretch of any number of nibs in a memory file.
WRBYTC	Write a byte to an opened byte-oriented file. See also RDBYTA.
WRITNB	Write an arbitrary number of nibbles to a an opened file of any type.

9.5 Flag Utilities

Entry	Description
UPDANN	Update annunciators according to user and system flags.
SFLAGC	Clear a system flag and update annunciators.
SFLAGS	Set a system flag and update annunciators.
SFLAG?	Test a system flag.
SFLAGT	Toggle a system flag.
RNDAHX	Pop, round, convert real argument to hex integer.

9.6 Math Utilities

What follows is a brief description of some built-in HP-71 math routines that may prove useful. The routines are grouped by category.

9.6.1 Numeric Comparison

Entry	Description
TST15	Compare two 15-digit arguments.

9.6.2 Trig Routines

Entry	Description
ARG15	Compute angle of pair (x,y) of 15-digit arguments.
SIN15	Sine of a 15-Digit argument.
COS15	Cosine of a 15-Digit argument.
TAN15	Tangent of a 15-Digit argument.

9.6.3 Inverse Trig Routines

Entry	Description
ASIN15	Arcsine of a 15-digit argument.
ACOS15	Arccosine of a 15-digit argument.
ATAN15	Arctangent of a 15-digit argument.

9.6.4 Arithmetic & Square Root

Entry	Description
ADDONE	Add one (x+1) to a 15-digit argument.
SUBONE	Subtract one (x-1) from a 15-digit argument.
1/X15	Invert (1/x) a 15-digit argument.
AD2-15	Add two 15-digit arguments.
AD15S	Add two 15-digit arguments, preserving SB & XM.
SB15S	Subtract two 15-digit arguments, preserving SB & XM.
MP2-15	Multiply two 15-digit arguments.
DV2-15	Divide two 15-digit arguments.
SQR15	Square Root of a 15-digit argument.
SQRSAV	Square Root of a 15-digit argument, preserving SB & XM.

9.6.5 Integer-Fraction Functions

Entry	Description
CLRFRC	Clear the fractional part.
INFR15	Locate decimal point.

9.6.6 Logarithmic Functions

Entry	Description
LN15	Natural Logarithm (ln(x)) of a 15-digit argument.
LN1+15	ln(1+x) of a 15-digit argument (LOGP1 in HP-71 BASIC).
LGT15	Log base 10 of a 15-digit argument (LOG10 in HP-71 BASIC).

9.6.7 Exponential & Involution

Entry	Description
EXP15	e^x of a 15-digit argument (EXP(x) in HP-71 BASIC).
EX-115	[e^x - 1] of a 15-digit argument (EXPM1(x) in HP-71 BASIC).
YX2-15	Involution of a 15-digit argument (power function y^x in HP-71 BASIC).
EX15	Exponent value of a 15-digit argument (EXPONENT(2x) in HP-71 BASIC).

9.6.8 Conversion Between 15-forms and 12-forms

Entry	Description
SPLITA	Split (unpack) 12-form in A into (A,B).
SPLITC	Split (unpack) 12-form in C into (C,D).
SPLTAC	Split 12-forms in A & C into (A,B) & (C,D).
SPLTAX	Split 12-form in A, replace signaling NaN, and set XM.

uRES12	Pack 15-form math result into a 12-form, consulting rounding modes & TRAP values.
uRESD1	Variation of uRES12 preserving D1.
uRND>P	Round 15-form to p digit 15 form.

9.6.9 Pop, Test, Prepare 1 Argument

Entry	Description
ARGPR+	Read user modes, fall into ARGPRP.
ARGPRP	Pop real, detect nonfiniteness, split & normalize.
ARGSTA	Read user modes, fall into AGRST-.
ARGST-	Pop real, error for NaN, detect nonfiniteness.
POP1R	Pop real, error for complex.

9.6.10 Scratch Math Stack

Entry	Description
RCSCR	Pop 15-digit value into (C,D) from top of stack.
RCLW1	Recall 15-digit value into (A,B) from top of stack.
RCLW2	Recall 15-digit value from 1 below top of stack.
RCLW3	Recall 15-digit value from 2 below top of stack.
RCL*	Recall 15-digit value from P below top of stack.
STSCR	Push 15-digit value in A/B onto top of stack.

9.6.11 Factorial

Entry	Description
FCSTRT	Factorial for finite 15-digit nonnegative integer.

9.6.12 Statistical Utilities

Entry	Description

GETSA	Get starting address of current STAT array, test number of variables and length of array.
VARNBR	Pop 1 real argument and fall into VARNB-.
VARNB-	Convert, round to hex integer, create NaN for invalid variable number.

9.6.13 Miscellaneous Math Utilities

Entry	Description
BIASA+	Bias (or unbias) the exponent of a 15-digit argument into (A,B).
BIASC+	Bias (or Unbias) the exponent of 15-digit argument into (C,D).
CLASSA	Classify argument into one of 12 pigeonholes.
DBLSUB	Double precision fixed-point subtract: (A,C), (B,D).
DBLPI4	Create 31-digit (double precision) PI/4 in (B,D).
EX15M	Fetch exponent of a 15-digit argument.
FINITA	Test for a finite number.
FINITC	Test for a finite number.
FLIP8	Toggle status bit S8.
FLIP10	Toggle status bit S10.
FLIP11	Toggle status bit S11.
GETCON	Fetch constant from Numeric Constant Table located at TRC90.
GETVAL	Fetch constant from constant table at arbitrary address.
HNDLFL	Set exception flags.
HTRAP	Consult TRAP values.
INVNaN	Exit code for an IVL operation.
MAKE1	Create 12-dig value '1' in C and compare with B.

MESSG	Send out warning messages.
MSN15	Select most significant NaN in 2-Argumnent function.
ORGSB	Set Sticky Bit (SB) if s5=1.
ORXM	Set External Module Missing bit (XM) if s9=1.
ORSB	Set Sticky Bit (SB) if s7=1.
PI/2	Create 15-digit PI/2 in (C,D).
SAVGSB	Save Sticky Bit (SB) in s5.
SAVEXM	Save External Module Missing bit (XM) in s9, and Sticky Bit (SB) in s7.
SAVESB	Save Sticky Bit (SB) in s7.
SHFLAC	Double precision left shift (A,C).
SHFRAC	Double precision right shift (A,C).
SHFRBD	Double precision right shift (B,D).
TWO*	Double precision doubler.
XYEX	Exchange (A,B) with (C,D).

9.7 Parse Utilities

9.7.1 Parse Input Utilities

Entry	Description
GNXTCR	Skips over any blanks, returns the first non-blank character in A(B); leaves D1 at the first non-blank character. In the case where D1 already points at a non-blank character at the time GNXTCR is called, D1 is not moved.
NTOKEN	Skips over any blanks, and returns the tokenization of what follows in register A. D1 is past what was tokenized. LEXPTR contains the value of D1 (past any blanks) prior to the call.
RESPTR	Restores D1 from the value saved in LEXPTR by NTOKEN.

URDSCN Parses current input characters into a token and checks for a match with one of a given table of tokens. If a match is found, the token is output and control is passed to the corresponding address specified in the table. This is an appropriate routine to use if the presence of any number of keywords is legitimate at this point in the input stream. For example, OPTION parse, which allows only BASE, ROUND, or ANGLE as following keywords:

```
   GOSUB  URDSCN
   CON(2) =tBASE
   REL(3) =FIXP          Goto FIXP if tBASE found
   CON(2) =tANGLE
   REL(3) OPTP10         Goto OPTP10 if tANGLE found
   CON(2) =tROUND
   REL(3) OPTP20         Goto OPTP20 if tROUND found
   CON(2) 0              Terminates table
*
   GONC   OPTP30         Returns here with carry clr if
     .                   nothing in table found
     .
```

This utility should be used to guarantee a specific keyword is found by the lexical analyzer. Since URDSCN automatically restarts the lexical analyzer, this prevents a shorter keyword in another LEX file from being returned instead.

9.7.2 Parse/Decompile Output Utilities

Often it is necessary to output characters or tokens to the output buffer, or just to skip D0 (output pointer) over a certain number of nibbles while checking for sufficient memory. There are numerous utilities to do this. In addition to the entry point names given below, each utility (except OUTNIB) has additional entry points to output from register C instead of A.

Entry	Description
OUTNIB	Outputs a single nibble from the low nib of C.
OUT1TK	Outputs a byte from A(B). Alternate entry point OUTBYT outputs a byte from C(B).
OUT2TK	Outputs two bytes from the lower 4 nibbles of A. Alternate entry point OUT2TC outputs from C.
OUT3TK	Outputs three bytes from the lower 6 nibbles of A. Alternate entry point OUT3TC outputs from C.

OUTNBS	Outputs n nibbles from the lower n nibbles of A. P must be set to n-1. Alternate entry point OUTNBC outputs from C.

9.7.3 Parse General Utilities

Entry	Description
FSPECp	Parses and outputs valid file specifier.
FILEP	Parses valid file name. If it is a string expression, then it is tokenized and written to output buffer. If it is a literal, the file name is returned in A with C(S) set for WP (word through pointer) write of the file name characters.
EXPPAR	Parses expression; returns information on whether expression was valid and whether it was string or numeric. If it was valid, calls NTOKEN on whatever followed the expression and returns.
NUMCK	Parses valid numeric expression; has numerous entry points.
STRGCK	Parses valid string expression.
CATCHR	Categorizes character in A(B) (or character pointed to by D1) as (a) digit, (b) letter, (c) special character [*,+,-,.,/, ,], or (d) other.
CNVWUC	Converts next 8 characters in input buffer to uppercase. There are multiple entry points, including one to skip over preceding blanks.
COMCK	Sees if next token is tCOMMA.
LBLINP	Parses line number or label.
EOLCK	Checks for statement terminator: t@, t!, tEOL.
ARRYCK	Verifies array subscripts; allows one or two subscripts. Number of subscripts returned in B(A).
SPLVRP	Parses and outputs simple variable, or error exits.
NXTP	Parses and outputs simple numeric variable, or error exits.
OUTVAR	Given a variable token in A, outputs the variable.

9.8 Statement Execution Utilities

Entry	Description
FSPECx	Evaluates file specifiers; will POLL for any not recognized by mainframe.
FILXQ^	Evaluates mainframe file specifiers and dedicated device specifiers; devices currently accepted are PORT, MAIN, CARD, PCRD.
EXPEXC	Evaluates expression pointed to by D0. Upon exit, evaluated expression is on the stack. See EXPEXC documentation for details.
FINDF	Given a file specifier returned from FSPECx or FILXQ^, searches for the given file. Indicates upon exit, whether or not file found. If file found, provides information on where. Numerous entry points.
EOLXCK	Given a token in A(B), returns with carry set if it is a statement terminator: tEOL, t@, t!, tELSE.

9.8.1 Utilities for PRINT class statements

PRINT and DISP statements are very similar. The mainframe may be extended to allow other statements of the same class, such as OUTPUT. What these statements have in common is that they take an expression list and output ASCII strings to a device. The way the system works is that a nibble of RAM (STMTR0(0)) is set to a digit that identifies the type of the statement. This nibble is used to determine the current information on how to output to the proper device.

The CKINFO routine looks at this nibble and sets up in statement scratch RAM all the information required. For DISP and PRINT this information includes the address of a handler routine for that device, a pointer to the relevant position/width counters, and the endline string. Other parts of statement scratch may be used to hold other information necessary for the handler. The handler is a routine that is capable of sending a block of characters to the output device. Immediately above the handler code is a 5 nibble relative offset to a routine that should be called once the entire statement has been finished--this allows for necessary cleanup, ect.

Thus, execution of statements of the PRINT class is divided into three parts:

PART1: Set STMTR0(0) to the statment type and call CKINFO to set up for parts 2 and 3.

PART2: This is the handler that knows how to send a block of characters to a device.

PART3: This is the clean up routine that is called at the end of the entire statement.

The at STMTR0 nibble is preserved throughout the execution of the statement. Even if the user changes the PRINTER IS assignment in the middle of a PRINT statement (via a multi-line user-defined function) this nibble will still say that it is a PRINT statement. If a multi-line user-defined function is referenced within an expression to be output, CKINFO will recalculate all the information pertinent to the current statement. This insures that the output always get sent to the right place in the right format.

To implement a new statement of the PRINT class, it is necessary to be allocated a unique statement type to be filled in STMTR0(0). The CKINFO routine polls (pPRTCL) to find a routine to fill in statement scratch area with the appropriate information. This poll must be handled. The PRINT statement causes a different poll (pPRTIS) which determines the PRINTER IS device if any.

9.9 System Buffer Utilities

Entry	Description
I/OFND	Given a buffer ID, returns pointer to that buffer.
I/OALL	Given a buffer ID and desired buffer length, either expands or contracts existing buffer or creates buffer of the specified length and ID.
I/OEXP	Expands buffer by a specified number of nibbles.
I/OCON	Shrinks buffer by a specified number of nibbles.
I/OCOL	Shrinks buffer to length zero.
I/ODAL	Deletes (deallocates) specified buffer.
I/ORES	Sets high bit of buffer ID to preserve buffer during pCONF.
IOFSCR	Finds available scratch buffer ID.

9.10 Variable Storage Utilities

To process an assignment statement, expression execute (routine EXPEXC) is called to evaluate the destination variable to the left of the equal sign. If the destination is legal, certain information must be saved away such that it can be retrieved after expression execute has been called to evaluate the expression to the right of the equal sign. The utility DEST saves this information away in Statement Scratch RAM so that when EXPEXC is called it will be preserved and updated if memory moves (see description of DEST below).

Following expression execute, the B register looks like:

```
    +-+----+----+---------+-------+
B:  |t|2nd Index|1st Index|Address|
    +-+----+----+---------+-------+
```

t = type (= C minus actual type)

- 2 -- Integer
- 1 -- Short
- 0 -- Real
- F -- Complex short
- E -- Complex
- D -- String
- 8 -- Nonexistent numeric array
- C -- Nonexistent string array

2nd Index = Second index of substring function (string only)

1st Index = First index of substring function (string only)

Address = Variable address if variable exists (high nibble will be nonzero).
= Variable name if variable does not exist (3-digit format with 00 in nibbles 3 and 4).
= 00000 if an out-of-bound array element has been specified.

Other destination information resides in function scratch following expression execute. F-R1-0 contains the element number computed by the array reference. This is used by TRACE. F-R1-3 contains the subscript count used in a reference to a nonexistent array. This is used when an implicit array declaration is recognized.

Since strings may be stored directly into substrings, the stack header for the actual strings must sometimes carry destination information. The substring function maintains the destination information kept in the B-register and in function scratch with this stack header information. See the section on "Data Types" in the "Internal Data Representation" chapter.

9.10.1 Summary

Entry	Description
DEST	Stores destination variable information in the following areas of Statement Scratch RAM: S-R0-0 = Variable address or name S-R0-1 = First substring parameter S-R0-2 = Second substring parameter S-R0-3 = Variable type S-R1-0 = Array element number S-R1-1 = Maximum string length S-R1-3 = Subscript count
STORE	Takes information placed in statement scratch and uses it to store a value from the top of the math stack into a variable. It will create the variable if necessary.
ADDRSS	Very low level routine that scans a variable chain to find the address of a variable. Alternative entry point is ADRS40.

```
+-----------------------------------------+----------------+
|                                         |                |
|   MESSAGE HANDLING                      |   CHAPTER  10  |
|                                         |                |
+-----------------------------------------+----------------+
```

This chapter describes, in five sections:

1) BASIC keywords involving messages

2) Details on using the message handling routine to generate errors, warnings or system messages.

3) Insufficient Memory Error.

4) Conventions for Foreign Language message translation.

5) Construction of message tables, as found in LEX files.

Except for two subsections ("BASIC Keywords Involving Messages", and "BASIC Error Trapping", below), the discussion in this chapter is from the viewpoint of assembly language. The options discussed are ones an assembly language routine may select when calling the message handling routines. Subsection "BASIC error trapping" discusses error trapping at the BASIC language level.

10.1 BASIC Keywords Involving Messages

10.1.1 ERRN

The function ERRN returns the number of the last error or warning detected by the computer. Assembly language routines which call the message handler determine if ERRN is set or not.

10.1.2 ERRL

The function ERRL returns the number of the last line in which an error or warning occurred; if it occurred in a non-BASIC program, ERRL returns zero. Assembly language routines which call the message handler determine if ERRL is set or not.

10.1.3 ERRM$

The function ERRM$ returns the last error or warning message, as a string. ERRM$ is derived from the value of ERRN.

If ERRN is an error number from a LEX file, and that LEX file is removed from the computer, ERRM$ will return the null string (until ERRN is again changed); this is because the ERRM$ searches the LEX file message table for the message.

The keystroke [g][ERRM] displays the last error or warning message as long as a key is held down. The message is built in the same manner as ERRM$.

10.1.4 MSG$ Function

The BASIC keyword MSG$ has been implemented in LEX file #82 of the User's Library. Its usage is similar to ERRM$, except that it accepts an argument (a decimal message number). E.g., MSG$(255131) returns message number 131 from LEX file 255. Its purpose is twofold:

1) Whereas ERRM$ returns the last error or warning message, MSG$ returns any standard message from any message table.

2) Through the use of the pTRANS poll ("translate"), it substitutes a foreign language translation of the desired message, if a language translator LEX file is present in the computer.

MSG$ allows a BASIC user to build custom messages from any message tables. In addition, the translation capability provides a powerful tool for BASIC application packs which accept commands in any language. An excellent example is the HP-71 Text Editor, a BASIC program which stores all its commands and responses, along with its help catalog, in a message table. User input (commands and responses) are compared to entries in the message table, using MSG$, allowing a language translator LEX file to drive the Text Editor in any language.

MSG$ uses the message building utility TBMSG$ in the message handler. When constructing message tables, take into consideration the use of MSG$ to display each message. More details are provided in the sections "Foreign Language Translators", and "Message Table Construction".

10.2 Message Handling

The message handling routine displays any standard message, including errors, warnings and system messages. Standard messages are found in tables and identified by a four digit hex number -- a two-digit LEX ID and a two-digit message ID number. In this chapter, the term "message number" usually refers to the complete four-digit constant; "message ID number" refers to the two-digit constant identifying the message within the LEX table.

The mainframe contains one message table. Each external LEX file may contain an associated message table.

The message handler is designed as a utility for any application, whether a LEX file used to extend the BASIC library, or a take-over subsystem (such as FORTH) with a distinct message style.

In its most powerful form, the message handling routine can be used as an error or warning utility, performing certain housekeeping functions such as:

- -- updating ERRN and ERRL
- -- checking if ON ERROR is in effect (errors only)
- -- sounding a beep
- -- re-displaying a parse error with the cursor positioned at the error

In its simplest form, the message handling routine can be used to build any message from "building block" words. These building blocks can be found in any LEX file, including the mainframe (LEX #00), the local LEX file, or a different LEX file entirely. Through the use of these building blocks, a message may be made to look like an error or warning, even if not treated that way by the message routines. See the section entitled "Foreign Language Translators", for details.

10.2.1 Message Types

The message handler allows several options, including message type, text insertion, storage of ERRN and ERRL, display delay, checking ON ERROR, and beep.

The four message types:

1) an error message

2) a memory error ("Insufficient Memory")
3) a warning message
or 4) a system message (text only).

The calling routine determines the message type by selecting the proper entry point and entry conditions into the handling routine. The calling routine is responsible for distinguishing between errors and warnings, such as in the case of DEFAULT OFF.

The distinguishing features of each type are as follows. Entry points are discussed in the next subsection.

10.2.1.1 Effects of Error Messages

Handling the message as an error has these effects (in this order):

1. Sends out a pERROR poll
2. If eMEM message, process Memory Error
3. Sets ERRN and ERRL if option selected
4. If ON ERROR in effect, branch to ONERR
5. Displays prefix "ERR:" if option selected
6. Disallows text insertion when sending message (*)
7. Sounds beep
8. If parse error, re-displays input line

Because of steps 4, 6 and 7, selecting the error message type is most useful for BASIC operating system errors. That is, a system such as FORTH may want to avoid those effects.

*Note: a special entry point allows text insertion in error messages, if necessary. See subsection "Entry point MFERsp", below.

10.2.1.2 Effects of Memory Error Messages

Memory error messages are a subset of error messages, but because of their insidious nature (i.e., a MEMERR can occur during any low-level utility), they have separate processing:

1. Sends out pMEM poll
2. Recovers available memory (at least LEEWAY)
3. Sets ERRN and ERRL if option selected
4. If ON ERROR in effect, branch to ONERR
5. Displays prefix "ERR:" if option selected
6. Disallows text insertion when sending message
7. Sounds beep

See section entitled "Insufficient Memory Error" for more details on memory errors.

10.2.1.3 Effects of Warning Messages

Handling messages as warnings has these effects (in this order):

1. Sends out pWARN poll
2. Checks Quiet (flag -1), if selected, exits if set
3. If eMEM message, process Memory Error
4. Sets ERRN and ERRL if option selected
5. Does NOT check ON ERROR!
6. Displays prefix "WRN:" if option selected
7. Displays msg, with text insertions if appropriate
8. Observes display delay, if option selected
9. Sounds beep, if option selected

10.2.1.4 Effects of System Messages

The term "system message" refers to any message which is displayed without an "ERR:" or "WRN:" prefix, and doesn't branch to ON ERROR. The system message facility allows building and displaying messages for the user's information without invoking the housekeeping functions of the error routines. A system message may be built to look like an error or warning, if desired. To display a system message, the message handling routine is used as if a warning were being displayed, with the appropriate options selected:

1. Sends out pWARN poll
2. Checks Quiet (flag -1), if selected, exits if set
3. If eMEM message, processes Memory Error
4. Sets ERRN and ERRL if option selected
5. Does NOT check ON ERROR!
6. Does NOT display "WRN:" prefix (by definition)
7. Displays msg, with text insertions if appropriate
8. Observes display delay, if option selected
9. Sounds beep, if option selected

10.2.1.5 Text Insertion

One option that warnings and system messages have is to insert text at certain points in certain messages. Normally, this option is not allowed for error messages, as explained in subsection "ERRN and ERRL Considerations".

Text insertion points are fixed; only certain messages allow them, and these are known by the calling routine. That is, you cannot insert text except at specific points in known messages. See section "Message Table Construction" for details on constructing a message to allow text insertion.

Text insertions are in the form of digits or ASCII characters, allowing dynamic message building. Consider the case of mainframe message number 88, used by TRANSFORM execution. The message in the table looks like:

```
TFM URN L{6}:{5}
     where {6} and {5} indicate two types of insertion points:
          {6} specifies digits or ASCII as passed by the
               calling routine (with no trailing space).
          {5} specifies insertion of an entire message from
               a LEX table, whose number is passed by the
               calling routine.
```

When the TRANSFORM execution routine calls the message handler to display this message, it might pass, say, line number 145 for the first insertion, and message number 0051 (LEX ID #00, message number 81 in decimal) for the second insertion. When displayed, the message would look like this:

```
TFM URN L145:Invalid Parm
```

10.2.1.6 ERRN and ERRL Considerations

Selecting to update ERRN will simultaneously cause ERRL to be updated, if indeed a program is running. This action is determined by S13 (CPU status bit 13): S13=1 implies a running program.

In addition, updating ERRN has an effect on two other functions: ERRM$ and the [g][ERRM] keystroke. Both are constructed from the value stored in ERRN (RAM location ERR#, hex address 27FE4.)

Any message which specifies text insertion will be reconstructed for ERRM$ and [g][ERRM] without text in that position! (It is infeasible to store the inserted text for later recall of ERRM$ or [g][ERRM].) For this reason, normal processing of error messages never allows text insertion; the restriction requires error messages to be succinctly contained in the tables (*). When deciding whether to select the ERRN storage option, consider the effects of missing text insertions.

*Note: calling the message routines at entry point MFERsp allows an error message to employ text insertion, if necessary. See subsection "Entry point MFERsp", below.

10.2.1.7 Messages During Running Programs

Any use of the message handling routines -- whether within the BASIC operating system or not -- must consider the effects of S13 (CPU status bit 13). S13=1 implies a running program, and will have the following effects:

For errors (including MEMERR):
- -- CURRL and PCADDR will be updated (if the running program is not BASIC, CURRL is set to zero).
- -- If ERRN update is selected, ERRL will be updated.
- -- ON ERROR will be checked.
- -- If "ERR:" prefix is selected, "ERR L<#>:" will be displayed, with the line number (if the running program is not BASIC, "ERR~~" is displayed).
- -- The execution pointer (D0) is left at a @ token, or at the line number.

For warnings and system messages:
- -- CURRL and PCADDR will be updated (if the running program is not BASIC, CURRL is set to zero).
- -- If ERRN update is selected, ERRL will be updated.
- -- If "WRN:" prefix is selected, "WRN L<#>:" will be displayed, with the line number (if the running program is not BASIC, "WRN~~" is displayed).

10.2.2 Error Message Handling

The main processing routine for error messages is MFERR*. Any message processed by this entry point will sound a beep.

10.2.2.1 Entry Points

MFERR* -- This is the main error handler, a subroutine which processes errors, then returns. MFERR* requires entry with the entire message number (LEX ID and message ID) specified. MFERR* is the preferred routine to use for a non-BASIC system, say, which wants to regain control after the message is displayed. MFERR* should be called as a subroutine.

MFERR -- ("Mainframe Error") always sets LEX ID=00, specifying a message in the mainframe table. Exits to BASIC main loop. MFERR should be called with a GOVLNG (not a subroutine).

BSERR -- ("BASIC System Error") allows entry with the LEX ID of the message number specified. BSERR can be called for a mainframe error, of course, if LEX ID=00 is specified. This entry point is used to process most BASIC errors, since it always exits to the BASIC main loop. A non- BASIC system, of course, might want to use this entry point if it doesn't care that processing exits to the main loop (CALC mode, for instance, allows errors to go through BSERR, but picks up processing through a branch at the main loop). BSERR should be called with a GOVLNG (not a subroutine).

MEMER* -- This is the main Memory Error handler, a subroutine which processes Memory Errors, then returns. MEMER* requires entry with the entire message number (LEX ID and message ID) specified. Normally eMEM (0018hex) is used, but a Memory Error message from any LEX file can be specified. MEMER* is the preferred routined to use for a non-BASIC system, say which wants to regain control after the Memory Error is displayed. MEMER* should be called as a subroutine.

MEMERR -- ("Memory Error") sets P=0 which selects certain options as explained below, then falls into MEMERX. MEMERR should be called with a GOVLNG (not a subroutine).

MEMERX -- Allows any value of P (which determines which options are selected), selects the mainframe message "Insufficient Memory" (number 0018hex), processes the error and exits to the BASIC main loop. MEMERX should be called with a GOVLNG (not a subroutine).

10.2.2.2 Entry Conditions for MFERR*

To display standard error messages, call the message handler (MFERR*, MFERR, BSERR or MEMER*) with:

```
 (1)-------
  | P set as follows:
  |  P= 1xxx  "This is a Parse error" (i.e., re-display
  |                input line w/cursor backup)
  |                NOT ALLOWED for a Memory Error!
```

```
|
|  P= x1xx  Do not store error number (Else store ERRN)
|
|  P= xx1x  Display message only
|                (Else display "ERR:" & ERRL)
|
|  Bit0 of P not used at present. (**)
|


(2)-------
|  C(3-2) = LEX ID# (Hex) in whose table the message
|               is found  (LEX ID#= 00 for mainframe)
|
|  C(B)   = Message ID number (Hex)


(3)-------
|  If parse error, then
|     Input pointer (D1) points to character in
|        input buffer where error occurred.
|     INBS points to beginning of input buffer.
|     A(A)= addr prompt string for input re-display;
|         = 0 if BASIC prompt string desired.
|  Else D1, INBS, and A(A) not used.
|
|
```

(**) Bit0 of the P register is reserved for future applications, as a way for the LEX file which generated the error to communicate with other LEX files. The meaning of this bit is not yet decided. In the meantime, bit0 must=0.

10.2.2.3 Parse Errors

As described above, a parse error is identified by setting bit3 in the P register before calling MFERR*. However, several entry points already exist for specific parse errors. They all set the necessary registers for entry into the message handler, report the error, re-display the line and exit to the BASIC main loop. See the chapter entitled "Statement Parse, Decompile and Execution", under the heading "Writing a Parse Routine -- Parse Errors", for details on these entry points.

10.2.2.4 Examples

```
                                                  P=
                                                -------
   Normal BASIC execution error                     0
     (Store ERRN & ERRL; display "ERR L<#>:")

   Normal BASIC Parse error                         8
     (Re-display input line, store ERRN,
      display "ERR:")
                          ------ A(A)=0
                          ------ D1=error location within
                                    input buffer


   External system (Text Editor, FORTH             14
   interpreter, etc.) Parse error
     (Don't store ERRN; display message text
      only; use given prompt string)
                          ------ A(A)= prompt string address
                          ------ D1=error location within
                                    input buffer
```

10.2.2.5 Entry Point MFERsp

In spite of the inability of text insertion to be reconstructed for ERRM$, it has been determined that several applications desire to display error messages with text insertion. Calling a special entry point in the MFERR* routine will allow this. This entry point, MFERsp, occurs after the pERROR poll of MFERR*, so some processing must be performed before calling MFERsp. This routine, like MFERR*, is a subroutine; processing does not jump to the BASIC main loop.

Entry conditions for messages using text insertion are given below, in condition (3) under "Entry Conditions for MFWRN". Instead of the P register being used for options, C(S) is used. Otherwise, entry conditions for MFERsp are as specified for MFERR*.

Calling MFERsp must be done in the following manner:

```
<set R2 according to text insertion options>
<set C(14-13) according to text insert options>
<set C(S) bits according to MFERR* options>
<set C(3-0)=message number>
R0=C                        Store options, msg# in R0
SETHEX
GOSBVL =POLL                pERROR poll.
CON(2) =pERROR
CPEX   15                   In case poll error, options.
```

```
        P=     12           P value for "error".
        LCHEX  00F          In case poll error...
        GOC    LABEL1       CRY=poll error.
        ?XM=0               Poll handled?
        GOYES  LABEL3       Yes! Abort message.
        C=R0
        LCHEX  F            C(12)=F for "error" flag.
 LABEL1 GOSBVL =MFERsp
 LABEL3 P=     0            (if necessary from ?XM=0
        .....                  jump, above....)
```

10.2.3 Warning Message Handling

The entry points for Warnings are MFWRN or MFWRNQ.

Most warnings are to be suppressed if the Quiet option (flag -1) is set. The entry point for these messages is MFWRNQ ; entry conditions are the same as for MFWRN, but a check is first performed on the Quiet option (the Quiet check is performed after pWARN poll). If Quiet is set, processing returns to the calling routine immediately.

The two warning handler entry points are always called as subroutines; warnings, since they do not halt processing, return to the calling routine.

The warning handler provides much the same options as the Error handler. Two notable exceptions are these:
-- warnings never branch to ON ERROR
-- warnings allow text insertion in designated messages.

10.2.3.1 Entry Conditions for MFWRN

To display standard messages, call the MFWRN (or MFWRNQ) routine with:

```
 (1)-------
  | P set as follows:
  |   P= 1xxx  Sound Beep.
  |
  |   P= x1xx  Do not store warning number (Else store ERRN)
  |
  |   P= xx1x  Display message only
  |                 (Else display "WRN:" & ERRL)
  |
  |   P= xxx1  Display message without observing DELAY.
  |                 (See "MFWRN DELAY Option", below)
  |
```

```
(2)-------
 | C(3-2) = LEX ID# (Hex) in whose table the message
 |             is found  (LEX ID#= 00 for mainframe)
 |
 | C(B)   = Message ID number (Hex)


(3)-------
 | If desired message has text insertion points:
 |  R2 register: source of text insertion.
 |  C(14):  type of insertion.
 |  C(13):  how many characters in insertion.
 |
 |   R2
 |   -----
 |    = actual output characters if C(14)= 1xxx
 |    = address of output characters if C(14)= 0xxx
 |    = additionally, if C(14)= 0000, upper byte
 |        of R2 contains control nibbles.
 |
 |   C(14)
 |   -----
 |    1xxx   use contents of R2 register as output
 |    0xxx   use address in R2 register to find output
 |
 |    x000   Output is already in ASCII form
 |
 |        Digit output (digits can be Hex or Dec):
 |    x001   Digit output-- replace leading 0's with blanks
 |    x010   Digit output-- don't suppress leading 0's
 |    x011   Digit output-- suppress leading 0's
 |
 |
 |        Hex-to-Dec conversions always generate
 |          decimal numbers with 7 digits:
 |    x100   Hex-to-Dec: suppress up to 3 leading 0's
 |    x101   Hex-to-Dec: suppress up to 4 leading 0's
 |    x110   Hex-to-Dec: suppress up to 5 leading 0's
 |    x111   Hex-to-Dec: suppress up to 6 leading 0's
 |
 |
 |   C(13)
 |   -----
 |     For C(14)= 1000 ("ASCII output is in R2")
 |         C(13)=  nibbles-1 to be output. Hence the
 |                 nibs MUST be even!!; C(13) odd. E.g.,
 |                if 5 chars for output, C(13)=9.
 |
 |     For C(14)= x0xx (hex or dec digit output)
```

```
|           C(13)=  digits-1 to be output, hence
|                   no more than 16.
|
|       For C(14)= x1xx (hex-to-dec conversion)
|           C(13)=  digits-1 in number to be converted
|                   Max hex value for conversion is FFFFF
|                   (1048575 dec), hence C(13) must be 4
|                   or less.
|
|       For C(14)= 0000 ("ASCII output from DAT1")
|           C(13)= 0: no output
|                  1: Send out specified number of
|                     character; R2(15-14)= chars-1.
|                  2: Send out chars until ASCII termin-
|                     ator is found.  ASCII terminator
|                     is passed in R2(15-14) (usually
|                     an FF terminator, but any byte
|                     value can be used).
|
```

10.2.3.2 MFWRN DELAY Option

Warning messages (and system message, which use the same entry point) have the option of observing DELAY. Most warnings observe DELAY, so that the message remains in the display for the user-specified delay time before execution resumes.

Selecting to observe DELAY means that the HP-71 will leave the message in the display until 1) the DELAY time expires, or 2) a key is pressed, whichever occurs first. Program execution halts (processing remains in a display utility which counts down the delay time), although this is transparent to the user; program execution resumes when the delay time expires.

Selecting to not observe DELAY means that the HP-71 continues execution immediately; the assembly language routines have control over how long the message remains in the display. For instance, the card reader system messages (such as "Pull Card") do not observe the DELAY setting. The card reader routine continues processing immediately; if the user starts pulling the card, the card reader routine will be able to detect it.

10.2.3.3 Multiple Text Insertions

Zero, one or two text insertions in any one message (including its building blocks) are allowed. If a message calls for zero insertions, R2 is not used by the message building routines. If one text insertion is used, as much of R2 as desired can be used to pass the number, characters or address (as appropriate); upper C indicates how much of R2 to use for the insertion.

When two text insertions are used in a message, the following must be observed:

- -- The two text insertions must be of the same type (i.e., the codes in C(14-13) are used for both).
- -- R2(A) must contain the entire number, characters or address (as appropriate) for the first insertion.
- -- R2(9-5) must contain the entire number, characters or address (as appropriate) for the second insertion.

10.2.3.4 Indirect Message Calling

A special type of text insertion is that of an entire message. This is different from a building block in that the calling routine passes the message number (four digit hex, including LEX ID and message ID) in R2, as it would pass any other text insertion. However, whereas other types of insertions allow the option of using R2 to point to the insertion, R2 must contain the NUMBER of the desired message (in R2(A) or R2(9-5), or both, as appropriate), not a pointer to the number.

For an indirect message call, C(14-13) must be nonzero. The value in these two nibbles is unimportant, unless a second text insertion requires a meaningful nonzero value; in this case, using that value is sufficient (see entry conditions, above).

10.2.4 System Messages

The term "system message" refers to any message which selects the following options:

1) displays message text only (no "WRN:" or "ERR:" prefix)
2) does not branch to ON ERROR.

This implies that a system message must enter through the MFWRN or MFWRNQ entry points (depending on whether it wants to observe the Quiet option, flag -1). In addition, a system message may elect to store ERRN (and ERRL), to sound the beeper, or to display the message without delay setting.

10.2.4.1 Entry Conditions for System Messages

Entering MFWRN or MFWRNQ with P set to the appropriate value will display system messages. For example:

```
P= 1110 (=14)
   1    Beep.
```

```
      1   Do not store msg number as ERRN (or ERRL, either).
       1  Display message text only.
        0 Observe display delay.
or,
 P= 0111 (=7)
    0     No beep.
      1   Do not store msg number as ERRN (or ERRL, either).
       1  Display message text only.
        1 Do not observe display delay.
```

The options and codes regarding text insertion are as specified above in "Warning Message Handling". Processing returns to the calling routine after system messages are displayed (MFWRN and MFWRNQ are called as subroutines).

10.2.4.2 Adding Prefixes to System Messages

System messages can be made to look like errors or warnings by displaying the appropriate prefix ("ERR:" or "WRN:") as part of the message.

For example, message number 88 in the mainframe looks like this in the table:

    TFM WRN L{6}:{5}

where {6} and {5} indicate two types of insertion points (see subsection "Text insertion", above). The message is displayed by TRANSFORM execution by calling MFWRNQ with the option to suppress the standard warning prefix, "WRN:". Thus, the message itself contains the WRN prefix, and looks similar to other warnings.

The same thing can be done by a subsystem which wants to generate its own error prefix -- "Error:", for example -- for the messages in its table. Each message in the table might include this "prefix" as part of its text. Then, by displaying them as system messages, they will look like other errors. (Multiple occurrences of this "prefix" can be handled efficiently by building blocks. See section "Message Table Construction" for details on building blocks.)

A foreign language message translator could use this feature to substitute a foreign prefix when intercepting the pMEM, pERROR or pWARN polls. For instance, a Spanish translator might suppress the standard "WRN:" prefix, and include as part of each warning the prefix "CDO:" (for 'cuidado'). Again, a building block in each message would make this easy.

Be aware that the ideas presented here are feasible with the message handler options. However, there are some problems to be overcome by the poll handlers which make implementation slightly more difficult than it may seem. Namely:

1) The new prefix should have the option of including a line number for a running program. E.g., using the example from above,

```
CDO:            for a keyboard warning
CDO L<#>:       for a warning from a running program
```

This could be effected by the poll handler which builds the appropriate text for a type (6) insertion before calling MFWRN.

2) If making a system message look like an error, remember that ON ERROR is not checked for system messages. In this case, ON ERROR should be checked locally in the poll handler, with a subroutine as follows:

```
DO=(5) =ERRSUB     Check if error in ON ERROR GOSUB...
C=DAT0 A
?C#0   A           Error in ON ERROR GOSUB... ?
RTNYES             Yes. Report error.
DO=DO+ 5           Check if ON ERROR in effect.
C=DAT0 A
?C=0   A           ON ERROR in effect?
RTNYES             No. Report error.
RTNCC              Yes.  Don't report error.
```

If ON ERROR is in effect, it would probably be best to call MFERR* and let it jump to ON ERROR, since it also sets ERRN and does other housekeeping.

10.3 Insufficient Memory Error

NOTE: The message handling routine checks ALL messages (error, warning and system) for the eMEM constant (value 0018 hex, or 24 decimal). If the message number is eMEM, an "Insufficient Memory" error is automatically generated. This is explained below. If for any reason an assembly language routine wants to generate "Insufficient Memory" as a NON-error message, it must be set up as a separate message in a LEX file.

10.3.1 Reporting MEMERR

An "Insufficient Memory" error can be generated during execution of any routine which uses available memory, which is to say, during execution of almost any statement or command. Any routine which uses available memory (either claiming it for "permanent" storage or for use as a temporary buffer) must ensure that AVMEMS (Available Memory Start) and AVMEME (Available Memory End) are not exceeded. In addition, a routine which claims "permanent" memory MUST insure that the LEEWAY (available memory safety factor) is not violated. For rules involving correct memory management, see the "Memory Structure" chapter, under the section "Available Memory Management."

If for some reason LEEWAY has been violated (permanent memory allocation has left less than LEEWAY available), the computer will enter an infinite loop when it finds there is not enough room to build the "Insufficient Memory" message.

If RAM usage reaches AVMEMS or AVMEME, an "Insufficient Memory" error should be generated in one of two ways:

1) Jump directly to MEMERR (BASIC system), or the subroutine MEMER* (non-BASIC system).

2) If found in a low-level utility, the convention is to return with carry set for ANY error, with C(3-0)=error number. The calling routine is responsible for checking carry and jumping to BSERR for any error.

A Memory Error is an insidious condition; it can crop up at the point, say, when a routine is trying to report a less severe error. In fact, the message routine itself requires available memory to build a message, which might easily cause any message to be converted into a Memory Error. Some of the problems which require special handling for MEMERR are:

- -- Some low level routines exit with carry set to indicate an error, with C(3-0)= error number. A MEMERR is treated like any other error in these instances, and it might pass right through to the standard error entry point BSERR. At this point, the message handler must intercept all MEMERRs, to make sure they are handled properly.
- -- A MEMERR may occur several levels deep in a low-level utility; returning to the caller may be infeasible because execution was not completed (this is what would happen if the message handler ran out of building space).
- -- A LEX file may need to be alerted immediately that an operation failed, so that it can recover without corrupting memory (such as encountering MEMERR halfway through a file manipulation).

A Memory Error should never be generated while handling a slow poll, if the poll is intended to continue. Since available memory is recovered, the crucial poll storage area is lost. Calling the entry point MEMERR is permissible, since it exits to the BASIC main loop (thereby aborting the poll).

As a precaution to fast poll handlers, generating a Memory Error may exceed the subroutine stack limit, since a pMEM poll is issued. Therefore, it is inadvisable to generate a Memory Error while handling a fast poll, if the poll is intended to continue.

10.3.1.1 Calling MEMER*

MEMER* is a subroutine which processes Memory Errors. It requires the calling routine to load a message constant into C(3-0). Normally eMEM (0018 hex) is used, but a message constant from any LEX file can be used. This would allow a subsystem to report, say, "Out of Scratch Area", process the Memory Error in the standard manner, then recover control after the message.

For LEX files operating within the BASIC system (including foreign language files), a Memory Error could be generated by calling MEMER* with the desired message constant. But the preferred way is to call MEMERR (i.e., use the mainframe eMEM constant), intercept the pMEM poll and substitute the alternate message constant at that time.

A subsystem which generates its own Memory Error message may desire to construct one with text insertion, (such as "Write Limit:<filename>"). The appropriate way to do this is to set up the text insertion in R2, call MEMER* (a subroutine) with the appropriate message number, and adjust C(14-13) as text insertion controls during the pMEM poll handling. Text insertion is described completely under "Warning Message Handling."

See subsection "Error Message Handling" for details on calling MEMER*.

10.3.2 MEMERR Handling

A Memory Error ("Insufficient Memory") allows the same options as any other error (store ERRN & ERRL, display message text only). However, a Memory Error should never be called as a parse error. For details of these options when calling MEMERR (or MEMER*) see subsections "Message Handling Options", and "Error Message Handling", above.

To prevent the message handler from running out of memory (a building area for the message) during a MEMERR and thus causing an infinite loop, available memory is first recovered, using routines COLLAP and CLCOLL. COLLAP sets the pointer in AVMEME to the value of the pointer in FORSTK (recovers AvMemEnd), and CLCOLL sets the pointers in AVMEMS, OUTBS and SYSEN to the value of the pointer in CLCSTK (recovers AvMemSt).

This frees an area of memory at least as large as LEEWAY (212 nibbles). Correct memory management is imperative (as it always is) because at this point if LEEWAY is not available, someone has really screwed up! Guaranteeing an area of RAM at least as large as LEEWAY means that any "Insufficient Memory" message (whether re-worded or translated into a foreign language) cannot exceed 106 characters (including prefix), or about 80 characters (excluding a long prefix). However, no one should ever consider any message longer than 30 characters anyway.

10.3.2.1 MEMERR Poll

A separate poll is sent out when a Memory Error is encountered. Be aware that if a Memory Error enters through BSERR (that is, a routine calls BSERR with eMEM constant), two polls will be issued -- one for pERROR, and then when the eMEM constant is intercepted, another one for pMEM. The same would happen if eMEM were issued as a warning -- first pWARN, then pMEM.

The main purposes of the pMEM poll are:

- -- To allow the poll handler to substitute another message constant for eMEM. If this is done, the message will still be handled as a memory error.
- -- To allow the poll handler to load its own return address to capture processing after the memory error is reported. (For instance, if the FORTH system calls a mainframe utility which in turn generates a MEMERR, FORTH can recover control after the message is displayed.) If this is not done, then processing returns to the BASIC main

loop.

-- To allow a LEX file to clean up pending operations which might have been interrupted by the Memory Error.
-- To allow a LEX file to generate a custom Memory Error message with text insertion, by adjusting the values in C(14-13).

10.4 Foreign Language Translators

A Foreign Language Translator is a LEX file whose sole purpose is to translate HP-71 messages from the resident English to a foreign language. It is a simple LEX file which contains nothing but a message table and a poll handler which intercepts the pMEM, pERROR, pWARN and pTRANS polls to substitute alternate message numbers.

10.4.1 BASIC Error Trapping

Using ON ERROR in a BASIC program allows error trapping for applications. In the message handler, the sequence of steps when processing an error is:

1) send out pERROR poll
2) set ERRN (and ERRL)
3) jump to ON ERROR if in effect

A language translator will intercept the pERROR poll and substitute an alternate message number before the ON ERROR jump. Thus, a check of ERRN in the ON ERROR routine must allow for foreign language message numbers.

The following convention has been set up to facilitate error trapping with language translators.

For mainframe messages:
translated message number= ERRN+1000

For other LEX files:
translated message number= ERRN+128

For example, mainframe error 57 is "File Not Found". If an ON ERROR routine is trapping for this error and must allow for foreign language messages, the appropriate statement is:

```
IF ERRN=57 OR ERRN=1057 THEN ....
```

The HPIL error 255031 is "Directory Full". If an ON ERROR routine is trapping for this error and must allow for foreign language messages, the appropriate statement is:

```
IF ERRN=255031 OR ERRN=255159 THEN ....
```

This extended error trapping can be shortened with the user-defined function:

```
10 DEF FNE(X)= (X=ERRN) OR (X=ERRN+128+(X<1000)*872)
```

and the two examples above can be compressed to

```
IF FNE(57) THEN ...
IF FNE(255031) THEN ...
```

The following subsections describe how this convention is implemented.

10.4.2 LEX File Number Sharing

The LEX ID of a language translator is based on the ID of the LEX file whose messages are to be translated. All language translator LEX files which have the same LEX ID will share the same numbering scheme. That is, related language translators will share the SAME four-digit (hexadecimal) message numbers, including LEX ID number and message ID number. This implies that only one language translator will be active in the computer at one time (the first one in the file search order).

Language translator LEX files should not, in general, have any extended BASIC statements or functions, decompile or execution routines, since the proliferation of similar LEX numbers would be confusing for a user trying to determine their source.

10.4.2.1 LEX File #00 (Mainframe) Translation

A Foreign Language Translator for LEX file #00 (mainframe) messages will have LEX #01. Its message table will contain a one-to-one correspondence between mainframe messages and the translated messages. This means that each message in LEX file #01 will have the same meaning as the correspondingly numbered message in the mainframe.

For example, message number 0039 hex (57 decimal, as expressed by ERRN) in the mainframe is "File Not Found". The corresponding message 0139 (1057 as expressed by ERRN) in LEX file #01 must be the foreign language equivalent of "File Not Found".

This one-to-one correspondence of mainframe messages applies to message 1 through 97, and message 229. Message #229 is "(trk ### of ###)", and is referenced by the card reader execution routines; it must also have a translated equivalent.

The building blocks in the mainframe table numbered 230 through 248 are simply frequently-used words. They are NOT messages, per se, since they are never referenced as message constants by a routine calling the message handler. Because of this, a language translator need not contain the same building blocks; even if it does, it need not number them the same. In addition, the language translator may use any building blocks it desires to construct messages, and may number them in any manner that does not conflict with messages 0 through 97 and 229. Building blocks used for this purpose are simply means of saving ROM, and are not subject to the one-to-one correspondence.

Note that the mainframe contains a partial LEX file (all but a file header) numbered 01. This partial LEX file does not contain a message table; therefore, no conflict will arise because of this convention.

10.4.2.2 Other LEX File Translation

For LEX files other than #00 (mainframe), a language translator will have the same LEX number, and its message table will be offset by 128 decimal from the master LEX file table. There will be a one-to-one correspondence between the messages in the two tables, with message number n in the master table being equivalent to message number n+128 in the translated table. (It has been determined that it is unlikely that any LEX file will need more than 127 messages, allowing message ID numbers 128 through 255 to be reserved for the translators. **)

For example, the HPIL ROM has LEX ID=FF (255 decimal). The HPIL message number FF1F (255031 in decimal, as expressed by ERRN) is "Directory Full". A language translator for HPIL messages would also have LEX ID=FF, and the corresponding message FF9F (255159 as expressed by ERRN) would be the foreign language equivalent to "Directory Full."

Building blocks used solely to save ROM (those never referenced as messages by routines calling the message handler) are not "true" messages; they need not have a one-to-one correspondence with building blocks in the translator. Such building blocks need not be duplicated in the translator LEX file, and if they are, they may be numbered in any manner which does not conflict with the numbering of the "true" messages in that LEX file.

** NOTE: The split in the message tables into blocks of size 128 requires that the master LEX file be restricted to messages 1 through 127, and the translator be restricted to messages 128 through 255. Message number 00 (the LEX file name -- see subsection "Message Construction", below) is used by the message handler as a prefix for errors and warnings; if the master LEX file includes it, then the translator file should include it, too (perhaps in a translated form). That is, EVERY message table (including language translators) should have a message 00, unless they do not want a prefix for errors and warnings.

If a LEX file requires more than 127 messages, and its author knows for certain that it will never be subject to language translation, it can use the full range of messages from 0 to 255. Using messages in the range 128-255 will prevent the use of standard message translation and error trapping for future applications.

If a LEX file requires more than 127 messages and its author wants to preserve the capability of standard error trapping with language translation, use of a second LEX ID number is necessary; using a second LEX table will provide 127 more messages.

For details on message range and numbering, see section "Message Table Construction", under "Message Range".

(This restriction to blocks of 128 does not apply to LEX file #01, the translator for the mainframe. This is described in the previous subsection.)

10.4.2.3 HPIL Message Range

Because of a bug in the first version of the HPIL ROM, any translator for this ROM will have to reside in RAM in order to be implemented. The message range was inadvertently left as 00-255; this means that when the message handler goes to look for message 255159, say, it will search this ROM's table, since the range covers this message. In order for a translator to be implemented, it must occur before the HPIL ROM in the file search order, so that its message table will be found first. The easiest way to do this is for the user to copy the HPIL translator into RAM so that it will be found first.

10.4.3 Poll Handlers for Translators

Besides the VER$ poll, a language translator requires a poll handler to intercept pMEM, pERROR, pWARN and pTRANS polls. Upon intercepting these polls, an alternate message number is substituted for the original, providing the message came from the translator's master table. That is, a translator only translates messages from one specific LEX file.

Poll handlers for pMEM, pERROR and pWARN should not set XM=0 (i.e., do not indicate "handled"), since this causes the message to be suppressed. Poll handlers for pTRANS should set XM=0 to indicate that message number has been adjusted to generate a translated message.

The algorithm is described for the two classes of translators:

10.4.3.1 Poll Handler for LEX ID #01

Translators for the mainframe messages (LEX ID #00) have LEX ID #01. The poll handler for pMEM, pERROR, pWARN and pTRANS polls should perform the following:

1) Fetch message number from R0.
2) If LEX ID of message is not 00, then go to 5).
   Else, set LEX ID of message =01 and replace
   message number in R0.
3) If pTRANS poll, exit with carry clear, XM=0.
4) If message ID number is not 88, then go to 5).
   Else, a separate (nested) poll is required to
   translate the insertion message (message #88
   is "TFM WRN L<#>: <insertion message>"):

   3a) Shift number of insertion message to R2(A).
       Swap R0 and R2. Poll with pTRANS constant.
   3b) When returned from nested poll, swap R0 and
       R2. Shift message number to R2(8-5).

5) Return from poll (carry clear, XM=1).

An example in the "HP-71 Code Examples" chapter, "Foreign Language Translation of Messages", demonstrates the assembly language necessary to implement this.

10.4.3.2 Poll Handler for Other LEX Files

Translators for other LEX files (LEX ID's above 00) have the same number as the master LEX file. The poll handler for pMEM, pERROR, pWARN and pTRANS should perform the following:

1) Fetch message number from R0.
2) If the message number does not have the right LEX ID,
   go to 5).
   Else, add 128 to the message number, replace in R0.
3) If pTRANS poll, exit with carry clear, XM=0.
4) If the message allows type (5) insertion (see section
   entitled "Message Table Construction"), a separate
   nested poll is required to translate the insertion
   message.
5) Return from poll (carry clear, XM=1).

An example in the "HP-71 Code Examples" chapter, under "Foreign Language Translation of Messages", demonstrates the assembly language necessary to implement this.

10.4.4 Two Types of Language Translators

An HP-71 design team has come up with two types of language translators: one-shot translators, and selectable translators.

One-shot translators provide a fixed translation capability, in one language only. Selectable translators allow the user to select the language (including English -- "no translation").

10.4.4.1 One-shot Translator

A one-shot language translator is a LEX file which, as long as it is present in the computer, ALWAYS translates messages. Such a LEX file serves only one language, would most likely be RAM based, and would probably be available on a card. Several one-shot translators might be in memory, one each for the mainframe, HPIL, the MATH ROM, etc.

It's possible that one-shot translators for several different languages might reside in memory at the same time (e.g., Spanish, German, French, etc.). In this case, the one that occurs first in the file search order will be the one which is in effect. To switch languages, the file chain must be manipulated by the user (with COPY, PURGE, etc.), so that the new language translator is "selected" by being first in the file search order.

On the other hand, as long as a one-shot translator resides in memory, the resident English language messages cannot be accessed. Only by purging all such translators can the user regain English messages.

Examples of two one-shot translators (one for the mainframe, one for HPIL) are in the "HP-71 Code Examples" chapter, under "Foreign Language Translation of Messages."

10.4.4.2 Selectable Translator

A selectable language translator consists of a "controlling" LEX file, and additional "satellite" LEX files which contain message tables for several different languages. The controlling LEX file provides a keyword to select which language to implement (including the resident English). Such a scheme may be implemented in a ROM, and distributed either separately as a "Translator ROM for Spanish, German, French, ...", or as an integral part of an application pack (such as the Text Editor).

(The selecting keyword and the selecting syntax have not been decided upon.)

The selectable translator scheme offers several advantages over one-shot translators:

1) It allows selecting a particular language for all messages, or suppressing translation entirely.
2) The equivalent collection of one-shot translators would increase the number of LEX files many times over, which would make the the HP-71's processing relatively slower.

The collection of LEX files for a selectable translator would look like this:

Controlling LEX file
-- contains selecting keyword.
-- contains VER$ poll handler for the entire entourage.
-- contains code for implementing the language selection.

First satellite LEX file
-- services mainframe message translation.
-- contains pMEM, pERROR, pWARN and pTRANS poll handlers.
-- contains truncated LEX file and table for mainframe Spanish translation.
-- contains truncated LEX file and table for mainframe German translation.
-- contains truncated LEX file and table for mainframe French translation.
-- etc. (other languages)

Second satellite LEX file
-- services HPIL translation (for example).
-- contains pMEM, pERROR, pWARN and pTRANS poll handlers.
-- contains truncated LEX file and table for HPIL Spanish translation.
-- etc. (other languages)

Third satellite LEX file
-- services MATH ROM translation (for example).
-- contains pMEM, pERROR, pWARN and pTRANS poll handlers.
-- etc. (truncated LEX files and tables)

... As many satellite LEX files as desired.

The term "truncated LEX file" refers to a file which looks identical to a LEX file, except that the file header is omitted. That is, the following fields are left out:

File Name
File Type
Flags
Time

Date
File Length (offset to next file)

The file, then, starts at the Id field. Each truncated LEX file would be identical to a one-shot translator, except for the missing header.

The controlling LEX file has some important housekeeping to perform:

1) When a language is selected, it must open a system buffer (if it doesn't already exist) to store the language name (or a code). This system buffer has ID# "bTRANS".

2) It must go into the LEX system buffer and adjust the addresses of each of the satellite LEX files so that they point to the truncated header of the appropriate languages.

3) At the time of each configuration (pCONFG poll), step 2 must be repeated, using the stored language in the system buffer for reference.

An example of a selectable translator can be found in the "HP-71 Code Examples" chapter, under "Foreign Language Message Translation."

10.5 Message Table Construction

A Message Table contains a list of standard messages; standard messages are those messages which can be displayed by the message handling routines. The table may include error messages, warnings and system messages. One message table serves the mainframe, but each LEX table may have an associated message table to support the parse and execute routines for its keywords.

Messages are identified within a LEX file table by a two-digit hex number. Message number 00 is reserved to be the LEX file name; it is used in the prefix of a message to identify the source of the message. For instance, the HPIL ROM (LEX ID=FF), has message number 00 (and its LEX file name) "HPIL ", so that any error generated by this LEX file will display "HPIL ERR:". If a LEX file does not desire a name on the error prefix, it can leave message 00 out of the tables entirely.

10.5.1 Message Formats

It is recommended that standard messages be kept short, since more than 22 characters in the display will cause scrolling. Scrolling is especially undesirable for an error message.

10.5.2 Message Prefix

The standard error message prefix for mainframe messages is "ERR:"; for warnings, the prefix is "WRN:". This leaves 18 characters for the message before scrolling starts. For a run-time error, the standard message prefix is "ERR Llll:", where lll is the line number (1 to 4 digits). This leaves (16-i) characters, where i is the number of digits in the line number.

Most LEX files will provide an LEX file name to identify a message, such as "HPIL ERR:". Thus, for a LEX file error, scrolling starts at (18-k) characters for a parse or keyboard execution error, and at (16-i-k) characters for run-time errors, where k is the number of characters in the LEX file name, and i is the number of digits in the line number.

10.5.3 Message Construction

10.5.3.1 Message Range

The first entry in a message table is the listed range of messages found in the table. The first byte of the range is the lowest numbered message; the second byte is the highest numbered message.

When the message handler searches the LEX files for a message, it will not find the specified message unless its number is within the listed range of the table. A message table can contain messages outside the listed range; they can be used as local building blocks, but will not be found by the message handler. Such messages cannot be generated as errors or warnings by assembly language routines, and they can't be accessed by MSG$.

Message number 00 is taken to be the LEX file name (for error displaying purposes). Even if a LEX file has message numbers from 6F to E3, for instance, it may include a message number 00 for its LEX file name; message number 00 need not be included in the listed range. Even if message 00 is not included in the listed range, it will be found and used for the error prefix. However, not including it in the listed range will prevent its access by the MSG$ function. For a specific application of this feature, see section entitled "Foreign Language Translators."

The ability to fragment LEX files (have different files with the same LEX ID#, but different message ranges) requires some coordination. Although each of the fragments would include the same message 00 (if they want an error prefix), one of the LEX files will need to include message 00 in its range. This would reasonably be done by the fragment with the lowest range. Again, this is suggested so that MSG$ can access message 00 in the LEX file.

The function MSG$ (in LEX file #82) should be able to access all "true" messages ("true" messages are those which are referenced as errors or warnings, and do not include local building blocks).

Foreign language translation puts restrictions upon the numbering of messages in LEX files with ID# greater than 01. See section "Foreign Language Translators" for details.

Message range
- -- The master LEX file message range (as listed in the range field of the table) must be within the interval 00-127 decimal.
- -- The translator LEX file message range (as listed in the range field of the table) must be within the interval 128-255 decimal.

message number 00
- To be used by BOTH master and translator (if a prefix for errors and warnings is desired) or to be used by NEITHER (if no prefix desired).

messages numbered 01 through 127
- The master LEX file's true messages (those referenced by routines calling the message handler) MUST be in this range. Any other numbers in this range can be used as building blocks by the master LEX file (such building blocks can be accessed by MSG$).

- The translator file can use local building blocks in this range, providing that they are NOT included in its listed range! (Such building blocks cannot be accessed by MSG$.) These building blocks can only be referenced by other messages in the same table.

message number 128
- For a language translator, this message MUST be identical to message 00 in the same table (easy to do with a building block). The reason is that, for example, MSG$(125000), because of the pTRANS poll, will fetch message 125128. (If message 00 is not used, message 128 need not be in the table either.)

- The master LEX file can use message 128 as a local building block, providing that it is NOT included in the message range! (This message will not be accessed by MSG$.) It can only be referenced by other messages in the same table.

messages numbered 129 through 255
- The translator LEX file's true messages (those referenced by routines calling the message handler) MUST be in this range. Any other numbers in this range can be used as building blocks by the translator LEX file (such building blocks can be accessed by MSG$).

- The master LEX file can use local building blocks in this range, providing that they are NOT included in its message range! (Such building blocks cannot be accessed by MSG$.) These building blocks can only be referenced by other messages in the same table.

10.5.3.2 Message Blocks

The term "message block" refers to a complete message entry in a message table, including total length, message number and message cells.

All entries which follow the listed range are standard messages in message blocks. They can be in any numerical order (even message number 00 need not be first), although they can be arranged for more efficient table search: messages near the beginning of the table will be found first.

```
+------------------+
| Min Range Number |     2 nibbles (hex value)
+------------------+
| Max Range Number |     2 nibbles (hex value)
+---------------+--+
| Message Block |        (see below)
+---------------+
| Message Block |
+---------------+
| Message Block |
+---------------+
  ...
+----+
| FF |                   Table Terminator
+----+
```

The first nibble following the range field MUST be a 0 . This means that the FIRST MESSAGE IN THE TABLE MUST HAVE A TOTAL LENGTH OF 16 (or a multiple of 16). Since the table can be arranged in any numerical order, it is easy to move a qualifying message to this location. If there is no message which meets this requirement, construct a dummy message (one whose number is not needed) of 5 blanks, or anything that gives a total length of 16.

A message block length of FF terminates a message table.

Message number 00, the LEX file name, should, if it is included, contain a trailing space.

Each message block consists of several parts:

```
+-----------------+
| Length of Block |      2 nibbles (hex value)
+---------------+-+
| Message ID number |    2 nibbles (hex value)
+---------+---------+
| Cell #1 |              (see below)
+---------+
| Cell #2 |
+---------+
  ...
+---------+
| Cell #n |
```

```
+---+-----+
| C |               "C" nibble, terminates block
+---+
```

Message cells are of seven types:

1) Text cell.

Text cells are preceded by a length field:
one nibble if length <= 11 characters,

```
CON(1)  6
NIBASC  \7 chars\
```

or "B" followed by length nib if length > 11.

```
CON(1)  11
CON(1)  12
NIBASC  \13 chara\
NIBASC  \cters\
```

2) Mainframe Building Block cell.

Identified by an "E" nibble.
This type of cell fetches an entire message from the mainframe table (some building blocks are simply frequently-used words).
For example,

```
CON(1) 14       identifies mainframe bld block
CON(2) =eFILE   fetches "File" building block
```

3) Local LEX file Building Block cell.

Identified by a "D" nibble.
Similar to a Mainframe Building Block cell, this fetches an entire message from the local LEX file. The local building block need not be included in the table's listed range.
For example,

```
CON(1) 13       identifies local building block
CON(2) =eARRAY  fetches "Array" building block
```

4) Different LEX file Building Block cell.

Identified by an "F0" byte ("F" means "special cell"). This fetches an entire message from a different LEX file. Similar to above building blocks, except that this terminates the current message. The calling routine must know that the second LEX file is present! The referenced message must be included in the listed range of the second LEX file.
For example,

```
NIBHEX F0        identifies diff LEX bld block.
CON(4) =eXIRR    transfers to "XIRR" message in
                   another LEX file.  The 4-nibble
                   constant contains the LEX#
                   and message# of the message.
```

5) Indirect message cell.

Identified by an "F1" byte ("F" means "special cell"). This cell identifies a transfer to another message text; the message number is passed to the message handler by the calling routine. The indirect message number can call any message in any LEX table, provided the message is included in the listed range of the second LEX file. (The 4-digit message number is passed in R2 to the message handler MFWRN -- see subsection "Entry Conditions for MFWRN", above.) (A type (5) insertion requires special handling in foreign language translators -- see the section which describes their implementation. Consider this overhead when using type (5) insertions.)
For example,

```
NIBHEX F1        identifies indirect msg cell
```

6) Insert Text cell: no trailing space.

Identified by an "F2" byte ("F" means "special cell"). This cell identifies the fixed location where the message allows the calling routine to insert text. The text is inserted without a trailing space. (The text is passed to MFWRN through codes in R2 -- see subsection "Entry Conditions for MFWRN", above.)
For example,

```
NIBHEX F2        identifies insertion point,
                 no trailing space.
```

7) Insert Text cell: with trailing space.

Identified by an "F3" byte ("F" means "special cell").
This cell identifies the fixed location where the message allows the calling routine to insert text. The text is inserted WITH a trailing space. (The text is passed to MFURN through codes in R2 -- see subsection "Entry Conditions for MFURN", above.)
For example,

```
NIBHEX F3       identifies insertion point,
                with trailing space.
```

A message terminates with a "C" nibble.

There are two levels of building block "subroutines" available; that is, a building block itself may reference one other building block.

Message numbers need not be entered sequentially in a message table. In particular, message numbers may be missing entirely. This permits reserving a block of numbers for a certain type of message (such as 80 through 90 for errors concerning matrix dimensions).

10.5.3.3 ROM Savings With Building Blocks

Building blocks (either local -- type (3), or mainframe -- type (2)) can save many bytes of ROM. Here's the formula for deciding whether you will save ROM by making a string a local building block:

Let n= #characters in string (2n= #nibbles)
Let j= #times the string is used.

Then k= #times necessary to guarantee savings with bldg block
k'=#times necessary to guarantee loss with bldg block

i.e., if j> k, guaranteed savings by using bldg block
if j<k' , guaranteed loss
if k'<j<=k , check individually (*)

In table form:

n	k'	k
1	inf	inf
2	6	11
3	4	5
4	3	3

n (chars)	occurrences 1	2	3	4	5	6	7	8	9	10	11...	
2	-	-	-	-	-	?	?	?	?	?	+	+
3	-	-	-	?	+	+	+	+	+	+	+	+
4	-	-	+	+	+	+	+	+	+	+	+	+

```
 5   3   3  ||     5  | -  -  +  +  +  +  +  +  +  +  +  +
 6   2   3  ||     6  | -  ?  +  +  +  +  +  +  +  +  +  +
 7   2   2  ||     7  | -  +  +  +  +  +  +  +  +  +  +  +
 8   2   2  ||     8  | -  +  +  +  +  +  +  +  +  +  +  +
 .   2   2  ||
 .   2   2  ||   -= loss (or breakeven) if bld block is used
 .   2   2  ||   += savings if bld block is used
            ||   ?= check individually (*)
```

*Note: In the cases where you must check individually to verify a savings, the factor which affects this is the possible breaking up of a type {1} cell into two type {1}'s and a building block. For example, consider the following type {1} message cells:

```
    CON(1) 9                (Length of NIBASC=10)
    NIBASC \No Matches\
and
    CON(1) 9                (Length of NIBASC=10)
    NIBASC \Good Match\
```

Each cell takes 21 nibbles. If you wanted to make " Match" a local building block, these cells would now look like this:

```
    CON(1) 1                (Length of NIBASC=2)
    NIBASC \No\
    CON(1) 13               (Indicator for type {3})
    CON(2) eMATCH           (Symbol for building block)
    CON(1) 1                (Length of NIBASC=2)
    NIBASC \es\
and
    CON(1) 3                (Length of NIBASC=4)
    NIBASC \Good\
    CON(1) 13               (Indicator for type {3})
    CON(2) eMATCH           (Symbol for building block)
```

Whereas the cells originally took 21 nibbles each, the first case now uses 13 nibbles and the second uses 12. The difference is that the first needs another length nibble for the "\es\" cell. This new type {1} fragment is the factor which requires some cases to be determined individually.

The building block for " Match" would take 6 nibbles for overhead (message length, number and terminator nibble), plus 12 nibbles for the characters " Match". The entire building block would add up to 18 nibbles, whereas the savings from the above cells was only 17.

Formula: Use a building block if

    the number of nibbles required for the building block
      2*(#characters in bld block)
      + overhead for bld block (6 nibs, if under 11 chars)

+ 3 nibbles for each bld block reference (3j)
+ extra length nibbles for fragmented cells

is less than the total number of nibbles it would take to leave in the characters without bld blocks
2*(#characters in bld block)*j

i.e., use a building block if

$2n+6+3j+x < 2nj$

where n= #characters in building block
j= number of references to building block
x= number of new type(1) fragments

10.5.3.4 Example

An example of a message table:

```
        CON(2) 43              Range: minimum msg number
        CON(2) 50              Range: maximum msg number
*
*!!! XMSG49 is placed first because it has a total length of 16.
*!!! The first nibble following the range field MUST be a 0 !!
*!!! (See note in subsection entitled "Message Range", above.)
*!!!
XMSG49 CON(2) (LEXNAM)-*       Length to next msg.
       CON(2) 49               Message number.
       CON(1) 14               Mainframe building block.
       CON(2) =eFILE           Use "File" from m/f.
       NIBHEX F3               To insert file name here w/space.
       CON(1) 13               Local building block.
       CON(2) =eXMSG1          Use "Private" building block.
       CON(1) 0
       NIBASC \!\
       CON(1) 12               Msg terminator.
*
LEXNAM CON(2) (XMSG43)-*       Length to next msg.
       CON(2) 0                Msg 00 reserved for LEX file name.
       CON(1) 3                Length-1 of NIBASC.
       NIBASC \XRM \           LEX file name.
       CON(1) 12               Message terminator.
*
XMSG43 CON(2) (XMSG48)-*       Length to next msg.
       CON(2) 43               Message number.
       CON(1) 6
       NIBASC \Private\        Text.
       CON(1) 12               Message terminator.
*
XMSG48 CON(2) (XMSG50)-*       Length to next msg.
```

```
        CON(2) 48            Message number.
        CON(1) 14            Mainframe building block.
        CON(2) =eILPAR       Use "Illegal Param" from m/f.
        CON(1) 12            Message terminator.
*
 XMSG50 CON(2) (XMSGf)-*     Length to next msg.
        CON(2) 50            Message number.
        CON(1) 13            Local building block.
        CON(2) =LEXNAM       Use "XRM " building block.
        CON(1) 7
        NIBASC \Catalog:\    Text.
        CON(1) 12            Msg terminator.
*
 XMSGf  NIBHEX FF            Table terminator.
```

The "HP-71 Code Examples" chapter contains more examples of message tables, under "Foreign Language Message Translators."

```
+-------------------------------------------+-----------------+
|                                           |                 |
|   FILE SYSTEM                             |   CHAPTER  11   |
|                                           |                 |
+-------------------------------------------+-----------------+
```

11.1 File Chain Structure

The HP-71 maintains a file area in main RAM which is comprised of a linked list, or chain, of file entries. Each file entry in the chain begins with a file header, which contains identifying information about the file along with the link to the next file entry in the chain. This link is referred to as the "File Chain Length field." The end of the file chain is marked by a zero byte. Each plug-in ROM module and independent RAM also contains its own file chain. A later section in this chapter describes the order in which the various file chains are searched for a given file.

Certain file types require special information between the file header and the file's data. The Implementation Field, when present after the file header, corresponds to the Implementation Field of the file's directory entry when it is copied to or from mass media, such as magnetic tape, which use the HP Logical Interface Format (LIF). The Implementation Field is always present after the file header for files of copy codes 1 (e.g. DATA) and 8 (user-defined), and otherwise is never present after the file header. The DATA file type, for example, requires that its Implementation Field be present to indicate the number and length of records in the file.

Furthermore, some file types require a subheader immediately following the file header or Implementation Field. The BASIC file type, for example, requires a 6-byte subheader which contains two pointers into the data (program) portion of the file (see the description of the BASIC file type later in this chapter). The subheader presence, length, and format depends upon the file type. When a file containing a subheader is copied to external media, the subheader is not stored in the file's directory entry like the Implementation Field, but is stored at the beginning of the data portion of the file. In this way the subheader is restored to its correct position after the file header when the file is copied back into memory.

The following diagram shows the general structure of the file chain, showing one file without Implementation Field or subheader, one file with Implementation Field, and one file with a subheader.

FILE CHAIN STRUCTURE

```
+--------------------------+
|                          |
|       FILE HEADER        |
|                          |
|- - - - - -- - - - - - - -|
| File Chain Length field  |--------+
+--------------------------+        |
|                          |        |
|                          |        |
|        File Data         |        |
|                          |        |
|                          |        |
+--------------------------+        |
|                          |<-------+
|       FILE HEADER        |
|                          |
|- - - - - -- - - - - - - -|
| File Chain Length field  |--------+
+--------------------------+        |
|   Implementation Field   |        |
+--------------------------+        |
|                          |        |
|                          |        |
|        File Data         |        |
|                          |        |
|                          |        |
+--------------------------+        |
|            .             |<-------+
               .
|                .         |--------+
+--------------------------+        |
|                          |<-------+
|       FILE HEADER        |
|                          |
|- - - - - -- - - - - - - -|
| File Chain Length field  |--------+
+--------------------------+        |
|     File  Subheader      |        |
+--------------------------+        |
|                          |        |
|                          |        |
|        File Data         |        |
|                          |        |
|                          |        |
+--------------------------+        |
| 00 byte (ends chain)     |<-------+
+--------------------------+
```

11.1.1 File Header

The format of the file header is described below.

FILE HEADER

```
+-----------------------------------+
| File Name                         |    16 nibbles
+----------+------------------------+
| File Type |                             4 nibbles
+-------+---+
| Flags |                                 1 nibble
+-------+---+
| Copy Code |                             1 nibble
+-----------+---+
| Creation Time |                         4 nibbles
+---------------+
| Creation Date |                         6 nibbles
+---------------+---+
| File Chain Length |                     5 nibbles
+-------------------+
```

Each file has a file header. The file header contains the 8 character file name in ASCII, blank filled on the right (high memory).

The 4 nibble file type field contains the file's 16-bit signed integer file type, ranging from -32768 to 32767. HP-71 file types are explained in the File Types section.

Next are 4 system flags. The two bits in the low end of the flag field indicate file protection. When set, the lower of the two bits indicates a file is SECURE; the higher of the two bits indicates a file is PRIVATE. The following two bits of the flag nibble are unused.

File Header Flags

```
        +--+--+--+--+
  Low   |  |  |  |  |   High
        +--+--+--+--+
         ^  ^
         |  |
         |  +---- PRIVATE
         +--- SECURE
```

The next file header field is the Copy Code nibble. This nibble indicates the file attributes neccessary for external copying. The specific encoding of Copy Code is explained under File Type Table in the "Table Formats" chapter.

The creation time and date are set when the file is created. Creation date and time are stored in BCD. The time field contains 4 nibbles; the minutes are in the low byte and the hour is in the high byte. The date field contains 6 nibbles; the day is represented in the low byte, the month in the next byte, and the year in the high byte. For example: The internal representation of 3:45 12/16/81 would be as follows:

```
 Time   Date
+-+-+-+-+-+-+-+-+-+-+
|5|4|3|0|6|1|2|1|1|8|
+-+-+-+-+-+-+-+-+-+-+
```

The next entry is the File Chain Length field. This is the offset to the next file (header) in memory.

11.1.2 Implementation Field

The HP-71 HP-IL Interface Module maintains external file systems on tape or other mass memory devices according to the HP Logical Interface Format standard. This format defines an 4 byte field in each file's directory entry, called the Implementation Field, which may contain arbitrary information according to the file type.

For certain file types, this 8 nibble Implementation Field must immediately follow the file header when the file is present in memory. Whether or not the Implementation Field is present is determined by the file's copy code, which is taken from the File Type Table entry for that file type (the copy code is stored in the file header). Copy codes 1 and 8 always have the Implementation Field present after the file header; all other copy codes have no Implementation Field present after the file header.

When a file is copied to external mass media, the Implementation Field written to the new file's directory entry is either generated by the operating system according, or is copied directly from the Implementation Field present after the file's header. See the section below on "File Header Structure by Copy Code" for further information.

```
+----------------------+
| Implementation Field |   8 nibbles
+----------------------+
```

11.1.3 File Subheader

Aside from the file header format and Implementation Field given above, for certain file types additional information may accompany the file header in the form of a subheader, which immediately follows the file header or Implementation Field. Subheaders must be an even number of nibbles in length and must be no more than 250 nibbles long. The format of a subheader is determined by the file type.

The presence of a subheader after the file header or Implementation Field is determined indirectly by the Offset to Data field in the File Type Table entry for that file type. This field gives the offset from the start of the File Chain Length field in the file header, to the actual start of data, skipping over the Implementation Field and/or the subheader, if either are present. The presence and length of the subheader can therefore be determined using the Offset to Data field and the copy code (which determines whether the Implementation Field is present) according to the chart below. Refer to the following section for further details concerning copy codes.

File Header Structure	Data Offset in Nibs	Applicable Copy Codes 0	1	2	4	8
* No subheader, No Implementation Field	5	x		x	x	
* No subheader, Implementation Field	13		x			x
* Subheader of n nibs, No Implementation Field	5+n	x			x	
* Subheader of n nibs, Implementation Field	13+n					x

11.1.4 File Header Structure by Copy Code

The presence of the Implementation Field after the file header is determined by the file's copy code, as outlined in the chart below. The copy code originates in the File Type Table entry for that file type, and is stored in the file header.

FILE HEADER STRUCTURE BY COPY CODE

Copy code:	0	1	2	4	8
Exemplary file type	BASIC,KEY, LEX, etc	DATA	SDATA	TEXT	User-defined
Imp. Field after file header?	No	Yes	No	No	Yes
Imp. Field contents on ext media	Format A	Format B	Format C	Zero	User-defined
May have subheader?	Yes	No	No	Yes	Yes

	Nib	Contents
Format A:	7 - 2	Length of file in nibbles
	1 - 0	Unused (zero)
Format B:	7 - 4	Unsigned integer specifying number of records in file, byte reversed
	3 - 0	Unsigned integer specifying number of bytes in record, byte reversed
Format C:	7 - 6	Protection; if set to 08 hex, file may not be purged, renamed, or written to; otherwise should be set to 0
	5 - 2	Signed integer specifying number of registers (8-byte records) in file
	1 - 0	Unused (zero)

11.2 File Types

The following file types are directly supported by the HP-71 mainframe. OEM software developers may support other file types by first reserving the file type with HP (see the "HP-71 Resource Allocation" chapter), and then by including the appropriate poll handlers in a LEX file. Each file type is identified by a 16-bit value which conforms to HP's Logical Interchange Format for Mass Media.

When HP-71 files are stored on external media, file security and privacy are encoded, if applicable, in the numeric file type as shown in the chart below. When files are stored in memory, privacy and security are encoded in the flags field of the file header, and the file type stored in the file header is ALWAYS the normal file type.

			Hex Numeric Value			
Type	Description	Security**:	Normal	S	P	E
BASIC	Tokenized BASIC program		E214	E215	E216	E217
BIN	HP-71 Microcode		E204	E205	E206	E207
DATA	Fixed Data		E0F0	E0F1	n/a	n/a
KEY	Key Assignment		E20C	E20D	n/a	n/a
LEX	Language Extension		E208	E209	E20A	E20B
SDATA	Stream Data		E0D0	n/a	n/a	n/a
TEXT	ASCII text, in LIF Type 1 format		0001	E0D5	n/a	n/a

** Meaning of the Security Symbols:

Symbol	Meaning
Normal	File is not protected
S	File is SECURE
P	File is PRIVATE
E	File is SECURE and PRIVATE

11.2.1 File Protection

The default protection for a file is no protection. A file with no protection can be edited, purged and executed. File protection is specified through the SECURE and PRIVATE commands.

File protection is detected by two bits in the flag field of the file header. When set, the lowest bit of the field indicates the file is SECURE; the next bit (bit 1) indicates the file is PRIVATE.

11.2.2 BASIC

A BASIC file contains tokenized BASIC programs or subprograms, and is created by the HP-71 BASIC editor. A BASIC file has a copy code of 0, a 12-nibble subheader, and no Implementation Field present after the file header.

A main program, if present, must start immediately after the subheader. Any subprograms present are chained sequentially thereafter. The file's Subprogram chain is headed by a link in the subheader.

11.2.2.1 Subheader

The BASIC file subheader contains 3 fields. The Subprogram Chain Head contains the first pointer of the Subprogram chain in the file. Similarly, the Label/User-Defined Function Chain Head contains the starting pointer of the chain of labels and user-defined functions within the main program. A permanent EOL (hex F0) always precedes the start of data (start of the first line) for a BASIC program file. This causes every program line to conform to the same format.

BASIC Subheader

```
+--------------------+
|     Subprogram     |     5 nibbles
|     Chain Head     |
+--------------------+
| Label/User-Defined |     5 nibbles
|      Function      |
|     Chain Head     |
+--------------+-----+
               | F0  |     2 nibbles
               +-----+
```

The chain head and links have the following values and meanings:

Chain head or chain link	Meaning
00000	Chain is not yet established (head only)
nnnnn	Offset to next link
FFFFF	End of chain

11.2.2.2 Subprogram Chain

The purpose of the Subprogram chain and Label/User-Defined Function chain is to speed up searching for subprograms, labels, and user-defined functions.

Subprograms are only chained within a file. The Subprogram chain head contains the offset in nibbles to the next chain link in the file. The only two BASIC statements in this chain list are SUB and ENDSUB. A five-nibble relative address is tokenized in association with these statements, which is used to hold the link to the next entry in the chain.

11.2.2.3 Label/User-Defined Function Chain

The Label/User-Defined Function chain is similar to the Subprogram chain, except label declaration and user-defined function definitions are chained within a program or subprogram. If a file contains one main program and several subprograms, the main program and subprograms will each have their own Label/User-Defined Function chain.

Statements and other constructs linked in this chain are:

Label declarations
DEF FN
ENDDEF

A five-nibble relative address is tokenized in association with these constructs, which is used to hold the link to the next entry in the chain.

11.2.2.4 Statement Tokenization

A BASIC program line begins with a line number and terminates with an End of Line (EOL) token. A program line may contain multiple statements. A multi-statement line is preceded by an @ token. Following each line number or @ token is a statement length byte. This statement length is a relative offset to the next terminating token. Statements within a BASIC file are linked together using these relative offsets.

See the subsection on Statement Tokenization in the "Statement Parse, Decompile, and Execution" chapter for examples of statement tokenization.

11.2.3 BIN

A BIN file is a binary or machine language file which is executed directly by the operating system. A BIN file is created using an assembler such as the FORTH/Assembler ROM. It has a copy code of 0, a 12-nibble subheader, and no Implementation Field is present after the file header.

Each BIN file may contain one or more subpgrograms, which are linked in a manner similar to BASIC files. However, each BIN program MUST have a main program, since a BIN file may be executed directly by a RUN statement. This main program should end with the statement:

```
GOSBVL =ENDBIN
```

in order to end execution of the main program and return control to the operating system. If no useful main program is appropriate to a BIN file, the "main program" should consist only of the above statement.

A binary file's main program can be invoked by RUN or CHAIN. It may also be called as a subprogram by the CALL statement. (In this case no parameters will be passed, and the subprogram will have the caller's local environment.)

11.2.3.1 Subheader

The subheader of a BIN file is the same length as that of a BASIC file, and has a similar format. Its Subprogram chain field is used to chain subprograms within the binary file. The Label/User-Defined Function chain field is always FFFFF (empty) since there are no labels and user-defined functions within the context of a binary program. The "20" code at the end of the subheader is a filler to make the BIN subheader size equal to that of the BASIC file subheader to facilitate use of common access routines.

BIN Subheader

```
+-------------------+
|    Subprogram     |    5 nibbles
|    Chain head     |
+-------------------+
|       FFFFF       |    5 nibbles
+-------------+-----+
              | 20  |    2 nibbles
              +-----+
```

11.2.3.2 Subprogram Chain

The purpose of the Subprogram chain is to enable a BASIC program to CALL a binary subprogram and pass parameters to it, just like to CALL a BASIC subprogram.

If there are binary subprograms in the BIN file, each binary subprogram must start with a tokenized SUB statement, which is tokenized exactly as in a BASIC statement, except that a line number is not required. The SUB tokenization starts with the 2-nibble line length field, then the tSUB token, then the 5-nibble Subprogram chain link field, then the rest of the SUB statement. The tokenization ends in a format that parallels the last 7 nibs of the BIN file subheader: a 5-nibble Label/User-Defined Function field set to FFFFF hex (meaning a null chain) followed by the terminating code "20". The first machine language instruction of the binary subprogram then follows immediately. See the section on "SUB Tokenization" in the "Statement Parse, Decompile, and Execution" chapter.

The mechanism for chaining subprograms in a BASIC file is the CHAIN routine. However, this routine will not work for a binary file. The chaining of the subprograms in a binary file has to be done by the assembler programmer. At execution time, if a BIN file's Subprogram chain has not been established, the binary subprograms in this file will not be found.

11.2.4 DATA

A DATA file is created by the CREATE statement. It has a copy code of 1, no subheader, and its file header is followed by an Implementation Field.

11.2.4.1 Implementation Field

The DATA file Implementation Field is always present after the file header. It contains two 16-bit unsigned integers which give the number and length of records in the file. These integers are stored in byte-reversed format when the file is written to external media (that is, the low-order byte is written first) so that, when the file is in memory, these fields may be conveniently read (using an instruction such as A=DAT1 4).

DATA Implementation Field

```
+-----------------------+
|   Number of records   |  4 nibbles
|    (byte-reversed)    |
+-----------------------+
| Record length in bytes |  4 nibbles
|    (byte-reversed)    |
+-----------------------+
```

11.2.4.2 File Structure

A DATA file is a series of records with fixed record length. Within a record, numeric and string data is stored in sequentially contiguous segments. If a string data item overflows the bounds of a record in sequential access, it is broken into smaller segments.

If one or more bytes remains in the current record but this is not enough to write the next data segment, an End-of-record byte (see below) is written and the file is positioned to the next record in order to write the data segment.

The first byte of a data segment indicates the type of the data, and is called the determiner byte:

Data Segment Determiner Byte	Meaning	Data Segment Length
2 BCD digits	Floating point number	16 nibs
FF hex	End of file	2 "
EF "	End of data in this record	2 "
DF "	Entire string falls in this record	6+n "
CF "	Start of string is in this record	6+n "
7F "	Middle of string is in this record	6+n "
6F "	End of string is in this record	6+n "

where n is the length of the data portion of the string data segment, determined by a 16-bit unsigned byte count which immediately follows the determiner byte of the data segment. For Start-of-string (CF) and Middle-of-string (7F) determiners, this byte count is NOT the length of the data segment, but is the remaining string length in bytes. In this case the end of the data segment is determined by the end of the current record, with which it must coincide. For instance, if the byte count for a Start-of-string (CF) segment is 0032 hex, it means the entire string is 50 bytes long but not all of the 50 bytes is in this record. So the beginning of next record must be a Middle-of-string

(7F) or End-of-string (6F) data item.

STRING DATA FORMAT

In the case of string data, the two bytes immediately following the determiner byte contain a 16-bit unsigned integer which specifies the total remaining length in bytes of the string data. The determiner byte and the two-byte length count are NOT part of the data itself (and are not included in the length count).

When stored on mass media, the length count field is byte-reversed, as in the HP-85, with the low order byte written first. For example, 01AB hex is written "AB01". This is so that, when the file is in memory, this field may be conveniently read as a normal 4-nibble number (using an instruction such as A=DAT1 4).

If a string data item is written sequentially to a DATA file and the string is too long to fit into one logical record, it will be stored in consecutive logical records. The first portion of the string, which must contain at least one character, will be prefixed with a Start-of-string determiner (CF hex). The logical record with the end of the string will contain the End-of-string determiner, the remaining length of the string (at least 1), and the remainder of the string data. All other records which contain part of the string will contain the Middle-of-string determiner (7F hex), the remaining length of the string, and a section of the string.

Each byte of a string may have a value between 0 and 255 decimal.

STRING DATA SEGMENT FORMAT

```
+-----------------------+
| String Data Segment   |     1 byte
|   Determiner Byte     |
+-----------------------+
|                       |
| Remaining Data Length |     2 bytes
|   (byte-reversed)     |
|                       |
+-----------------------+
|                       |
|      String data      |     n bytes
|                       |
+-----------------------+
```

NUMERIC DATA FORMAT

Each numeric value is represented as an 8-byte register. All values written to this file type are normalized, except in the case of IEEE exceptional values explained below. The register is divided into 3 BCD fields:

NUMERIC DATA ITEM FORMAT

Field	Size in Digits	Description
Mantissa sign	1	Symbol is MS. 0 - Positive 9 - Negative
Manitssa	12	Digits are referred to as M0 through M11, with M0 the most significant digit. M0 is nonzero for normalized nonzero numbers.
Exponent	3	Digits are referred to as E0 through E2, with E0 the most significant digit. E0 may be non-BCD for exceptional values described below. For normalized values, if the exponent is: 0 to 499 it is represented as 0 to 499; -1 to -499 it is put in 10's complement and represented as 999 to 501.

The register is written to the file as follows:

```
-----------------------------------------------------------------
| E1 E2| E0 MS| M10 M11| M8 M9| M6 M7| M4 M5| M2 M3| M0 M1|
-----------------------------------------------------------------
   ^                                                      ^
first byte                                         last byte
```

MS : Mantissa sign
M0 : Most significant digit of mantissa
M11: Least significant digit of mantissa
E0 : Most significant digit of exponent
E2 : Least significant digit of exponent

For example, the value $3.14159265359 * 10^{-123}$

would be written as :

```
-----------------------------------------
| 77 | 80 | 59 | 53 | 26 | 59 | 41 | 31 |
-----------------------------------------
```

SUMMARY OF BCD NUMERIC VALUE REPRESENTATION

	E1	E2	E0	MS	M1-M11	M0
Normalized:						
Nonzero	----e----			s	---n---	
Zero	0	0	0	s	0	0
Denormalized	0	1	5	s	n	0
	----d----					
Positive Infinity	0	0	F	0	m	m
Negative Infinity	0	0	F	9	m	m
Not-a-Number (NaN)	--c--		F	s	---t---	

Where :

F = Fifteen
c = Class of NaN (non-zero BCD integer, 1-99).
e = 10's complement exponent; any BCD integer except 500
d = Denormalized exponent 501, which is -499 in 10's complement
m = Meaningless
n = Non-zero BCD integer
s = Sign (0 or 9)
t = Tag identifying origin or type of NaN if class other than 99, else meaningless

Normalized Values

Generally, a BCD number is normalized and within the range of -1.00000000000 E -499 to -9.99999999999 E 499 if negative and +1.00000000000 E -499 to +9.99999999999 E 499 if positive. A number is considered normalized if M0 is nonzero, or if M0 is zero and M1 through M11 are also zero.

Exceptional Values

However, certain mathematical operations may result in an exceptional value that may not be normalized, as in the case of underflow, or may not even be a real number, as in the case of Infinity or Not-a-Number (NaN). These values are encoded in the following manner:

```
a. If E0 = F, the number is either Infinity or Not-a-Number (NaN)
     and if E1&E2 = 00 - the number is Inf (infinity)
     and if E1&E2 # 00 - the number is NaN (E1&E2 are the
                              class number of the NaN)
```

The IEEE standard states that in the case of NaN, the sign of the mantissa and the mantissa may contain system specific information regarding the origin of the NaN. For example, there may be information in the mantissa stating the line where the NaN occurred, the error number generated, and the origin of the NaN, such as 0 divided by 0 or square root of a negative number.

The format by which any extra information has been encoded in the mantissa is identified by a 'class number' which is contained in nibbles E1 and E2 of the exponent. The class number is a BCD number in the range 1 to 99. Currently the only class number defined is 99, which means no useful information is contained in the mantissa. To reserve a class number, contact the PCD LIF coordinator.

```
b. If E0 # F and M0 = 0, the number is either 0 or denormalized
     and if exponent = 0 - the number is zero
     and if exponent # 0 - the number is denormalized
                              e.g. 0.00012E501.
```

11.2.5 KEY

If no system file 'keys' exists, then if a key is redefined using the DEF KEY statement or if a MERGE is done using a KEY file, a new KEY file 'keys' is created. This is the only way in which KEY files are created.

The KEY file type has a copy code of 0, no subheader, and no Implementation Field is present after the file header.

11.2.5.1 File Structure

Each entry in a KEY file is a key assignment. Entries are encoded as follows:

```
+---------+--------------+----------------+-----------------+
| Keycode | Entry length | Assignment Type | String constant |
+---------+--------------+----------------+-----------------+
```

```
Keycode    : 1 byte hexadecimal key number;
             Keys are numbered in row major order
Entry
 Length    : 1 byte representation of the entry length;
             Length from keycode to next entry or end of file
Assignment
 Type      : 1 nibble assignment type
             0 = Automatically sends End Line
             1 = No End Line sent     (specified by ; in DEF KEY)
             2 = Direct Execute       (specified by : in DEF KEY)
```

11.2.6 LEX

Language Extension (LEX) files are the most powerful type of software file used by the HP-71 operating system. They are typically created by an assembler such as the FORTH/Assembler ROM.

The LEX file type has a copy code of 0, no subheader, and no Implementation Field is present after the file header.

11.2.6.1 File Structure

The structure, creation, and use of this file type is described in detail in the "Language Extension and Binary Files" chapter.

11.2.7 SDATA

The SDATA file type is the data file format used by the HP Series 40 calculators (41C, 41CV, 41CX) under the catalog name of DA. The HP-71 can read string or numeric data from this file using READ#. However, the HP-71 can write only numeric data to this file type, using the PRINT# command. An SDATA file is created by the CREATE statement.

The SDATA file type has copy code of 2, no subheader, and no Implementation Field is present after the file header.

11.2.7.1 File Structure

The SDATA file is a collection of registers. A register is always 8 bytes in length and can contain a BCD floating point number or a string data item of up to 6 characters.

The format of a number is the same as that used in the HP-71 DATA file (see the description of DATA file structure earlier in this chapter). If the register contains string data, the sign field S will be equal to 1. The characters are stored in nibbles M1 to M10, right justified with leading zeros.

11.2.8 TEXT

A TEXT file is created by the CREATE statement. The TEXT file type has a copy code of 4, no subheader, and no Implementation Field is present after the file header.

11.2.8.1 File Structure

The format of the TEXT file is determined by the HP Logical Interface Format (LIF) standard for the ASCII Interchange file type (LIF file type 1). A TEXT file is a series of contiguous variable length records. Each record starts with a two-byte data length count (in bytes). A data length count of FFFF hex marks the end of the file (however, the end-of-file marker is not required to be present). If the data length count is FFFF hex, there is no data in the record.

Otherwise, if the data length count is odd, a pad byte of arbitrary content is appended to the end of the record so that the total record length will be an even number of bytes. Therefore, if the data length is an even number, the total length of the record (in bytes) is the data length plus 2. If the data length is an odd number, the total length of the record is the data length plus 3.

Note that when a TEXT file is stored on external media, the data length count field is NOT byte-reversed, in accordance with the HP Logical Interface Format standard mentioned above. This means that when this field is read into a register using an instruction such as A=DAT1 4 the two bytes must then be reversed before the value can be properly interpreted as a number. The routine SWPBYT is used for this purpose.

TEXT FILE RECORD FORMAT

```
+--------------------+
|                    |     2 bytes (FFFF hex marks
|    Data Length     |             end-of-file)
| (NOT byte-reversed)|
|                    |
+--------------------+
|                    |     n bytes (where n is
|       Data         |             data length)
|                    |
|                    |
|                    |
|                    |
+--------------------+
.     Pad Byte       .     1 byte  (only present if
......................              odd data length)
```

11.3 Copying a File

11.3.1 Copying to/from Card

The FILCRD and CRDFIL subroutines provide for data transfer between memory and cards.

FILCRD copies a file to cards. Input conditions call for providing the address of the file to be copied out, the new name to be used on the card, and a flag indicating privacy. Files may be copied out from any memory device to cards.

Routine CRDFIL copies a file in from cards. Input conditions call for providing the name of the file on card (if specified) and the name to be used in RAM (if specified). Files may be copied in to main RAM only.

Both routines prompt the user and handle the complete copy operation. They return if the copy was successful, and take error exits if the copy errors out or is aborted.

The CARD reader buffer is used to hold a copy of the card header during card reader operations. During CAT CARD or CAT$(1,":CARD"), a somewhat larger buffer holds not only the card header, but a dummy file header used by the catalog entry formatting routines.

11.3.2 Copying to/from External Media

If other devices are specified in the copy command, such as "COPY A:TAPE" or "COPY 'A:TAPE'", the filespec parse poll and filespec execute poll give lexfiles the opportunity to recognize and act on the commands. See the poll interface descriptions for more details.

11.3.3 Copying to/from Other Memory Devices

The HP-71 mainframe code does not support copying to or from memory devices other than RAM. However, hooks exist in the COPY code to handle future devices, such as EEPROMs, EPROMs, PROMs, or whatever else may come along.

When COPY is asked to copy to an external memory device, it examines the configuration table to determine the memory type. If it is not RAM, it will poll for a copy handler. Failure to find a copy handler will result in an "Illegal File Spec" error.

Details on the polling conditions can be found in the documentation on the pCOPYx poll.

11.4 Opening a File

A file can be opened by executing the ASSIGN # statement or by call?? routine OPENF internally. The information required to access the fi?? be written to an entry in the File Information Buffer (FIB), which ?? system buffer maintained by the operating system.
Up to 64 files can be opened at the same time since there is room f?? this many entries in the FIB.

All access to an opened file is controlled by its entry in the FIB, which is identified internally by an FIB entry number. This number, also referred to as the file's entry in the FIB, is not to be confused with a channel number which may (or may not) be associated with the FIB entry through the ASSIGN buffer. When discussing a particular opened file, the file's entry in the FIB is also loosely referred to as "the FIB" for brevity.

Whenever an opened file is accessed, the file pointer in the file's FIB entry should be upd?? When the file is closed, as with the CLOSEF utility, its FIB entry ??

removed.

The format of the FIB entry is given in the "Table Formats" chapter??

11.5 File Searching

When presented with the name of a file to find in memory, the operating system automatically searches the various file chains?? according to the algorithm described here. The operating system do?? automatically search for a file on external devices.

If no device is specified for the search, file searching starts wit?? RAM and continues onto the ports, in port specifier order (only ROM?? Independent RAM ports are searched). If the Main RAM file chain is specified (:MAIN), only that chain is seached. S?? the file chain on a particular port may be specified with :PORT(n). Or, a file seach may be restricted to only the port file chains if :PORT is used without a particular port specified.

The routine used in internal file searching is FINDF. A detailed description of its algorithm is given below.

```
        FINDF File Search Algorithm
        ---------------------------

    Clear Single File Chain Flag (S8);
    If there is no file chain specified
      THEN goto B;
    If :MAIN is specified
      THEN goto A;
    Save file name in R2;
    If :PORT is specified (search all ports)
      THEN goto F;
    Set Single File Chain Flag (S8);
    Call ROMF-1; goto G

 A: Set Single File Chain Flag (S8);
 B: Set up to search Main RAM;
    Clear S6 (Initial Port Search not Done)
 C: Search file chain;
    If file found in this chain
      THEN exit with carry clear;
    If S8 is set
      THEN load up error; exit with carry set;
    If S6 is clr (Initial Port Search not done)
      THEN Save file name in R2;
 F:         Call ROMCHK;
 G:         Restore file name; Set S6;
            If no (more) plug-ins
```

```
          THEN load error; exit with carry set;
          ELSE goto C;
  Call ROMFND; goto G.
```

11.6 File Creation

Mainframe files are created by the routine CREATF, and be created in either Main RAM or on an Independent RAM (IRAM) port. Depending on entry conditions, a file may be created in Main RAM, on a specified port, or on the first port found to have enough room for the file. Unlike the entry conditions for FINDF, if no particular device is specified, Main RAM is assumed.

Routine CRTF is a general-purpose utility to create a file either in the mainframe or on an external device. CRTF performs rudimentary initialization of a file depending on its file type, and makes use of CREATF or the HP-IL Module (via polls) depending on the specified device.

The CREATF algorithm for creating a mainframe file is described below.

```
  CREATF Algorithm for Creatung a Mainframe File
  -----------------------------------------------

   Save desired file size in R0;
   If MAIN or no file chain specified
     If not enough memory with LEEWAY check
1:     Load error; return with carry set;
     Open up memory; Write time/date in header;
     Write File Chain Length field;
     Goto RFADJ+;
   If no particular port specified
     Call ROMCHK;
     If no (more) plug-ins
A:     Load error; return with carry set;
     Call B;
     If create done sucessfully on that plug-in
       Return with carry clear;
     Call ROMFND; goto A
   Call ROMF-1;
   If plug-in not found
     THEN goto A;
B: If plug-in is not RAM
     THEN load error; return with carry set
   Calculate amount of available memory on
   plug-in (LSTADR-EOFLCH);
   If not enough room
```

```
  THEN goto 1;
Write creation date/time; write file chain length
Return with carry clear.
```

```
+------------------------------------------+-----------------+
|                                          |                 |
|   TABLE FORMATS                          |  CHAPTER   12   |
|                                          |                 |
+------------------------------------------+-----------------+
```

12.1 ASSIGN Buffer

The ASSIGN buffer (bASSIGN) saves all the open channel #'s. Every entry in the buffer takes 5 nibbles. The information contained is:

Channel # (Device #)	2 nibbles
Code Nibble	1 nibble
FIB# or Indirect Channel #	2 nibbles

Every entry occupies 5 nibbles. The maximum ASSIGN buffer size is 4095 nibbles, so a maximum of 819 channels and stack markers will fit in the ASSIGN buffer.

If the channel # is zero, then the remaining high 3 nibbles are a count of subroutine levels without any channels on that level. The maximum count of SUB levels in the stack marker is 4096 (0-FFF hex).

The assign table is always searched from the end of the buffer to the start of the buffer or a stack marker, whichever occurs first. This implies that all new entries are appended to the buffer.

If the code nibble is non-zero, the search routine should search in the previous stack level for the indirect channel #. These links continue back until either a FIB# is found or an entry with an FIB# of zero is found. A zero FIB# means this channel is no longer open.

12.2 Card Reader Buffer

This is a buffer (bCARD) used by the card reader subsystem for building a copy of the card header being written out or read in to/from card. The format of the information is as follows (all numbers shown in bytes):

```
 0:  sub-format (1): 00 for LIF1 file (HP-75 subformat)
                     01 for HP-71 files (HP-71 subformat)
 1:  track# (1)
 2:  # of tracks in set (1)
 3:  # bytes in this track (2)
 5:  # bytes in file (2)
 7:  file type (2): Kangaroo filetype (HP-75 subformat)
                    LIF filetype number (HP-71 subformat)
 9:  creation date (4): hex seconds since start of
                        century.
 13: file name (8)
 21: password (4): blanks for LIF1 filetype (HP-75 sbfmt)
     implementation (4): (HP-71 subformat)
 25: marker (2): checksum of entire file, including
                 file header.
 27: partial statement status (1)
 28: s1--partial statement size information (2)
 30: s2--partial statement size information (2)
 32: data checksum (2): 2-byte checksum of data field.
 34: header checksum (1): 2-byte sum of header field,
                          folded to one byte without
                          wraparound carry.
 35: (reserved) (1)
```

When a CAT CARD is performed, some additional space is created at the end of the buffer for building a dummy file header. This dummy header is used by the CAT formatting routines to create a catalog entry listing.

12.3 Character Sets

12.3.1 Standard Character Set

The standard character set consists of ASCII characters 0 through 127.

12.3.2 Alternate Character Set Buffer

The alternate character set buffer (bCHARS) has two possible formats. If the alternate set is contained in the buffer itself, the buffer will have an even number of nibbles. If the buffer is merely a pointer to a character set stored elsewhere, then the buffer will have an odd number of nibbles.

In the case of an even length buffer, the contents of the buffer consist of n groups of 6 bytes (where 1 <= n <= 127). Each byte describes one column of one character. Each group of bytes defines one character pattern. The least significant bit of each byte corresponds the top row of the display.

In the case of an odd length buffer the contents of the buffer consist of a 5 nibble absolute address of the actual character set table followed by a byte which uniquely identifies which character set is being pointed to. This byte is used during the configure poll (pCONFG) to help individual ROMs determine if they are responsible for this buffer. The table pointed to should consist of a three nibble length field followed immediately by a table in the same format as would be in the buffer. As long as the buffer contents remain an odd length the buffer may be extended as desired, since it will be ignored when the character set is referenced (ie. it might be used to preserve a previously active character set to be restored later).

This buffer's update count field should contain a 1 if it is an odd length buffer so that the address pointing to the character set table will be updated when memory moves.

When the auto-delete of I/O buffers is performed during CONFiguration, this buffer will be deleted only if it is odd in length and no ROM responds to the poll (pCONFG) by marking the buffer and updating the address pointing to the actual character set table.

12.4 External Command Buffer

The External Command Buffer (bECOMD) may be created by a LEX file during the pDSWNK or pDSWKY poll. It contains BASIC ASCII text in the same format as in the Startup Buffer. The text will be executed on return from Deep Sleep IF Deep Sleep was called from the Power Down routine. A system flag, flPWDN, identifies if deepsleep was called from powerdown, which is useful during the powerup polls.

12.5 File Information Buffer

The FIB is a system buufer maintained by the operating system which contains an entry for each open file. All access to an opened file is controlled by its entry in the FIB, which is identified internally by its entry number. See the "File System" chapter for further information on file access. The format of each FIB entry is as follows.

FILE INFORMATION BUFFER (FIB) ENTRY FORMAT

1. FIB entry number (2 nibs) - If 00, end of FIB

2. File I/O Buffer number (3 nibs) -
   If the file is on external device, it has a 256-byte system buffer associated with it to hold the current sector. If this field is not 000, it is the ID of this associated File I/O buffer.

3. File type (4 nibs) - File type number of the file

4. File protection nibble (1 nibs) - This is the same nibble in the file header

5. File copy code (1 nib) - This is the same value as in the File Type Table entry for this file type

6. Access code (1 nib) -
   This nibble is only useful for files on external devices. It is set to 1 when the current contents of the file I/O buffer has been altered. It is set to zero when a new record has been read into the file I/O buffer.

7. Device type(1 nib) - This nibble indicates where the file is located :

0 - Mainframe	1 - Independent RAM
2 - ROM	8 - HP-IL device

8. File begin (6 nibs) -
   For file in RAM/ROM :

Nibs	Field
4-0	Abs address of file header start
5	Unused

   For file in mass memory device :

Nibs	Field
0	Nth entry in the directory record
4-1	Record number in the directory area

9. Subheader length (2 nibs)
   This length is the number of bytes of the subheader. It is computed as follows:

   Subtract 5 nibbles from the Offset to Data field of the File Type Table entry for that file type. If the copy code is 1 (e.g. DATA) or 8 (user-defined), subtract another 8 nibbles for the Implementation Field, which is present after the file header. Then divide by 2 to convert into bytes:

```
        Copy code 1 or 8:      (Offset to Data - 13) / 2
        Copy code 0, 2, or 4: (Offset to Data -  5) / 2
```

10. File data start (11 nibs) -
    This is the absolute address of the start of data of the file.

    For file in RAM/ROM:

Nibs	Field
4-0	Abs addr of data start (skip over the subheader)
6-5	Hex F0
8-7	Port address: Port #, Extender #
10-9	Unused

    For file on an external (HP-IL) device:

Nibs	Field
3-0	Record addr of the first record of the file. If the file has subheader, the subheader starts from byte 00 of this record.
6-4	Device address
9-7	Assign code
10	Assign type

11. File length(4 nibs) -
    For fixed record length file, this is the file length in number of records.

12. Record length(4 nibs) -
    For fixed record length file, this is the record length in number of bytes.

13. Current position(6 nibs) -
    This is the current file pointer. It is the offset from the file data start.

Nibs	Field
5-0	Offset from file data start in nibbles.

14. File data length(6 nibs) -
    This is the file data length not including the subheader.

Nibs	Field
5-0	File data length in nibbles.

15. Remaining length in current record(5 nibs) -
    This field is used by PRINT# and READ# to keep track of

how many bytes to the end of current record.

16. Device size (6 nibs)
    This field is only useful for file in an external mass memory device(HP-IL)

Nibs	Field
5-0	Number of sectors allocated to this file.

12.5.1 Open Files and Protection

Whatever a file's protection is at the time it is opened, is the access capability that is stored in the FIB. Therefore all subsequent commands which reference the FIB for their operation will be subject to the access capability of the file AT THE TIME IT WAS OPENED. The implication here is that the user may choose to open a file, SECURE it, proceed to do a series of PRINTs to that file, and then close the file. Until the file's protection is changed, all that transpired while the file was open is protected.

12.6 File Type Table

A File Type Table defines the attributes of one or more types of file. Each type of file contains one entry in the table. See the "HP-71 Code Examples" chapter for a listing of the File Type Table that defines the file types recognized by the HP-71 mainframe.

The entry defines the name, file type numbers, and types of protection which are associated with that type of file. The entry also defines the create code and copy code for the file, which describes in a very general way the structure of the file. These codes determine the presence or absence of an implementation field following the file header when the file is in memory, and determine whether the mainframe can copy the file into or out of the HP-71 without the aid of a LEX file.

The File Type Table terminates with an FF byte.

FILE TYPE TABLE FORMAT

Field	Size (nibs)	Meaning
Create code	1	0: Normal mainframe file structure (BASIC, BIN, LEX, KEY, etc) File length measured in nibs, arbitrary format, subheaders

		allowed 1: DATA file structure; up to 65535 fixed length records of up to 65535 bytes each; subheaders not allowed; file is initialized to FF's 2: SDATA file structure; records are fixed length, 8 bytes each; file initialized to zeros; subheaders not allowed 4: TEXT file structure; records are variable number of bytes; file initialized to FF's; subheaders are not allowed 8: Special handler routine required to create this file; system will issue pCRT=8 poll
Copy code	1	0: Normal mainframe file structure; File can be copied into or out of HP-71 without aid from LEX file; Implementation Field contains file length on external copy, but is not present after file header when file is in memory 1: DATA file structure; file can be copied into or out of HP-71 with no aid from LEX file; on external copy, the Implementation Field contains number of records and record length, and it is present immediately after file header when file is in memory 2: SDATA file structure; file can be copied into or out of HP-71 with no aid from LEX file; on external copy, the Implementation Field contains number of records, but is not present after file header when file is in memory 4: TEXT file structure; file can be copied into or out of HP-71 with no aid from LEX file; on external copy, the Implementation Field is zero, and it is not present after file header when file is in memory 8: Special copy routine is required to copy file to or from HP-71; system will issue pUCRD8 poll; Implementation Field is present

		after file header when file is in memory
Execution code	1	1: File is executable (can be run) 0: File is not executable
Offset to Data	2	Offset in nibs from start of file chain length field (in file header) to start of file data, skipping the Implementation Field, if present after file header (see Copy code, above), and also skipping the subheader (if any); this value is used to calculate the subheader length, when present
File type name	10	5 character ASCII name of the file type as displayed by CAT; padded with trailing blanks
Number of types	1	Number of file type numbers used by this type of file to indicate SECURE or PRIVATE states, if allowed; up to four type numbers may be used:

Type Number	Protection Indicated
First	No protection
Second	SECURE
Third	PRIVATE
Fourth	SECURE and PRIVATE

LIF type numbers	4	4 nibbles for each LIF type number indicated by the previous field

12.7 Keycode Table

The mainframe contains a table (KEYCOD) which specifies the default meaning of each key on the keyboard. This table is arranged as three sets of 56 bytes. The first set describes the unshifted function of the keys. The second set describes the f-shifted function of the keys. The third set describes the g-shifted function of the keys. Within each set, the keys are in the order QWERTY ... 0.,+ which is the same order as used in DEF KEY.

The byte in the table specifies the meaning of the key as follows:

0 - 31	Special code
32 - 127	ASCII letter with same code
128 - 255	Typing aid key of keyword with same code

The codes in range 0 - 31 are for keys that do not have a simple ASCII letter or a typing aid associated with them. Such keys require special processing.

Code	Symbol	Function
0	kc-CHR	Delete char
1	kcLC	Lowercase toggle
2	kcI/R	Insert/Replace toggle
3	kcUSER	User mode toggle
4	kc-LIN	Delete through EOL
5	kcFLFT	Cursor far left
6	kcFRT	Cursor far right
7	kcBKSP	Backspace
8	kcLFT	Cursor left
9	kcRT	Cursor right
10	kcCTRL	CTRL prefix
11	kcVIEW	VIEW prefix
12	kcUSEX	1USER
13	kcEOL	Endline
14	kcATTN	ATTN
15	kcRUN	RUN
16	kcCONT	CONT
17	kcSST	SST
18	kcUP	Up
19	kcDOWN	Down
20	kcTOP	Top
21	kcBOT	Bottom
22	kcGON	g-ON
23	kcCALC	CALC
24	kcOFF	OFF
25	kcLAST	Command stack
26	kcLERR	Last error message
27		Reserved
28		Reserved
29		Reserved
30		Reserved
31		Reserved

12.8 Language Tables

12.8.1 MAINT and XROM01

Several mainframe tables are used by the lexical analyzer, parse, execution and decompile drivers. These tables are used to identify BASIC keywords, functions and system commands and assign a unique internal token number. These tables comprise a mainframe LEX file, following the format of LEX files, as explained under LEX files in the "Language Extension and Binary Files" chapter. This table is called: MAINT.

Due to the large number of built-in keywords, one internal LEX "file" is not large enough. A second internal LEX File: XROM01 with LEX ID #01, holds the overflow of built-in keywords. Both LEX "files" contain a SPEED table, main table, and text table. The internal LEX "files" reside within system ROM and are not part of the RAM file chain.

Keywords contained in XROM01 are less frequently used keywords or keywords that are not programmable. The tokenized length of XROM01 keywords is 2 bytes longer than keywords contained in MAINT.

12.8.2 Message Table

Details on message table construction are found in the "Message Handling" chapter.

12.8.3 Lexical Type Table

The lexical type table (LXTYPT) describes a character type and ASCII or internal representation (token) for each character.

For each character in the ASCII range, 20-7E, is an entry:

```
TYPE      - Categorize character
              0 - Miscellaneous
              1 - Digit, Decimal Point
              2 - Letter A - Z
              3 - Relational Character < = > ?
            1 nibble

CHARACTER - ASCII representation or internal token or character
            2 nibbles, backwards
```

This table resides in mainframe ONLY.

12.8.4 FG Table

The "FG Table" defines a state machine for processing f and g shifts. The state machine has 7 input bits and 4 output bits. The seven input bits are as follows:

- Bit 6 F key currently down
- Bit 5 G key currently down
- Bit 4 Some non-FG key newly down (X)
- Bit 3 g annunciator on
- Bit 2 f annunciator on
- Bit 1 Ghost bit (*)
- Bit 0 F or G key was down during last key scan (h)

The ghost bit is used to indicate that an f or g shift has been performed but the annunciator was left on because the corresponding key was still down.

The lower 4 bits are stored between key scans in the display RAM nibble that contains the f and g annunciators. The lower two bits do not affect the display since there are no annunciators in the LCD to correspond to these bits. These 7 bits form an offset into the table which gives the new "state" of the state machine and is stored back into display memory. If bit 4 is set but bits 5 and 6 are clear then all bits should be cleared following putting the f or g modified key codes in the buffer.

FGX	gf*h 0000	gf*h 0001	gf*h 0010	gf*h 0011	gf*h 0100	gf*h 0101	gf*h 0110	gf*h 0111	gf*h 1000	gf*h 1001	gf*h 1010	gf*h 1011	gf*h 1100
000	0000	0000	....	....	0100	0100	0000	0000	1000	1000	0000	0000	....
001	0000	0000	....	....	0100	0100	0100	0100	1000	1000	1000	1000	....
010	1001	0001	....	....	1001	1001	1001	1001	0001	1001	1011	1011	....
011	1011	1011	....	....	1011	1011	1011	1011	1011	1011	1011	1011	....
100	0101	0001	....	....	0001	0101	0111	0111	0101	0101	0101	0101	....
101	0111	0111	....	....	0111	0111	0111	0111	0111	0111	0111	0111	....
110	0000	0000	....	....	0000	0000	0000	0000	0000	0000	0000	0000	....
111	0000	0000	....	....	0000	0000	0000	0000	0000	0000	0000	0000	....

12.9 LEX Entry Buffer

The LEX entry buffer (bLEX) resides in the main system RAM following the statement buffer. It is a list of pointers to language extension files in RAM and ROM and to the 2 built-in language "files" within the mainframe. Associated with each LEX file is the range of entry numbers within the LEX file.

For each LEX file entry:

ID#	Low Entry#	High Entry#	LEX File MAIN Table Start	
2	2	2	5	nibs

The range of LEX IDs is 0 to 255. The range of entry numbers within a LEX file is 0 to 255. Several LEX files may have the same LEX ID, but different ranges of entry numbers. There is a separate entry for each separate LEX file.

The LEX entry buffer is recreated every time the configuration of the machine may change or a LEX file is added or removed. This includes coldstart, warmstart, power on, module pulled, CLAIM, COPY, FREE and PURGE.

12.9.1 Search Order of LEX Files

First, main memory is searched for LEX type files. The standard file header is skipped and the LEX ID# and entry# range is read. The main table address is calculated, based on the presence or absence of the optional speed table. The LEX ID#, entry# range and main table start address are added to the LEX entry buffer. LEX files may be chained together internally. All LEX files within one system file are added to the table.

ROMs and independent RAMs are searched next. For each configured ROM/IRAM, the entire file chain is searched. Each language extension file within the ROMs file chain is added to the LEX entry buffer.

Plug-ins are searched in port-device order; i.e. Port 0 through Port 5, with internal devices within each port searched in order.

The two final buffer entries are the built-in XROM and the mainframe main table. The built-in XROM LEX ID# is 01, with a token range of 0 to 95. The mainframe LEX ID# is 00, with a token range of 0 to 255. LEX ID#00 is useful for detecting the end of the buffer when searching for a particular external keyword or function.

12.9.2 Usage

The LEX entry buffer is used by the lexical analyzer when scanning for valid keywords, functions and commands. This allows the BASIC language and system command set to be extended and overridden.

This buffer is also used determine addresses to decompile and execute external keywords and functions, and display external error messages.

12.10 Startup and Immediate Execute Key Buffers

These two buffers are used to hold a string of characters which will later be parsed and executed. The STARTUP buffer (bSTART) is set up by the STARTUP command and is parsed and executed when the user turns the machine on with the ON/Attention key. The immediate execute key buffer (bIEXKY) is created whenever a colon-type key definition is executed. This buffer is automatically deleted at MAINLP since it no longer has meaning at that point.

The string stored in these buffers is always terminated with a CR (ASCII 13). This is required since the buffers will be parsed.

12.11 Statistic Buffer

The Statistic Buffer is a scratch buffer used during summary statistics accumulation in ADD and DROP.

12.12 System Flags

A flag is a variable that can have one of only two possible states, set and clear. The numeric values 1 and 0 are assigned to these states, respectively. Flags are generally used to control the flow of a program and to record the status of certain modes. Flags are global variables; flag settings remain in effect before during, and after subprogam execution.

There are 64 system flags (numbered -64 to -1) and 64 user flags (numbered 0 to 63). These flags are stored in 128 consecutive bits starting at address SYSFLG. (See the diagram in the Memory

Structure description.)

The following table summarizes the system flag assignments, from low to high memory.

Flag Name	Flag #	Set/Clear by user?	Cold-start Status
Quiet Mode	-1	Yes	Clear
Beep On	-2	Yes	Clear
Continuous On	-3	Yes	Clear
Inexact Result	-4	Yes	Clear
Underflow	-5	Yes	Clear
Overflow	-6	Yes	Clear
Divide-by-Zero	-7	Yes	Clear
Invalid Operation	-8	Yes	Clear
User Mode	-9	Yes	Clear
RAD trig Mode	-10	Yes	Clear
Rounding Mode (POS/NEG)	-11	Yes	Clear
Rounding Mode (ZERO/NEG)	-12	Yes	Clear
Display Mode (FIX/ENG)	-13	Yes	Clear
Display Mode (SCI/ENG)	-14	Yes	Clear
Lower Case	-15	Yes	Clear
Base Option	-16	Yes	Clear
Display digit	-17 to -20	Yes	Clear
Reserved for HPIL	-21 to -24	Yes	Clear
BEEP loud	-25	Yes	Clear
Don't prompt	-26	Yes	Clear
Unassigned	-27 to -32	Yes	Clear
Unassigned	-33 to -42	No	Clear
Machine is dormant	-43	No	Clear
Always return from MEMERR	-44	No	Clear
Clock Mode (1 sec update)	-45	No	Clear
EXACT flag	-46	No	Clear
Command Stack Active	-47	No	Clear
Control Key Hit	-48	No	Clear
DSLEEP from PWR down	-49	No	Clear
Req set TRNOF in MAINLP	-50	No	Clear
Turnoff at MAINLP	-51	No	Clear
VIEW key pressed	-52	No	Clear
Reserved for HPIL	-53 to -56	No	Clear
AC Annunciator	-57	No	N/A
User Mode Susp	-58	No	Clear
Key repeated	-59	No	Clear
Alarm Annunciator	-60	No	Clear
Low Battery Annunciator	-61	No	N/A
Program Annunciator	-62	No	Clear
Suspend Annunciator	-63	No	Clear
CALC Mode Annunciator	-64	No	Clear

User flags	0-63	Yes	Clear

12.12.1 Display Format Information

Display format information is contained in the system flags. System flags -13,-14 indicate the current display mode:

0 = STD	2 = SCI
1 = FIX	3 = ENG

System flags -17 through -20 contain the number of digits in hexadecimal for the current display setting, i.e. "n" in FIX n, SCI n, ENG n.

12.13 Traps

As used here, a trap is a one-nibble numeric code (0 to F in hex) which determines what action, out of a menu of possible actions, is to be taken when a corresponding flag is set by the system (i.e. by other than an SFLAG statement or FLAG function).

Traps are implemented only for the arithmetic exception flags (inexact, underflow, overflow, divide-by-zero, invalid operation). Associated with each of these five exceptions is an action (or trap) to be taken whenever that exception occurs. There are, at present, three categories of trap actions and they are denoted by trap values 0, 1 and 2. The trap values and their associated trap actions are given below:

TRAP MENU

Trap Value	Trap Action
0	Halts and displays an error message
1	Returns the traditional finite default values
2	Returns the default values specified by the IEEE Standard (see the Default Values Table)

DEFAULT VALUES TABLE

EXCEPTION	TRAP VALUE 1	TRAP VALUE 2
INX	rounded 12 digit result	rounded 12 digit result
UNF	0	denormalized result
OVF	+-maxreal (9.99999999999e499)	+-infinity
DVZ	+-maxreal	+-infinity
IVL	0^0=1, else halt	0^0=1,else NaN

If present, an ON ERROR statement overrides the trap values for all exceptions except INX. OFF ERROR will return control back to the trap action settings. The DEFAULT statement sets the traps as follows:

OFF — trap values for UNF,OVF,DVZ,IVL set to 0
trap value for INX set to 1 (same as 2)

ON — trap values for INX,UNF,OVF,DVZ,IVL set to 1

EXTEND — trap values for INX,UNF,OVF,DVZ set to 2
trap value for IVL set to 1

```
+-------------------------------------------+-----------------+
|                                           |                 |
|   INTERNAL DATA REPRESENTATION            |   CHAPTER  13   |
|                                           |                 |
+-------------------------------------------+-----------------+
```

This chapter discusses the format in which the HP-71 represents numeric or string data in memory or in the CPU registers.

13.1 Data Types

The HP-71 supports seven data types. The data type of a variable is identified by looking at the variable register (explained in the section on variable chains, below). Real scalar numbers are stored directly into the variable register, and can be identified by the low-order nibble, which falls in the range 0-9 (and is interpreted as the low-order nibble of the exponent field). If the low-order nibble of a variable register is anything else, the register is serving as a pointer to one of the other six data types:

```
A   Integer         (simple or array)
B   Real short      (simple or array)
C   Real array
D   Complex Short   (simple or array)
E   Complex         (simple or array)
F   String          (simple or array)
```

13.2 Registers

The following section will discuss the representation of variables in memory. This section contains an introduction to the representation of numbers in the CPU; that topic is treated more thoroughly in the section on mathematical operands in this chapter.

13.2.1 Numbers in CPU Registers

When a number is brought into a CPU register, the process of recalling it (finding it in memory and bringing it into the CPU by way of the mathstack) will convert it into one (for real data types) or two (for complex data types) numbers in a standard representation as follows:

```
15                                      0
+-+-------------------------------+-----+
|S|  Mantissa                     | Exp |
+-+-------------------------------+-----+
 1          12                       3        16 nibbles
```

with the low-order digits in the low-order nibbles of the register:

```
15                                      0
+-+------------------------------+------+
|S| M11 . . . . . . . . . . . M0 |E2..E0|
+-+------------------------------+------+
 1          12                      3         16 nibbles
```

The mantissa is unsigned with a separate field (the S-field) representing the sign (0 = +, 9 = -). The Exponent is represented in 10's complement form. This representation is the normal entry condition for all routines which expect a floating-point argument(s) in the 12-digit form.

Many of the computation algorithms work with a 15-digit form so that intermediate results can be computed and retained with greater accuracy. Typically, when implementing a function, you will take the parameters (which are in 12-digit form), expand them into the 15-digit form, call whatever computation routines are necessary, and round the 15-digit result back into 12 digits.

The 15-digit form occupies two registers as follows:

```
15                                      0
+-+-------------------------------------+
| |  Mantissa                           |
+-+-------------------------------------+
 1                15                          16 nibbles

15                                      0
+-+--------------------------+----------+
|S|                          |   Exp    |
+-+--------------------------+----------+
 1          10                   5            16 nibbles
```

where the exponent has been extended (including sign-extend if negative) to five digits.

The published entry and exit conditions for various numerical algorithms state what registers are expected to contain which parts of the argument(s).

13.2.2 Strings in CPU Registers

Unlike numbers, the actual values of strings are not usually recalled into CPU registers - they generally don't fit. The procedure for accessing a string is to place the string on the mathstack by evaluating an expression and then to "pop" its descriptor (mainframe POP1S routine), which provides a pointer and a character count.

13.3 Variables

Every variable has a corresponding variable register in which the value is stored (for simple variable types) or a reference to the value is stored (for complex, arrays and strings). Variables (i.e., their corresponding registers) can be created either explicitly (DIM, INTEGER, REAL statements) or implicitly (by storing into a non-existent variable or array element) and will continue to exist until wiped out by a DESTROY <var name> or DESTROY ALL.

For speed of reference, each variable register is contained in one of 26 lists ("chains"); the alphabetic part of the variable name determines which chain. Operating system RAM contains a list of pointers to the various chains, the format of which is described below.

13.3.1 Variable Chains

There are 26 reserved pointers to the variable chains A-Z. This list of 7 nibble pointers begins at 2F5BE. The first 2 nibbles indicate the number of variables in the variable chain. The next 5 nibbles (Chain Head Pointer) give the absolute address of the first variable in the chain.

```
             [2F5BE]
         Reserved Pointers                      A-Variable Chain

         2        5                             3         16
       +---+--------------+                   +-------+---------------+
A      | # | addr to 1st  |------------------>| label | Var register  |
       +---+--------------+                   +-------+---------------+
B      | # | addr to 1st  |                   |       |               |
       +---+--------------+                   +-------+---------------+
C      | # | addr to 1st  |                   |       |               |
       +---+--------------+                   +-------+---------------+
.      |          |       |
.      |          |       |
.      |          v       |
       +---+--------------+
Z      | # | addr to 1st  |
       +---+--------------+
```

The A-variable chain will contain the variable registers for all variables whose names begin with A (A, A7, A$, A5$, etc.). Variables in each chain are listed in the order in which they were created; the chain is not sorted in any way.

A particular variable chain contains a 19 nibble entry for each variable beginning with that letter. The first 3 nibbles are the variable label, and the next 16 are the variable register.

The first two nibbles of the label field are the ASCII code for the letter associated with the variable. An uppercase letter indicates a numeric variable, lowercase a string variable. That is, an AND of the ASCII code and the constant 20H will produce 0 for a numeric variable and nonzero for a string variable. The third nibble is the digit+1 of an alpha-digit variable, 0 for alpha variables.

The data space for variables is allocated, as required, between the RAM pointers ACTIVE and CALSTK.

In the discussion below it is important to keep in mind that when memory is read into a register, the CPU places the lowest addressed nibble in the least significant nibble of the register. Thus, in the diagrams below, the nibbles lowest in memory are shown on the right side of the register. The nibbles in the register are numbered from 15 to 0 going most to least significant.

If a recall is attempted on a non-existent variable, a value of zero is returned if the variable is numeric and null if the variable is string. The variable is not created at this time.

13.3.2 Variable Internal Representation

Nibble 0 of the variable register is the Data Type nibble and contains the data type code, except for indirect variables (see below). The following information is encoded in nibble 0:

0-9	Real scalar	(default)
A	Integer	(simple or array)
B	Real short	(simple or array)
C	Real array	
D	Complex Short	(simple or array)
E	Complex	(simple or array)
F	String	(simple or array)

If the variable is default type (Real scalar), the variable register contains the actual value of the variable, and nibble 0 is the low-order digit of the exponent. In all other cases, the nibbles in the variable register mean the following:

Nibble 1 indicates the dimension. If the variable is a scalar, its dimension is 0. If the variable is an array, the dimension is either 1 or 2. The dimension of string arrays must be 1. A value of F or E in this nibble identifies an indirect variable, explained below.

The meaning of the remaining nibbles depends on whether the variable is scalar (that is, a simple variable), array or string.

13.3.2.1 Scalar Numeric Variables

A scalar variable is a simple variable, as opposed to an array. For scalar variables of type integer and real short, the value of the variable is contained in part of the variable register. For scalar complex variables, nibbles 11-15 are a relative pointer to the data. The exact representation is illustrated graphically below.

```
                15                 2 1 0
                +-+-----------------+-----+
    Real        |S|    Mantissa     | Exp |
                +-+-----------------+-----+
                 1       12           3

                15  10    7          2 1 0
    Real        +---+-----+-+---------+-+-+
    Short       |///| Exp |S| Mantissa|0|B|
                +---+-----+-+---------+-+-+
                  5   3   1     5      1 1
```

```
                 15       7              2 1 0
                 +--------+-+-------------+-+-+
    Integer      |////////|S|  Mantissa   |0|A|
                 +--------+-+-------------+-+-+
                     8    1       5        1 1

                 S field of Integer is packed:

                     Bits 0-2     Exponent if less than 6.
                                  Value of 6 means Inf.
                                  Value of 7 means NaN.
                     Bits 3       Sign.


                 15     11               2 1 0
                 +-------+---------------+-+-+
    Complex   +--|Pointer|///////////////|0|E|
              |  +-------+---------------+-+-+
              |      5                    1 1
              |
              +--------------------------------+
                                               |
                 +-+------------------+------+ |
                 |S| Real Pt Mantissa | Exp  |<-+
                 +-+------------------+------+
                 |S| Imag Pt Mantissa | Exp  |
                 +-+------------------+------+
                  1         12           3

                 15     11               2 1 0
    Complex      +-------+---------------+-+-+
    Short       +-|Pointer|///////////////|0|D|
                |  +-------+---------------+-+-+
                |      5                    1 1
                +--------------------------------+
                                                 |
                          +-----+-+-------------+ |
                          | Exp |S| Real Mant   |<-+
                          +-----+-+-------------+
                          | Exp |S| Imag Mant   |
                          +-----+-+-------------+
                             3   1       5
```

13.3.2.2 Numeric Arrays

For arrays, the information contained in the variable register is referred to as the "dope vector".

Nibble 2 of the variable register indicates the Base Option of 0 or 1. If this variable is the current STAT array, the high bit of nibble 2 is set. A STAT array always has Base 0 and a different format for the remaining information in the variable register (see STAT array discussion below).

The next 8 nibbles give the limits of each dimension, where the first 4 are the second dimension (meaningless if one-dimensional) and the next 4 are the first dimension. The limit in each dimension is 65535.

Nibbles 11-15 are a relative pointer to the start of the array data. To calculate the actual data address subtract the relative pointer value from the address of the relative pointer. The subroutine RECADR is useful for this calculation.

```
                15    11            2 1 0
   Real         +-------+-----------+-+-+-+
   Array      +-|Pointer| Dim Limits|b|#|C|
              | +-------+-----------+-+-+-+
              |    5          8      1 1 1
              +-----------------------------+
                                            |
                +-+-----------------+-----+ |
                |S|    Mantissa     | Exp |<-+
                +-+-----------------+-----+
                |S|    Mantissa     | Exp |
                +-+-----------------+-----+
                |S|    Mantissa     | Exp |
                +-+-----------------+-----+
                 1        12   .      3
                               .
                               .

                15    11            2 1 0
                +-------+-----------+-+-+-+
   Real       +-|Pointer| Dim Limits|b|#|B|
   Short      | +-------+-----------+-+-+-+
   Array      |    5          8      1 1 1
              |
              +-----------------------------+
                                            |
                      +-----+-+-----------+ |
                      | Exp |S| Mantissa  |<-+
                      +-----+-+-----------+
                      | Exp |S| Mantissa  |
                      +-----+-+-----------+
                      | Exp |S| Mantissa  |
                      +-----+-+-----------+
                         3   1   .   5
                                 .
                                 .
```

```
                 15    11              2 1 0
     Integer     +--------------------+-+-+-+
     Array     +-|Pointer| Dim Limits |b|#|A|
               | +--------------------+-+-+-+
               |    5           8      1 1 1
               |
               +-------------------------------+
                                               |
                             +-+------------+  |
                             |S|  Mantissa  |<-+
                             +-+------------+
                             |S|  Mantissa  |
                             +-+------------+
                             |S|  Mantissa  |
                             +-+------------+
                              1   .  5
                                  .
                                  .

                 15    11              2 1 0
     Complex     +-------+------------+-+-+-+
     Short     +-|Pointer| Dim Limits |b|#|D|
     Array     | +-------+------------+-+-+-+
               |    5          8       1 1 1
               +-------------------------------+
                                               |
                      +-----+-+-------------+  |
                      | Exp |S| Re Mant     |<-+
                      +-----+-+-------------+
                      | Exp |S| Im Mant     |
                      +-----+-+-------------+
                      | Exp |S| Re Mant     |
                      +-----+-+-------------+
                      | Exp |S| Im Mant     |
                      +-----+-+-------------+
                      | Exp |S| Re Mant     |
                      +-----+-+-------------+
                      | Exp |S| Im Mant     |
                      +-----+-+-------------+
                         3   1   .  5
                                 .
                                 .
```

```
                 15    11               2 1 0
    Complex      +-------+---------------+-+-+-+
    Array     +--|Pointer| Dim Limits    |b|#|E|
              |  +-------+---------------+-+-+-+
              |     5          8          1 1 1
              |
              +--------------------------------+
                                               |
                 +-+-------------------+-----+ |
                 |S| Real Pt Mantissa  | Exp |<-+
                 +-+-------------------+-----+
                 |S| Imag Pt Mantissa  | Exp |
                 +-+-------------------+-----+
                 |S| Real Pt Mantissa  | Exp |
                 +-+-------------------+-----+
                 |S| Imag Pt Mantissa  | Exp |
                 +-+-------------------+-----+
                 |S| Real Pt Mantissa  | Exp |
                 +-+-------------------+-----+
                 |S| Imag Pt Mantissa  | Exp |
                 +-+-------------------+-----+
                  1        12 .          3
                              .
                              .
```

13.3.2.3 Statistical (STAT) Array

A statistical array is a specialized one-dimensional real array used to accumulate and store summary statistics for a data set. It is set up by the STAT statement. The chapter on "Numeric Computation Algorithms" discusses the elements of a statistical array and their meaning.

A statistical array has base option 0 and the high bit of nibble 2 is set. Nibble 2 therefore has the value 8.

Nibble 3 gives the number of variables in the data set represented by the statistical array. If a linear regression model has been specified by the LR statement, nibbles 4 and 5 give, respectively, the independent and dependent variable numbers. Otherwise, these nibbles have value zero. The maximum value for each of nibbles 3-5 is 15.

Nibble 6 is not used.

The next 4 nibbles give the dimension limit of the statistical array. The maximum value is 65535, although the STAT statement will not dimension a statistical array to a dimension greater than 135.

Nibbles 11-15 are a relative pointer to the start of the array data.

```
                 15    11 10   7 6 5 4 3 2 1 0
  Statistical    +-------+------+-+-+-+-+-+-+-+
  Array        +-|Pointer|Dim Lm|  |D|I|V|8|1|C|
               | +-------+------+-+-+-+-+-+-+-+
               |    5        4    1 1 1 1 1 1 1
               +----------------------------+
                                            |
                 +-+-----------------+-----+ |
                 |S|    Mantissa     | Exp |<-+
                 +-+-----------------+-----+
                 |S|    Mantissa     | Exp |
                 +-+-----------------+-----+
                 |S|    Mantissa     | Exp |
                 +-+-----------------+-----+
                  1        12  .        3
                               .
                               .
```

13.3.2.4 String Variables

String variables are allocated when dimensioned or assigned during program execution. The data type is F. The variable register always contains a pointer to the string contents. A value of "null" is returned whenever a nonexistent string variable is referenced.

The value of nibble 1 in the variable register indicates whether the string variable is scalar (value 0) or an array (value 1). Note that string arrays can have only one dimension.

As with numeric arrays, nibble 2 indicates the base option for a string array. This number is meaningless for scalar strings (where nibble 1 is 0).

Nibbles 3-6 contain the maximum string length. Maximum value is 65535.

Nibbles 7-10 contain the string dimension limit (meaningless if not an array). Maximum value is 65535.

Nibbles 11-15 are a relative pointer to the start of the string (or string array) data.

```
                 15    11     7       2 1 0
                 +-------+-----+------+-+-+-+
String         +-|Pointer|/////|Maxlen|/|0|F|
               | +-------+-----+------+-+-+-+
               |     5       4     4    1 1 1
               |
               +----------------------------+
                                            |
                                    +-----+ |
                                    |  S  |<-+
                                    |  t  |
                                    |  r  |
                                    |  i  |
                                    |  n  |
                                    |  g  |
                                    +-----+

                 15    11     7       2 1 0
String           +-------+-----+------+-+-+-+
Array          +-|Pointer| Dim |Maxlen|b|1|F|
               | +-------+-----+------+-+-+-+
               |     5       4     4    1 1 1
               |
               +----------------------------+
                                            |
                                    +-----+ |
                                    |  S  |<-+
                                    |  t  |
                                    |  r  |
                                    |  i  |
                                    |  n  |
                                    |  g  |
                                    +-----+
                                    |  S  |
                                    |  t  |
                                    |  r  |
                                    |  i  |
                                    |  n  |
                                    |  g  |
                                    +-----+
                                    |  S  |
                                    |  t  |
                                    |  r  |
                                    |  i  |
                                    |  n  |
                                    |  g  |
                                    +-----+
                                       .
                                       .
                                       .
```

13.3.3 Indirect Variables

Indirect variables are used for parameter passing in subprograms. The data register for the variable is used as an indirect address to the actual variable. Note that if a variable which has been passed to a subprogram is itself an indirect variable, the new pointer will not be linked indirectly through that variable but will point to the originally allocated variable register.

```
                         +------------------+
                         |                  |
15           7 6         |    2 1 0         V
+------------+----------+-+-+          +---------------+
|////////////| Address  |F|x|          | Data          |
+------------+----------+-+-+          +---------------+
      9            5     1 1
```

If nibble 1 is F, it indicates that nibbles 2-6 are an absolute pointer to the data of the variable. Nibble 0 is the data type:

```
A -- Integer
B -- Real Short
C -- Real
D -- Complex Short
E -- Complex
F -- String
```

For a string variable, nibble 0 is F and nibble 1 is F. The address field is the absolute address of the string. The maximum length of the string is kept in nibbles 7-10.

```
                         +------------------+
                         |                  |
15    10      7 6        |    2 1 0         V
+-----+-------+----------+-+-+          +-----------------+
|     |Max Len| Address  |F|F|          | String          |
+-----+-------+----------+-+-+          +-----------------+
         4         5      1 1
```

For an array, nibble 1 is E and nibbles 2-6 hold an absolute pointer to the descriptor for the array. Nibble 0 is meaningless. To find the type of the array, it is necessary to follow the pointer to the variable register and look at the type specified there.

```
                       +-------------------+
                       |                   |
15            7 6      |    2 1 0          V
+-------------+------------+-+-+      +-------------------+
|/////////////| Address    |E|?|      | Variable register |
+-------------+------------+-+-+      +-------------------+
      9            5        1 1
```

13.3.4 Accessing Variables from Binary Programs

13.3.4.1 Finding the Address of a Variable

The ADDRSS (and ADRS40) utility is useful for seaching through a variable chain to find the address of a variable. This is a low level utility that does not handle special cases such as indirect variables. If the variable is not found it merely indicates that condition.

13.3.4.2 Recalling a Variable

The following procedure can be employed to recall variables by name:

Have in memory a stream in the form of a tokenized
  variable followed by a comma or EOL token.
Point D0 at this stream.
Call EXPEXC to evaluate this expression.
Pop the value off the math stack.

This procedure will return a value of zero for non-existent numeric variables and null string for non-existent string variables.

13.3.4.3 Storing into a Variable

The following procedure can be employed to store into variables by name: Have in memory a stream in the form of a tokenized variable followed by a comma or EOL token. For example: A$ EOL is tokenized D2 14 0F and Q9(1,2) EOL is tokenized 13 23 D7 96 15 2 0F. Point D0 at this stream. Call EXPEXC to evaluate this expression. Call DEST to save the address to store into in statement scratch. Get the value to be stored on the math stack by evaluating an expression or by other means. Call STORE to store this value into the variable. This will create the variable if necessary. If calling STORE from a binary program, be sure to zero S-R1-2 to prevent tracing.

13.3.4.4 Creating Variables and Arrays

The methods described above are a way of accessing variables without dealing with any messy problems such as what if the variable you are recalling or storing into doesn't exist. The drawback of these methods is that no control is possible in selecting nondefault attributes for the variables/arrays when they are created -- strings are 32 characters, arrays have an upper bound of 10 in any dimension and numeric variables are of type real. If sizes or types other than these defaults are required, the assembly programmer must explicitly create them. The following procedure will do this.

Set S-R1-3 to the data type (not necessary if string)

A = Integer
B = Real Short
C = Real
D = Complex Short
E = Complex

Point D0 at a token stream in the format of a DIM statement. The following are examples:

Description	Tokenization
A EOL	14 0F
A(2) EOL	D7 14 23 0F
B8(3,4) EOL	D7 86 24 33 43 0F
A$(6) EOL	D7 D2 14 63 0F
B$[80] EOL	D2 24 2F C008 0F
C$(6)[80] EOL	D7 D2 34 63 2F C008 0F

```
        GOSBVL =PREP
        GOSBVL =DPVCTR
        R1=C
*        Iff creating a COMPLEX or COMPLEX SHORT variable
*          then set status bit 0 (ST=1 0).
        GOSBVL =SPACE
        GOSBVL =DMNSN
```

13.3.4.5 Destroying Variables and Arrays

The following method can be used by the assembly language programmer to destroy variables and arrays:

Point D0 at the tokenized stream for the desired variable.
Call DSTRY* to destroy that variable.

13.4 Mathematical Operands

Floating point arguments sent to the math routines from the system generally come off the stack. They are the 12-digit numbers that are visible to the external world and are referred to synonymously in the documentation as 12-forms or packed numbers. This is to distinguish them from the 15-digit numbers used internally in the math routines.

In order to deliver accurate final results to the system the math routines do intermediate calculations with 15-digit mantissas and 5-digit exponents. These internal values are referred to as 15-forms or unpacked numbers.

A typical procedure for a math operation is to pop the 12-form argument from the stack into CPU register A, then call SPLITA to unpack the number into registers A and B. Now the math routine will use this 15-form input to obtain a 15-form result. This accurate 15-form result can then serve as an input to another math routine, or as a final result to be packed by uRES12 into an external 12-form for the system.

13.4.1 Packed Representation (12-form)

13.4.1.1 Normal Values

Let x stand for a floating point value with sign S, 12-digit mantissa MMM...M, and 3-digit exponent EEE . Then x is represented as

```
 1     12     3
+-+-----------+---+
|S|MMM ... M|EEE|
+-+-----------+---+
```

where

```
    / 0    for "+"
S = <                ,
    \ 9    for "-"
```

MM...M = the 12-digit BCD mantissa with implied decimal point after the leading digit, (i.e. 1<= mant <10) and

EEE = the 3-digit BCD exponent in 10's complement notation. (+0 and -0 are represented with an exponent of 0)

Examples

```
                +-+------------+---+
   1) +10  -->  |0|100000000000|001|
                +-+------------+---+

                +-+------------+---+
   2) -.21 -->  |9|210000000000|999|
                +-+------------+---+

                +-+------------+---+
   3) -0   -->  |9|000000000000|000|
                +-+------------+---+
```

13.4.1.2 Extended Values

The values Inf and NaN are distinguished by the hexadecimal digit F in the XS field of the register. Denormalized 12-forms are allowed, but must have an exponent of -499.

Examples

```
                         +-+------------+---+
  1) 0.0051E-499  -->    |0|000510000000|501|
                         +-+------------+---+

                         +-+------------+---+
  2)    -Inf      -->    |9| unspecified|F00|
                         +-+------------+---+

                         +-+------------+---+
  3)     NaN      -->    |S|????????????|F##|
                         +-+------------+---+

                            where ## is non-zero.

                                 +-+------------+---+
   a) HP-71 Quiet NaN  -->       |S|mmmm00000000|F01|
                                 +-+------------+---+

                                    where m=msg#
```

```
                              +-+------------+---+
    b) HP-71 Sig. NaN   -->   |S|000000000000|F02|
                              +-+------------+---+
```

13.4.2 Unpacked representation (15-form)

For greater precision during calculations, 12-digit numeric parameter values (called "12-forms") are expanded or "unpacked" into a form that has a 15-digit mantissa and a 5-digit exponent field. This form is called a "15-form."

A 15-digit form is represented in the CPU as a register pair (A,B) or (C,D). For example, the pair (A,B) has the following format:

```
        +-+----------+-----+
    A = |S|??????????|EEEEE|
        +-+----------+-----+
    B = |?|MMMMMMMMMMMMMMM|
        +-+----------------+
```

where E is a 5-digit 10's complement exponent and B contains a 15-digit mantissa.

The HP-71 math routines assume that unpacked numbers are normalized (denormalized 12-forms are normalized by routine SPLITA). The exceptional values Inf and NaN are indicated by the exponent field alone. Ordinarily, all five digits in the exponent field are BCD. However, if nibble 2 (the XS field) is F, then the number is Inf or NaN; nibbles 0 and 1 then distinguish between Inf and the two types of NaNs (signaling and quiet).

NUMERIC COMPUTATION ALGORITHMS	CHAPTER 14

14.1 Standard Math Inputs and Outputs

The HP-71 standard internal math routines accept 15-forms as inputs and deliver 15-forms as outputs (see sec on Mathematical Operands in chapter on "Internal Data Representation"). The routine names usually end with "15". For example, SQR15, LN15, TAN15, AD2-15, etc.

For single argument functions (e.g. SQR15, LN15) the argument x is a 15-form in the CPU registers A and B (denoted here as the register pair (A,B)). Two argument functions (e.g. AD2-15, MP2-15) have their 15-form arguments in (A,B) and (C,D), with the first argument generally in (C,D). That is, x/y calls DV2-15 with (A,B)=y and (C,D)=x. The only other standard input is that DEC MODE must be set before entering a math routine.

The standard math output is a 15-form TRUNCATED (as opposed to ROUNDED) result in (A,B) along with other information in the hardware status bits SB (sticky bit) and XM (external module missing). Whenever SB=0 on exit, that implies that the 15-form result in (A,B) represents the mathematical function result EXACTLY. For example, 1/2 is 0.5 and that is precisely the result delivered by DV2-15, while 1/3 can not be represented exactly in 15 digits so it must have SB=1 on exit.

In writing mathematical routines, keep in mind that in some cases it is not practical to detect exactness, and in these cases SB should be set to 1. With the function y^x, for example, it would be hopeless to try to detect all exact values (e.g. 39.0625 ^ .25 = 2.5 exactly) and so in these cases SB should be set to 1. Therefore, whenever SB=0 the result is expected to be exact, while if SB=1 on exit then this indicates that the result COULD be inexact.

The XM bit is used to indicate an exceptional calculation. When XM=1 on exit , that indicates that a Divide-by-Zero (DVZ), Invalid Operation (IVL) or a 0^0 type exception has occured. In this case the pointer P identifies the exception:

P	Exception
3	Divide by zero
4	Invalid Operation
14	0^0

EXAMPLES:

1) 2/0 -- Output: (A,B)=Inf,SB=0,XM=1 & P=3

2) LN15(0) -- (A,B)=-Inf,SB=0,XM=1 & P=3

3) SQR15(-1) -- (A,B)=NaN,SB=0,XM=1 & P=4

4) 0^0 -- (A,B)=1,SB=0,XM=1 & P=14

5) EXP15(1E-20) -- (A,B)=1,SB=1,XM=0.

The HP-71 follows the proposed IEEE standard for exceptional math calculations. See the HP-71 reference manual for details.

The Standard Math Output is the input to uRES12. This routine packs a 15-form into a 12-form for delivery to the system and its documentation (in volume II) is worth reading at this point. Notice that all math routines deliver IEEE default values and thus avoid loss of control to error exits. These default values may be altered by uRES12 if the TRAP settings demand it. For anyone coding or using math routines, understanding inputs to uRES12 is an good place to start.

There is a Math Scratch Stack available to math routines. Its utility routines (STSCR,RCSCR,etc) save and restore the 15-forms in (A,B) and (C,D). It holds 4 15-forms.

Another restriction on math routines is that they do not alter CPU data pointers D0 & D1 since expression execution routines require that information on return from the math routine.

14.2 Statistical Algorithms

14.2.1 Summary Statistics

A sample is a collection of observations of a random variable. A matched sample consists of one or more samples where each observation in a sample is matched with an observation in each of the other samples. Each sample has the same number of elements, which we will denote by N. NVAR will denote the number of variables (samples).

A matched sample data set can then be visualized as a table with N rows and NVAR columns:

var:	x(1)	x(2)	...	x(j)	...	x(k)	...	x(NVAR)
1	x(1,1)	x(1,2)	...	x(1,j)	...	x(1,k)	...	x(1,NVAR)
2	x(2,1)	x(2,2)	...	x(2,j)	...	x(2,k)	...	x(2,NVAR)
.	.	.		.		.		.
.	.	.		.		.		.
.	.	.		.		.		.
i	x(i,1)	x(i,2)	...	x(i,j)	...	x(i,k)	...	x(i,NVAR)
.	.	.		.		.		.
.	.	.		.		.		.
.	.	.		.		.		.
N	x(N,1)	x(N,2)	...	x(N,j)	...	x(N,k)	...	x(N,NVAR)

Each row of this table represents a point in NVAR-dimensional space and will be called a data point. A data point could be considered an observation or realization of a NVAR-dimensional random variable, and we would have N such realizations.

Regardless of how such a data set is thought of (whether as a matched sample or as a sample of a vector-valued random variable), it may be useful to perform various statistical operations on it.

For the purposes of performing the HP-71 mainframe statistical operations and functions, we do not need to store the entire data set. Instead, we reduce, or summarize, the data in the following way. Let x(ij) represent the entry in row i and column j for i=1,2,...N and j=1,2,...,NVAR. The summary statistics are then:

N
NVAR

$$T(j) = E_{i}\, x(ij) \qquad j=1,2,...,NVAR$$

$$S(jk) = E_{i}\, [x(ij)-T(j)/N][x(ik)-T(k)/N] \qquad j,k=1,2,...,NVAR$$

Here E represents the summation symbol and we have deleted the commas between subscripts to save space. The T(j) represent the column totals and the S(jk) represent the mean-adjusted sums of squares and cross-products of the mean-adjusted variables.

Previous HP calculators accumulated the unadjusted sums of squares and sums of cross-products of the unadjusted variables:

$$\sum_i x(ij)x(ik)$$

rather than the S(jk). Three advantages to using the S(jk) are:

(1) They reduce the potential for loss of significance errors when the variables have zero means.
(2) Calculations based on them are faster than those based on the unadjusted ones.
(3) It is easier to use sample means, variances, and correlations in place of the original data.

The STAT statement reserves space for these summary statistics by dimensioning a statistical array. This array has one dimension and has length (NVAR+1)(NVAR+2)/2. NVAR is saved in nibble 3 of the statistical array's dope vector. (See "Internal Data Representation" chapter for more information about the statistical array dope vector.) The other statistics will be stored as

(N,T(1),S(11),T(2),S(12),S(22),...,S(NVAR,NVAR)).

Multiple matched samples can be stored simultaneously and analyzed in any order by using more than one statistical array.

A data point V = (V(1),...,V(NVAR)) is "added" to or "dropped" from the current data set using the ADD and DROP statements, respectively.

14.2.1.1 ADD operator

ADD updates the summary statistics according to:

```
 If N<0 then print "Invalid Stat Array" and stop

 For k=0 to NVAR

  For j=1 to k (skip if k=0)

   If N=0 then S(jk):=0

        else S(jk):=S(jk)+(N*V(j)-T(j))(N*V(k)-T(k))/[N(N+1)]

  Next j
```

```
T(k):=T(k)+V(k)

Next k

N:=N+1
```

14.2.1.2 DROP Operator

DROP updates the summary statistics according to:

```
If N<0 or 0<N<1 then print "Invalid Stat Array" and stop

If N=0 then print "Invalid Stat Operator" and stop

For k=0 to NVAR

 For j=1 to k (skip if k=0)

  If N=1 then S(jk):=0

        else S(jk):=S(jk)-(N*V(j)-T(j))(N*V(k)-T(k))/[N(N-1)]

 Next j

 T(k):=T(k)-V(k)

Next k

N:=N-1
```

14.2.2 Simple Linear Regression

The simple linear regression model is:

$$X(j) = a + b*X(k) + e$$

where X(j) is the dependent variable, X(k) is the independent variable, a and b are constants to be determined (estimated), and e represents random errors (uncorrelated with zero mean and unknown but constant variance). The constants a and b are determined by the method of least squares. That is, they are chosen to minimize the residual sum of squares:

$$\sum_i [X(ij) - a - b*X(ik)]^2$$

The solution is:

b = S(jk)/S(kk), and

a = [T(j) - b*T(k)]/N.

The LR statement specifies the current regression by specifying the dependent and independent variable numbers. These numbers are stored respectively in nibbles 5 and 4 of the current statistical array's dope vector.

Note that a (constant) random variable equal to one and having the coefficient a is implicitly present in the regression model. This interpretation can be quite useful when adding variables to or dropping variables from multiple linear regression models.

The mean-adjusted sum of squares for this constant variable and any mean-adjusted sum of cross-products involving this variable are zero. The total for this variable is N. Therefore, no additional summary statistics need be accumulated in order to implicitly include this variable in the data set.

For these reasons, this random variable, numbered 0, will always be considered present in a data set and 0 will be considered a valid variable number for all statistical statements and functions, except where explicitly stated otherwise.

```
+-----------------------------------------------+------------------+
|                                               |                  |
|   CLOCK SYSTEM                                |  CHAPTER  15     |
|                                               |                  |
+-----------------------------------------------+------------------+
```

The built-in clock system is an event scheduler for use in all time-keeping applications internal and external to the BASIC operating system. The clock system is built around one of the 24-bit countdown timers in the display driver chips.

15.1 Theory of Operation

15.1.1 Clock System Hardware

The hardware part of the clock system --the timer at address #2E2F8-- is a read/writeable 24-bit countdown timer which runs at 512 hz and which exerts a service request whenever the high bit is set. Treating this timer as a two's complement quantity, its range of values is 8388607 to -8388608 counts, where a count is 1/512th second. This is a range of about 4.55 to -4.55 hours.

15.1.2 Clock System Software

The software part of the clock system uses this timer to schedule the various events --ON TIMERs, 10-minute timeout, wait and external alarms-- that must be processed. It does so by setting the timer to go negative (exerting a service request) at the desired ("target") time, and maintaining a RAM location to keep track of the target time. The current time may be computed by subtracting the current timer value from the target time.

The clock system maintains several alarm slots for the various alarms which may be scheduled. One of these, the external alarm slot, is used for all external applications which need to schedule an alarm. The protocol for its use is explained below.

Whenever the clock system is accessed, it examines these slots and schedules whichever alarm is next due. When the alarm comes due, the timer exerts a service request. The CKSREQ (check-service-request) routine calls ALMSRV (alarm-serve), which will then schedule the next alarm. If an external alarm is due, the clock system will force an exception condition, which will cause a poll which will allow external alarm processing. More on that later.

If, when the clock system is accessed, there is no alarm due within 4 hours, the system will schedule a "clock system update". This is necessary simply to keep time because of the limited range of the hardware countdown timer.

15.2 Software Timebase Correction

Because of the finite accuracy of the timebase in the timer (estimated +-50 ppm), the clock system incorporates a software timebase correction scheme. The "Adjustment Factor" (or "AF") is a 24-bit 2's complement quantity which expresses a correction to be applied to the timebase. An adjustment factor of 0 indicates no correction. A non-zero adjustment factor indicates the number of counts to wait before adding (if AF is positive) or subtracting (if AF is negative) a count. In other words, it is the inverse of the inaccuracy. Whenever the clock system is accessed, it adds or subtracts the appropriate number of counts to keep the proper time.

The adjustment factor may be set by the user either directly (the AF(<arg>) function) or indirectly (the SETTIME, ADJUST and EXACT commands).

Several quantities are maintained in RAM to implement the adjustment factor scheme: TIMOFS (accumulated error), TIMLST (time of last EXACT), TIMLAF (time of last AF correction) and TIMAF (adjustment factor). Gory detail about its operation can be obtained from the documentation headers for CLKUPD and COMPAF.

15.3 Format of Time Information

Defining 1 Jan 0000 as the beginning of time and a "count" as 1/512 second, time in the clock system is maintained as number of counts since the beginning of time. The current time may be read by calling CMPT and easily converted to seconds by shifting right 9 bits.

Utilities exist to extract more useful quantities from the time. Here is a list, using the following terms: TIME = number of seconds since beginning of time; TIME-OF-DAY = number of seconds since midnight; DAY# = number of days since 1 Jan 0000.

TODT: Convert from TIME to TIME-OF-DAY and DAY#.

FROMDT: Inverse of TODT.

SECHMS: Convert from TIME-OF-DAY to hours/minutes/seconds.

HMSSEC: Inverse of SECHMS.

YMDDAY: Convert from year/month/day to DAY#.

DAYYMD: Inverse of YMDDAY.

JD2DAY: Convert from "Julian Date" (year and day-of-year) to DAY#.

DAY2JD: Inverse of JD2DAY.

15.4 Scheduling External Alarms

This section and the next contain the necessary information for interfacing with the clock system to schedule events.

Time is kept internally in 512ths of a second since 1 Jan 0000, which takes 48 bits. All time quantities, including alarms, are kept in these units. Scheduling an external alarm is simple: store the alarm time in RAM location ALRM6 and call CMPT. When the alarm comes due, the alarm can be processed and the next alarm can be scheduled. Certain rules must be followed in order to assure that alarms are not lost and the machine is not disrupted.

15.4.1 Scheduling Code

The SETALM subroutine sets an alarm given the absolute time at which the alarm is to come due. The SETALR subroutine sets an alarm relative to the current time. The routines are called with the time in A[11-0] and with C[0]=5. See the documentation headers for more information.

15.4.2 Priority of External Alarms

There is only one external alarm slot. If an application schedules an alarm through it, it must do so in such a way as not to destroy alarms which may have been scheduled by other applications. This simple protocol will insure that:

1) If alarm in ALRM6 is past due (i.e., current time > ALRM6), you can schedule your alarm.

2) If alarm in ALRM6 is not past due, you can schedule your alarm ONLY IF a) your alarm is not past due, AND b) your alarm occurs BEFORE the current alarm in ALRM6.

This is an important rule. If it is broken, external alarms can be lost.

15.4.3 When Alarms Come Due

When an alarm comes due, a service request will be exerted. This will lead to a pSREQ poll when the mainframe gets around to it and, if a program is running, a pEXCPT poll.

The pEXCPT poll will probably not be very useful for most time applications, except for those which should affect running program execution (such as ON-TIMER type statements). The pSREQ poll is useful, but it is not a time to disrupt the machine. It is, rather, a good time to schedule your next alarm (obeying the protocol, above) and to set up to process this alarm. See the Considerations section, below.

Accessing the clock system is fairly disruptive in terms of register usage and subroutine level usage. The RAM availability during the pSREQ poll does allow saving of R0, R1 and enough subroutine levels in scratch RAM to call the clock system safely. Since pSREQ can occur in any of many different states (during WAIT, during DISP, between statements, when machine is dormant, etc.), it is NOT a time to take over the machine. Performing a beep here would not be harmful; running a BASIC program would be harmful. The section below should provide some useful information in making the system work for you.

15.5 Developing Clock System Applications

15.5.1 Taking Control

The problem of taking control of the computer in a reasonable (i.e., not overly disruptive) way is an overriding consideration in development of a clock application. A good example of how to handle the problem is the processing of commands through HPIL in remote mode and device mode. That code would be good reading for somebody developing a clock system application.

For demonstration purposes, consider the HP-75C clock system. If an alarm comes due while the machine is turned off, the machine will wake up and process the alarm. If the machine is on, it will simply beep when the alarm comes due and process the alarm when the machine is turned off. This would be fairly simple to implement on the HP-71 by intercepting the following polls and doing the following:

pSREQ: Note past-due alarms. Beep if an alarm has become past due. Schedule new alarms.

pPWROF: Note machine entering sleep state. Schedule immediate wakeup through external alarm if you need to process an alarm.

pDSWNK: Wake machine. Put command in external command buffer to process alarm.

Developing an application which would process an alarm while the machine is awake would be more difficult. Recommended reading for this is the aforementioned HPIL code.

15.5.2 Insuring That the Alarm is Processed

Another consideration in light of the previous example: If the External Command Buffer is used to deliver a command which will process an alarm, there is no guarantee that the buffer will not be overwritten by another lexfile. Consider this scenario:

A pocket secretary application will execute a certain command when the alarm comes due. The alarm comes due, the machine wakes up and the pocket secretary puts the command in the external command buffer. The external command buffer is overwritten and the pocket secretary has no way to know if its command was ever executed.

A recommended solution would be for the pocket secretary to define a keyword (such as "PROCALRM") which is an instruction to process pending alarms (or to process the oldest alarm) and to delete them. This command may be executed from the keyboard by the user or it may be placed in the external command buffer. This way, if the external command buffer is overwritten, the alarm will not be deleted. The pocket secretary will know when its alarms have been processed.

15.5.3 Disrupting the Mainframe

A good, non-disruptive way to implement a program alarm would be to CALL the desired program.

15.5.4 Maintaining Your Own Alarm List

The clock system doesn't care how you maintain your own alarm list... it only cares that you schedule alarms in its time format: counts since 1 Jan 0000. And that you follow the scheduling protocol. No recommendation is expressed or implied as to whether you should keep your alarm list in an I/O buffer or a file.

15.6 Clock System Ram Usage

The following system memory is used in the internal clock system:

Name	Size(nibs)	Function
NXTIRQ	12	time of next sreq
ALRM1	12	on timer 1
ALRM2	12	on timer 2
ALRM3	12	on timer 3
ALRM4	12	timeout
ALRM5	12	pause
ALRM6	12	external alarm (set by pocket sec'y or controller)
PNDALM	2	bitmap of pending alarms
TIMOFS	12	time error offset for AF use
TIMLST	12	time of last exact
TIMLAF	12	time of last AF correction
TIMAF	6	accuracy factor

```
+----------------------------------------+-----------------+
|                                        |                 |
|   HP-71 ASSEMBLER INSTRUCTION SET      |  CHAPTER  16    |
|                                        |                 |
+----------------------------------------+-----------------+
```

This chapter describes the HP-71 assembler instruction set. The instruction mnemonics shown are those provided by the assembler used by the HP-71 software development team (which is available by special arrangement with HP). Almost all the mnemonics shown are also supported by the HP-71 FORTH/Assembler ROM.

16.1 CPU Overview

The HP-71 CPU is a proprietary CPU optimized for high-accuracy BCD math and low power consumption. The data path is 4 bits wide. Memory is accessed in 4-bit quantities called "nibbles" or "nibs". Addresses are 20 bits, yielding a physical address space of 512K bytes.

There are four working 64-bit registers, five scratch 64-bit registers, two 20-bit data pointer registers, one 4-bit pointer register, a 20-bit program counter, a 16-bit input register, and a 12-bit output register. Return addresses are stored on an eight-level hardware return stack that accepts 20-bit addresses. In addition, there 4 Hardware Status bits, a Carry bit, and 16 Program Status bits. The lower 12 Program Status bits can be manipulated as a 12-bit register.

16.1.1 Working and Scratch Registers

The working registers are used for data manipulation. Working registers A and C are also used for memory access.

The scratch registers are used to temporarily hold the contents of working registers. In addition, the lower 20 bits of scratch register R4 are used for interrupt processing by the operating system, and therefore are not normally available for data storage.

```
        WORKING REGISTERS                 SCRATCH REGISTERS
        -----------------                 -----------------
Name         Size                 Name          Size

    +--------------------+            +--------------------+
  A |      64 bits       |        R0  |      64 bits       |
    +--------------------+            +--------------------+

    +--------------------+            +--------------------+
  B |      64 bits       |        R1  |      64 bits       |
    +--------------------+            +--------------------+

    +--------------------+            +--------------------+
  C |      64 bits       |        R2  |      64 bits       |
    +--------------------+            +--------------------+

    +--------------------+            +--------------------+
  D |      64 bits       |        R3  |      64 bits       |
    +--------------------+            +--------------------+

                                      +--------------------+
                                  R4  |      64 bits*      |
                                      +--------------------+
```

* Note: the lower 20 bits of R4 are modified whenever an interrupt occurs, and are normally unavailable for storage

16.1.1.1 Field Selection

Subfields of the working registers may be manipulated by the use of field selection. The possible field selections range from the entire register to any single nibble of the register. Certain subfields are designed for use in BCD calculations. Others are used for data access or general data manipulation.

FIELD SELECTION FIELDS

```
 P      Digit pointed to by P register
UP      Digit 0 through digit pointed at by P
XS      Digit 2      - Exponent sign
 X      Digits 0-2   - Exponent and exponent sign
 S      Digit 15     - Mantissa sign
 M      Digits 3-14  - Mantissa
 B      Digits 0-1   - Exponent or byte field
 W      Digits 0-15  - Whole word
 A      Digits 0-4   - Address field
```

Nibbles of Register

```
-----------------------------------------------
15:14:13:12:11:10: 9: 8: 7: 6: 5: 4: 3: 2: 1: 0
-----------------------------------------------
S                                    XS|<-B->|
                               |<---- A ----->|
  |<------------- M --------------->|<-- X ->|
|<-------------------- W ------------------->|
```

16.1.2 Pointer Registers

The Data Pointer registers, D0 and D1, are used to contain addresses during memory access, and are used in conjunction with the working registers.

The P Pointer register is used in Field Selection operations with the working registers.

DATA POINTER REGISTERS

```
   +--------------------+         +--------------------+
D0 |      20 bits       |      D1 |      20 bits       |
   +--------------------+         +--------------------+
```

P POINTER REGISTER

```
  +----------+
P |  4 bits  |
  +----------+
```

16.1.3 Input, Output, and Program Counter Registers

The input/output registers are used to communicate with the system bus. The program counter points to the next instruction to be executed by the CPU.

INPUT AND OUTPUT REGISTERS

```
     +-----------+          +-----------+
  IN |  16 bits  |      OUT |  12 bits  |
     +-----------+          +-----------+
```

PROGRAM COUNTER REGISTER

```
     +-----------+
  PC |  20 bits  |
     +-----------+
```

16.1.4 Carry and Status Bits

The Carry bit is adjusted when a calculation or logical test is performed. During a calculation, such as incrementing or decrementing a register, it is set if the calculation overflows or borrows; otherwise it is cleared. During a logical test, such as comparing two registers for equality, it is set if the test is true; otherwise it is cleared.

The operating system uses the upper 4 Program Status bits to indicate the state of the operating system. The remaining 12 Program Status bits are generally available to applications software, and may be manipulated collectively as the ST register.

The four Hardware Status bits are set (but not cleared) by hardware-related events, and must therefore be cleared beforehand in order to detect a particular occurrence. They are individually accessible by name. The Module Pulled bit (MP) is set when a module is pulled from or added to the machine. The Sticky Bit (SB) is set when a "one" bit shifts off the right end of a working register as the result of a shift instruction. The Service Request (SR) bit is set as a result of the SREQ? instruction if any hardware service request is pending. The external Module Missing bit is set by execution of a zero opcode (RTNSXM instruction).

CARRY: 1 bit

PROGRAM STATUS: 16 bits

Bits	Usage
15 thru 12	Indicate state of operating system
11 thru 0	Available to programs, may be manipulated as the ST register

HARDWARE STATUS: 4 bits

Bit	Symbol	Name
3	MP	Module Pulled
2	SR	Service Request
1	SB	Sticky Bit
0	XM	External Module Missing

16.1.5 Loading Data from Memory

When data is read from memory into a register, the CPU places the lowest addressed nibble in the least significant nibble of the register. Thus the data appears to be loaded backwards in the register.

For example, if the data shown below in memory is read into the C register using the C=DAT1 4 instruction, the data in the register will be arranged as shown.

```
 Memory
Location  Value                C   Register
--------  -----         +--------------------------+
  1000      0           |  |       | 3 | 2 | 1 | 0 |
  1001      1           +--------------------------+
  1002      2            15 . . .   3   2   1   0
  1003      3
   .
   .
   .
```

This principle applies also to loading constants into a CPU register such as C, D0,or D1, since the CPU must read the constant from the instruction opcode in memory. For example, the instruction LCHEX 0123 procuces the opcode 333210 and the C register is loaded as shown above.

Note that the apparent reversal of data read from memory is compensated for by a similar reversal procedure when the data is written to memory from the CPU, which restores the data to its original orientation. (See below.)

16.1.6 Storing Data in Memory

When data is written to memory from a register, the CPU places the least significant nibble of the register in the lowest nibble of the addressed memory location. Thus, the data appears to be written in reverse order.

For example, if the data shown above in the C register is written to memory using the DAT1=C 4 instruction, the data will be written to memory as shown.

Note that the apparent reversal of data written to memory is compensated for by a similar reversal procedure when the data is read from memory by the CPU, which restores the data to its original orientation.

16.2 Instruction Syntax

16.2.1 Labels and Symbols

A label is a symbolic name for a numeric value. A label acquires its value by appearing in the label field of certain statements. The word "symbol" is a general term for a label, and the two are used interchangeably.

Labels are one to six alphanumberic characters with the following restrictions: the characters comma (,), space ( ) and right parenthesis are prohibited and the first character cannot be equal sign (=), sharp (#), single quote ('), left parenthesis, or the digits 0 through 9.

A label may be immediately preceded by an equal sign which declares the label to be an external symbol. An external symbol defined in one module may be referenced as an external symbol by another module. Such references are resolved when the modules are linked together. Certain HP-71 assemblers, such as the FORTH/ASSEMBLER ROM, have no associated linker and therefore do not support external symbols. In this case, any leading equal sign is ignored.

When a label is used as part of an expression, parentheses are required to delineate it. That is, AD1-10 is a label but (AD1)-10 is a computed expression.

16.2.2 Comments

A comment line begins with an asterisk (*) in column one, and may occur anywhere. An in-line comment may begin with any non-blank character and must follow the modifier field of an instruction (or the opcode if no modifier is required).

16.2.3 Expressions

Wherever an expression may appear in the modifier field of an instruction, it is represented by the symbol "expr" in the instruction descriptions below. Expressions consist of:

EXPRESSION COMPONENTS

Item	Examples
decimal constants	23434
hexadecimal constants	#1FF0 (less than #100000)
ascii constants	\AB\ (3 or less characters) 'AB' (3 or less characters)
operators	+ addition - subtraction % *256+ * multiplication / integer division ^ exponentiate & and ! or
*	Current assembly program counter
label	Symbol defined in the label field of an instruction
(expression)	Parenthesized expression

Two classes of instructions require a modifier field which contains a constant of a specific type that does not conform to the above rules. These are:

a) String constant which can exceed 3 characters

```
LCASC     'ascii'      or
LCASC     \ascii\

NIBASC    'ascii'      or
NIBASC    \ascii\
```

b) Unconditional Hex constant

```
LCHEX     4FFFFF
NIBHEX    4FFFFF
```

16.2.4 Sample Line Image

The format below is the recommended column alignment; however, the assembler is "free format" and only a space is required to delimit the different fields. A label, if present, must start by column 2.

```
1      8      15            31                                    80
v      v      v             v                                     v
--------------------------------------------------------------------
label  opcode modifier      comments
--------------------------------------------------------------------
```

16.3 Explanation of Symbols

In the following descriptions of the HP-71 assembler mnemonics, these symbols have the following meanings unless specified otherwise. In particular, note the symbols used to indicate the various values encoded within the assembled opcodes.

a	The hex digit used to encode the field selection in the assembled opcode of an instruction. See the Field Select Table in the next section for details.

b	The hex digit used to encode the field selection in the assembled opcode of an instruction. See the Field Select Table in the next section for details.

d	The number of digits represented by a field selection field. Used in calculating the execution cycle time of some instructions. See the Field Select Table in the next section for details. When used in an extended field selection fsd, represents an expression which indicates the number of nibbles of the register that will be affected by the instruction, proceeding from the low-order nibble to higher-order nibbles.

expr An expression that evaluates to an absolute or relocatable value, usually less than or equal to 5 nibbles in length.

fs Field selection symbol. See the Field Select Table in the next section for details.

fsd Extended field selection symbol. Represents either a normal field selection symbol fs, or an expression that gives the number of digits d of the register that will be affected by the instruction, proceeding from the low-order nibble to higher-order nibbles.

hh Two-digit hex constant, such as 08 or F2. Within an opcode represents the hex digits used to store the value of the expression in the opcode in reverse order (see "Loading Data From Memory").

hhhh Four-digit hex constant, such as 38FE. Within an opcode, represents the hex digits used to store the value of the expression in the opcode in reverse order (see "Loading Data From Memory").

hhhhh Five-digit hex constant, such as 308FE. Within an opcode, represents the hex digits used to store the value of the expression in the opcode in reverse order (see "Loading Data From Memory").

label A symbol defined in the label field of an instruction.

m A one-digit decimal integer constant.

n Represents an expression that evaluates to a 1-nibble value, unless specified otherwise. Within an opcode, represents the hex digit used to store the assembled value of the expression in the opcode.

nn Represents an expression that evaluates to a 2-nibble value, unless specified otherwise. Within an opcode, represents the hex digits used to store the assembled value of the expression in the opcode.

nnnn Represents an expression that evaluates to a 4-nibble value, unless specified otherwise. Within an opcode, represents the hex digits used to store the assembled value of the expression in the opcode.

nnnnn Represents an expression that evaluates to a 5-nibble value, unless specified otherwise. Within an opcode, represents the hex digits used to store the assembled value of the expression in the opcode.

16.3.1 Field Select Table

The following symbols are used in the instruction descriptions to denote the various possible field selections.

There are two ways in which field selection is encoded in the opcode of an instruction. These two patterns are shown in the table below, and are designated by the letter 'a' or 'b' in the opcode value given in the mnemonic descriptions below.

FIELD SELECT TABLE

Field	Name and Description	Opcode Representation (a)	Opcode Representation (b)	Number of Digits (d)
P	Pointer Field. Digit specified by P pointer register.	0	8	1
WP	Word-through-Pointer Field. Digits 0 through (P).	1	9	(P)
XS	Exponent Sign Field. Digit 2.	2	A	1
X	Exponent Field. Digits 0 - 2.	3	B	3
S	Sign Field. Digit 15.	4	C	1
M	Mantissa Field. Digits 3 - 14.	5	D	12
B	Byte Field. Digits 0 - 1.	6	E	2
W	Word Field. All digits.	7	F	16

16.4 Instruction Set Overview

The following pages briefly summarize the HP-71 instruction set. For further details please refer to the Mnemonic Dictionary which follows this summary.

16.4.1 GOTO Instructions

```
 ----
  l   = Statement Label
 ----

GOTO   label     Short unconditional branch
GOC    label     Short branch if Carry
GONC   label     Short branch if no Carry
GOLONG label     Long GOTO
GOVLNG label     Very long GOTO
GOYES  label     Short branch if test true (must
                   follow a Test Instruction)
```

16.4.2 GOSUB Instructions

```
GOSUB  label     Short transfer to subroutine
GOSUBL label     Long GOSUB
GOSBVL label     Very long GOSUB
```

16.4.3 Subroutine Returns

```
RTN              Unconditional return
RTNSC            Return and set Carry
RTNCC            Return and clear Carry
RTNSXM           Return and set XM bit (Module Missing)
RTI              Return and enable interupts
RTNC             Return if Carry set
RTNNC            Return if no Carry set
RTNYES           Return if test true (must follow a
                   Test Instruction)
```

16.4.4 Test Instructions

All test instructions must be followed with a GOYES or a RTNYES instruction. Although they appear to be two statements, in fact they combine to be one. Each test adjusts the Carry bit when performed.

16.4.4.1 Register Tests

```
----
 r,s = A,B,C  or  (r,s) = (C,D),(D,C)
 fs  = Field Select
----

?r=s   fs        Equal
?r#s   fs        Not equal
?r=0   fs        Equal to zero
?r#0   fs        Not equal to zero
?r>s   fs        Greater than
?r<s   fs        Less than
?r>=s  fs        Greater than or equal
?r<=s  fs        Less than or equal
```

16.4.4.2 P Pointer Tests

```
----
 0 <= n <= 15
----

?P=    n         Is P Pointer equal to n?
?P#    n         P Pointer not equal to n?
```

16.4.4.3 Hardware Status Bit Tests

```
?XM=0            Module Missing bit equal to zero?
?SB=0            Sticky Bit equal to zero?
?SR=0            Service Request bit equal to zero?
?MP=0            Module Pulled bit equal to zero?
```

16.4.4.4 Program Status Bit Tests

```
----
 0 <= n <= 15
----

?ST=1  n         Status n equal to 1?
?ST=0  n         Status n equal to 0?
```

```
?ST#1  n        Status not equal to 1?
?ST#0  n        Status not equal to 0?
```

16.4.5 P Pointer Instructions

```
----
 0 <= n <= 15
----

P=     n        Set P Pointer to n
P=P+1           Increment P Pointer, adjust Carry
P=P-1           Decrement P Pointer, adjust Carry
C+P+1           Add P Pointer plus one to A-field of C
CPEX   n        Exchange P Pointer with nibble n of C
P=C    n        Copy nibble n of C into P Pointer
C=P    n        Copy P Pointer into nibble n of C
```

16.4.6 Status Instructions

16.4.6.1 Program Status

```
----
 0 <= n <= 15
----

ST=1   n        Set Status n to 1
ST=0   n        Set Status n to 0
CSTEX           Exchange X field of C with Status 0-11
C=ST            Copy Status 0-11 into X field of C
ST=C            Copy X field of C into Status 0-11
CLRST           Clear Status 0-11
```

16.4.6.2 Hardware Status

```
SB=0            Clear Sticky Bit
SR=0            Clear Service Request bit (see SREQ?)
MP=0            Clear Module-Pulled bit
XM=0            Clear External Module Missing bit
CLRHST          Clear all 4 Hardware Status bits
```

16.4.7 System Control

SETHEX	Set arithmetic mode to hexadecimal
SETDEC	Set arithmetic mode to decimal
SREQ?	Sets Service Request bit if service has has been requested. C(0) shows what bit(s) are pulled high (if any)
C=RSTK	Pop return stack into A-field of C
RSTK=C	Push A-field of C onto return stack
CONFIG	Configure
UNCNFG	Unconfigure
RESET	Send Reset command to system bus
BUSCC	Send Bus command C onto system bus
SHUTDN	Stop CPU here (sleeps until wake-up)
C=ID	Request chip ID into A-field of C
INTOFF	Disable interrupts (doesn't affect ON-key or module-pulled interrupts)
INTON	Enable interrupts

16.4.8 Keyscan Instructions

OUT=C	Copy X field of C to OUTput register
OUT=CS	Copy nibble 0 of C to OUTput register
A=IN	Copy INput register to lower 4 nibbles of A
C=IN	Copy INput register to lower 4 nibbles of C

16.4.9 Register Swaps

----
s = R0,R1,R2,R3,R4

AsEX	Exchange register A with s
CsEX	Exchange register C with s
A=s	Copy s to register A
C=s	Copy s to register C
s=A	Copy register A to s
s=C	Copy register C to s

16.4.10 Data Manipulation

----
d = D0,D1

```
 1 <= n <= 16
 expr <= 5 nibbles
----

AdEX              Exchange Data ptr d with A-field of A
CdEX              Exchange Data ptr d with A-field of C
AdXS              Exchange lower 4 nibs of Data ptr d with
                    lower 4 nibs of A
CdXS              Exchange lower 4 nibs of Data ptr d with
                    lower 4 nibs of C
d=A               Copy A-field of A to Data pointer d
d=C               Copy A-field of C to Data pointer d
d=AS              Copy lower 4 nibs of A to lower 4 nibs
                    of Data pointer d
d=CS              Copy lower 4 nibs of C to lower 4 nibs
                    of Data pointer d
d=d+   n          Increment Data pointer d by n
d=d-   n          Decrement Data pointer d by n
d=HEX  hh         Load hh    into lower 2 nibs of Data ptr d
d=HEX  hhhh       Load hhhh into lower 4 nibs of Data ptr d
d=HEX  hhhhh      Load hhhhh into lower 5 nibs of Data ptr d
d=(2)  nn         Load nn into lower 2 nibs of Data ptr d
                    (any overflow is ignored)
d=(4)  nnnn       Load nnnn into lower 4 nibs of Data ptr d
                    (any overflow is ignored)
d=(5)  nnnnn      Load nnnnn into lower 5 nibs of Data ptr d
                    (any overflow is ignored)
```

16.4.11 Data Transfer

```
----
 fsd = Field select fs, or d (# of digits)
----

A=DAT0 fsd        Copy data from memory addressed by D0 into
                    A, field selected
C=DAT0 fsd        Copy data from memory addressed by D0 into
                    C, field selected
A=DAT1 fsd        Copy data from memory addressed by D1 into
                    A, field selected
C=DAT1 fsd        Copy data from memory addressed by D1 into
                    C, field selected
DAT0=A fsd        Copy data from A into memory addressed by
                    D0, field selected
DAT0=C fsd        Copy data from C into memory addressed by
                    D0, field selected
DAT1=A fsd        Copy data from A into memory addressed by
                    D1, field selected
DAT1=C fsd        Copy data from C into memory addressed by
```

D1, field selected

16.4.12 Load Constants

```
LCHEX  hhhhhhhh  Load hex constant into C
LC(m)  expr      Load the m-nibble constant into C
LCASC  'ascii'   Load up to 8 ASCII characters into C
LCASC  \ascii\   Load up to 8 ASCII characters into C
```

16.4.13 Shift Instructions

```
----
 r = A,B,C,D
 fs = Field Select
----

rSL    fs       Shift register r fs field Left  1 nibble
rSR    fs       Shift register r fs field Right 1 nibble
rSLC            Shift register r Left  Circular 1 nibble
rSRC            Shift register r Right Circular 1 nibble
rSRB            Shift register r Right          1 bit
```

16.4.14 Logical Operations

```
----
 r,s = A,B,C  or  (r,s) = (C,D),(D,C)
 fs = Field Select
----

r=r&s  fs       r AND s into r, field selected
r=r!s  fs       r OR  s into r, field selected
```

16.4.15 Arithmetics

The two groups of arithmetics differ in the range of registers available. In the first group (General usage) almost all combinations of the four working registers are possible; however, in the second group (Restricted usage) only a few select combinations are possible.

16.4.15.1 General Usage

```
----
 r,s = A,B,C  or (r,s) = (C,D),(D,C)
 fs  = Field Select
----

r=0     fs       Set r to zero
r=r+r   fs       Double r, adjust Carry
r=r+1   fs       Increment r by 1, adjust Carry
r=r-1   fs       Decrement r by 1, adjust Carry
r=-r    fs       10'S complement or 2'S complement, Carry
                   set if r#0 and in HEX mode, else clear
r=-r-1 fs        9'S complement or 1'S complement
                   Carry always cleared
r=r+s   fs       Sum r and s into r, adjust Carry
s=r+s   fs       Sum r and s into s, adjust Carry
r=s     fs       Copy s into r
s=r     fs       Copy r into s
rsEX    fs       Exchange r and s
```

16.4.15.2 Restricted Usage

```
----
 (r,s) = (A,B),(B,C),(C,A),(D,C)
----

r=r-s   fs       Difference of r and s into r, adjust Carry
r=s-r   fs       Difference of s and r into r, adjust Carry
s=s-r   fs       Difference of s and r into s, adjust Carry
```

16.4.16 No-Op Instructions

```
NOP3             Three nibble No-Op
NOP4             Four nibble No-Op
NOP5             Five nibble No-Op
```

16.4.17 Pseudo-Ops

16.4.17.1 Data Storage Allocation

```
----
 1 <= n <= 8
----
```

BSS	nnnnn	Allocate nnnnn number of zero nibs
CON(m)	expr	Generate m-nibble constant
REL(m)	expr	Generate m-nibble relative constant
NIBASC	'ascii'	Generate ascii characters, byte reversed
NIBASC	\ascii\	Generate ascii characters, byte reversed
NIBHEX	hhhh	Generate hexadecimal digits hhhh

16.4.17.2 Conditional Assembly

name	IF	expr	Start conditional assembly until ELSE or ENDIF if flag expr was set on invocation of assembler (optional use of name allows nesting of IF's)
name	ELSE		Conditional assembly if IF test was false
name	ENDIF		Ends conditional assembly started by IF

16.4.17.3 Listing Formatting

EJECT		Force new page in the assembly listing
STITLE	text	Force new page, set subtitle value to text
TITLE	text	Set title value to text

16.4.17.4 Symbol Definition

label	EQU	nnnnn	Defines label to have the value expr

16.4.17.5 Assembly Mode

ABS	nnnnn	Specify absolute assembly at adress given
END		Marks end of the assembly source

16.5 Mnemonic Dictionary

This section contains a description of each HP-71 assembler instruction or pseudo-op. The description shows the binary opcode generated by the mnemonic, if any, as well as the execution cycle time required if the mnemonic is an executable instruction.

The symbols used in these descriptions are explained in the "Explanation of Symbols" section earlier in this chapter.

```
**************************************************
*                   MNEMONICS                    *
**************************************************
```

```
?A#0   fs   -  Test for A not equal to 0
---------
fs = A                       opcode:  8ACyy
                             cycles:   13 + d (GO/RTNYES)
                                        6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)     opcode:  9aCyy
                             cycles:   13 + d (GO/RTNYES)
                                        6 + d (NO)
```

Test whether the fs field of A is not equal to 0. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

```
?A#B   fs   -  Test for A not equal to B
---------
fs = A                       opcode:  8A4yy
                             cycles:   13 + d (GO/RTNYES)
                                        6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)     opcode:  9a4yy
                             cycles:   13 + d (GO/RTNYES)
                                        6 + d (NO)
```

Test whether the fs field of A is not equal to the fs field of B. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?A#C fs - Test for A not equal to C
----

fs = A opcode: 8A6yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W) opcode: 9a6yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

Test whether the fs field of A is not equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?A<=B fs - Test for A less than or equal to B
----

fs = A opcode: 8BCyy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W) opcode: 9bCyy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

Test whether the fs field of A is less than or equal to the fs field of B. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?A<B fs - Test for A less than B
----

fs = A opcode: 8B6yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W) opcode: 9b6yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

Test whether the fs field of A is less than the fs field of B. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?A=0 fs - Test for A equal to 0

```
fs = A                          opcode:  8A8yy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)        opcode:  9a8yy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)
```

Test whether the fs field of A is equal to 0. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?A=B fs - Test for A equal to B

```
fs = A                          opcode:  8A0yy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)        opcode:  9a0yy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)
```

Test whether the fs field of A is equal to the fs field of B. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?A=C fs - Test for A equal to C

```
fs = A                          opcode:  8A2yy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)        opcode:  9a2yy
                                cycles:   13 + d (GO/RTNYES)
```

```
                                        6 + d (NO)
```

Test whether the fs field of A is equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?A>=B fs - Test for A greater than or equal to B

```
fs = A                        opcode:  8B8yy
                              cycles:   13 + d (GO/RTNYES)
                                         6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)      opcode:  9b8yy
                              cycles:   13 + d (GO/RTNYES)
                                         6 + d (NO)
```

Test whether the fs field of A is greater than or equal to the fs field of B. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?A>B fs - Test for A greater than B

```
fs = A                        opcode:  8B0yy
                              cycles:   13 + d (GO/RTNYES)
                                         6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)      opcode:  9b0yy
                              cycles:   13 + d (GO/RTNYES)
                                         6 + d (NO)
```

Test whether the fs field of A is greater than the fs field of B. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B#0 fs - Test for B not equal to 0

```
fs = A                         opcode:  8ADyy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)       opcode:  9aDyy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)
```

Test whether the fs field of B is not equal to 0. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B#A fs - Test for B not equal to A

```
fs = A                         opcode:  8A4yy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)       opcode:  9a4yy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)
```

Test whether the fs field of B is not equal to the fs field of A. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B#C fs - Test for B not equal to C

```
fs = A                         opcode:  8A5yy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)       opcode:  9a5yy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)
```

Test whether the fs field of B is not equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B<=C fs - Test for B less than or equal to C
----

fs = A   opcode: 8BDyy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)   opcode: 9bDyy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

Test whether the fs field of B is less than or equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B<C fs - Test for B less than C
----

fs = A   opcode: 8B5yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)   opcode: 9b5yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

Test whether the fs field of B is less than the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B=0 fs - Test for B equal to 0
----

fs = A   opcode: 8A9yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)   opcode: 9a9yy
cycles: 13 + d (GO/RTNYES)

6 + d (NO)

Test whether the fs field of B is equal to 0. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B=A fs - Test for B equal to A
----

fs = A	opcode:	8A0yy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode:	9a0yy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)

Test whether the fs field of B is equal to the fs field of A. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B=C fs - Test for B equal to C
----

fs = A	opcode:	8A1yy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode:	9a1yy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)

Test whether the fs field of B is equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B>=C fs - Test for B greater than or equal to C

```
fs = A                         opcode:  8B9yy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)       opcode:  9b9yy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)
```

Test whether the fs field of B is greater than or equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?B>C fs - Test for B greater than C

```
fs = A                         opcode:  8B1yy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)       opcode:  9b1yy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)
```

Test whether the fs field of B is greater than the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C#0 fs - Test for C not equal to 0

```
fs = A                         opcode:  8AEyy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)       opcode:  9aEyy
                               cycles:   13 + d (GO/RTNYES)
                                          6 + d (NO)
```

Test whether the fs field of C is not equal to 0. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C#A fs - Test for C not equal to A
----

fs = A	opcode: 8A6yy cycles: 13 + d (GO/RTNYES) 6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode: 9a6yy cycles: 13 + d (GO/RTNYES) 6 + d (NO)

Test whether the fs field of C is not equal to the fs field of A. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C#B fs - Test for C not equal to B
----

fs = A	opcode: 8A5yy cycles: 13 + d (GO/RTNYES) 6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode: 9a5yy cycles: 13 + d (GO/RTNYES) 6 + d (NO)

Test whether the fs field of C is not equal to the fs field of B. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C#D fs - Test for C not equal to D
----

fs = A	opcode: 8A7yy cycles: 13 + d (GO/RTNYES) 6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode: 9a7yy cycles: 13 + d (GO/RTNYES)

6 + d (NO)

Test whether the fs field of C is not equal to the fs field of D. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C<=A fs - Test for C less than or equal to A
----

fs = A	opcode: 8BEyy cycles: 13 + d (GO/RTNYES) 6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode: 9bEyy cycles: 13 + d (GO/RTNYES) 6 + d (NO)

Test whether the fs field of C is less than or equal to the fs field of A. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C<A fs - Test for C less than A
----

fs = A	opcode: 8B6yy cycles: 13 + d (GO/RTNYES) 6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode: 9b6yy cycles: 13 + d (GO/RTNYES) 6 + d (NO)

Test whether the fs field of C is less than the fs field of A. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C=0 fs - Test for C equal to 0
----

fs = A	opcode:	8AAyy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode:	9aAyy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)

Test whether the fs field of C is equal to 0. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C=A fs - Test for C equal to A
----

fs = A	opcode:	8A2yy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode:	9a2yy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)

Test whether the fs field of C is equal to the fs field of A. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?C=B fs - Test for C equal to B
----

fs = A	opcode:	8A1yy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)
fs = (P,WP,XS,X,S,M,B,W)	opcode:	9a1yy
	cycles:	13 + d (GO/RTNYES)
		6 + d (NO)

Test whether the fs field of C is equal to the fs field of B. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

```
?C=D   fs  -  Test for C equal to D
---------
fs = A                      opcode:  8A3yy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)    opcode:  9a3yy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)
```

Test whether the fs field of C is equal to the fs field of D. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

```
?C>=A  fs  -  Test for C greater than or equal to A
---------
fs = A                      opcode:  8BCyy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)    opcode:  9bCyy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)
```

Test whether the fs field of C is greater than or equal to the fs field of A. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

```
?C>A   fs  -  Test for C greater than A
---------
fs = A                      opcode:  8B2yy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)    opcode:  9b2yy
                            cycles:   13 + d (GO/RTNYES)
```

```
                                          6 + d (NO)
```

Test whether the fs field of C is greater than the fs field of A. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

```
?D#0   fs  -  Test for D not equal to 0
---------
fs = A                      opcode:  8AFyy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)    opcode:  9aFyy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)
```

Test whether the fs field of D is not equal to 0. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

```
?D#C   fs  -  Test for D not equal to C
---------
fs = A                      opcode:  8A7yy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)    opcode:  9a7yy
                            cycles:   13 + d (GO/RTNYES)
                                       6 + d (NO)
```

Test whether the fs field of D is not equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?D<=C fs - Test for D less than or equal to C
----

fs = A — opcode: 8BFyy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W) — opcode: 9bFyy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

Test whether the fs field of D is less than or equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?D<C fs - Test for D less than to C
----

fs = A — opcode: 8B7yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W) — opcode: 9b7yy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

Test whether the fs field of D is less than the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?D=0 fs - Test for D equal to 0
----

fs = A — opcode: 8AByy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W) — opcode: 9aByy
cycles: 13 + d (GO/RTNYES)
6 + d (NO)

Test whether the fs field of D is equal to 0. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?D=C fs - Test for D equal to C

```
fs = A                          opcode:  8A7yy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)        opcode:  9a7yy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)
```

Test whether the fs field of D is equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?D>=C fs - Test for D greater than or equal to C

```
fs = A                          opcode:  8BByy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)        opcode:  9bByy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)
```

Test whether the fs field of D is greater than or equal to the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?D>C fs - Test for D greater than C

```
fs = A                          opcode:  8B3yy
                                cycles:   13 + d (GO/RTNYES)
                                           6 + d (NO)

fs = (P,WP,XS,X,S,M,B,W)        opcode:  9b3yy
                                cycles:   13 + d (GO/RTNYES)
```

6 + d (NO)

Test whether the fs field of D is greater than the fs field of C. Must be followed by a GOYES or RTNYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?MP=0 - Test Module Pulled bit (MP)
----

opcode: 838yy
cycles: 13 (GO/RTNYES)
6 (NO)

Test whether the Module Pulled bit (MP) is zero. This hardware status bit is set whenever a module-pulled interrupt occurs, and must be explictly cleared by the MP=0 instruction. See the "HP-71 Hardware Specification" for more information. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?P# n - Test if P pointer not equal to n
----

opcode: 88nyy
cycles: 13 (GO/RTNYES)
6 (NO)

Test whether the P pointer is not equal to n. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?P= n - Test if P pointer is equal to n
----

opcode: 89nyy
cycles: 13 (GO/RTNYES)
6 (NO)

Test whether the P pointer is equal to n. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?SB=0 - Test Sticky Bit (SB)
----

opcode: 832yy
cycles: 13 (GO/RTNYES)
6 (NO)

Test whether the Sticky Bit (SB) is zero. The Sticky Bit is set on right shifts by a non-zero nibble or bit being shifted off the end of the field. The sticky bit must be cleared explicitly. Must be followed by a Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?SR=0 - Test Service Request bit (SR) for zero
----

opcode: 834yy
cycles: 13 (GO/RTNYES)
6 (NO)

Test whether the Service Request bit (SR) is zero. This hardware status bit is set by the SREQ? instruction, and must be cleared explicitly by the SR=0 instruciton. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?ST#0 n - Test status bit n not equal to 0
----

opcode: 87nyy
cycles: 14 (GO/RTNYES)
7 (NO)

Test whether Program Status bit n is set. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?ST#1 n - Test status bit n not equal to 1
----

opcode: 86nyy
cycles: 14 (GO/RTNYES)
7 (NO)

Test whether Program Status bit n is clear. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?ST=0 n - Test status bit n equal to 0
----

opcode: 86nyy
cycles: 14 (GO/RTNYES)
7 (NO)

Test whether Program Status bit n is clear. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?ST=1 n - Test status bit n equal to 1
----

opcode: 87nyy
cycles: 14 (GO/RTNYES)
7 (NO)

Test whether Program Status bit n is set. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

?XM=0 - Test External Module Missing bit (XM)
----

opcode: 83nyy
cycles: 13 (GO/RTNYES)
6 (NO)

Test the whether the External Module Missing bit (XM) is zero. This hardware status bit is set by the RTNSXM instruction, and must be explicitly cleared by the XM=0 instruction. Must be followed by a RTNYES or GOYES mnemonic. yy is determined by the following RTNYES or GOYES. Adjusts Carry.

A=-A fs - Two's complement of A into A
----

fs = A — opcode: F8, cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: Bb8, cycles: 3 + d

Complement the specified fs field of A. Complement is two's complement if in HEX mode, ten's complement if in DEC mode. Carry is set if the field is not zero, else Carry is cleared.

A=-A-1 fs - One's complement of A into A
----

fs = A — opcode: FC, cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: BbC, cycles: 3 + d

Perform a one's complement on the specified fs field of A. Carry is always cleared.

A=0 fs - Set A equal to 0
----

fs = A	opcode: D0 cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: Ab0 cycles: 3 + d

Set the specified fs field of A to zero. Carry is not affected.

A=A!B fs - A OR B into A
----

fs = A	opcode: 0EF8 cycles: 4 + d
fs = (P,WP,XS,X,S,M,B,W)	opcode: 0Ea8 cycles: 4 + d

Set the fs field of register A to its logical OR with the corresponding field of register B. Carry is not affected.

A=A!C fs - A OR C into A
----

fs = A	opcode: 0EFE cycles: 4 + d
fs = (P,WP,XS,X,S,M,B,W)	opcode: 0EaE cycles: 4 + d

Set the fs field of register A to its logical OR with the corresponding field of register C. Carry is not affected.

A=A&B fs - A AND B into A

```
fs = A                      opcode:  0EF0
                            cycles:    4 + d

fs = (P,WP,XS,X,S,M,B,W)    opcode:  0Ea0
                            cycles:    4 + d
```

Set the fs field of register A to its logical AND with the corresponding field of register B. Carry is not affected.

A=A&C fs - A AND C into A

```
fs = A                      opcode:  0EF6
                            cycles:    4 + d

fs = (P,WP,XS,X,S,M,B,W)    opcode:  0Ea6
                            cycles:    4 + d
```

Set the fs field of register A to its logical AND with the corresponding field of register C. Carry is not affected.

A=A+1 fs - Increment A

```
fs = A                      opcode:  E4
                            cycles:    7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  Ba4
                            cycles:    3 + d
```

Increment the specified fs field of register A by one. Adjusts Carry.

A=A+A fs - Sum of A and A into A
----

fs = A opcode: C4
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: Aa4
cycles: 3 + d

Double the specified fs field of register A. Adjusts Carry.

A=A+B fs - Sum of A and B into A
----

fs = A opcode: C0
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: Aa0
cycles: 3 + d

Set the specified fs field of register A to the sum of itself and the corresponding field of register B. Adjusts Carry.

A=A+C fs - Sum of A and C into A
----

fs = A opcode: CA
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: AaA
cycles: 3 + d

Set the specified fs field of register A to the sum of itself and the corresponding field of register C. Adjusts Carry.

A=A-1 fs - Decrement A

```
fs = A                      opcode:  CC
                            cycles:   7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  AaC
                            cycles:   3 + d
```

Decrement the specified fs field of register A by one. Adjusts Carry.

A=A-B fs - A minus B into A

```
fs = A                      opcode:  E0
                            cycles:   7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  Ba0
                            cycles:   3 + d
```

Set the specified fs field of register A to the difference between itself and the corresponding field of register B. Adjusts Carry.

A=A-C fs - A minus C into A

```
fs = A                      opcode:  EA
                            cycles:   7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  BaA
                            cycles:   3 + d
```

Set the specified fs field of register A to the difference between itself and the corresponding field of register C. Adjusts Carry.

A=B fs - Copy B to A
----

fs = A	opcode:	D4
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ab4
	cycles:	3 + d

Copy the fs field of register B into the corresponding field of register A. Carry is not affected.

A=B-A fs - B minus A into A
----

fs = A	opcode:	EC
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	BaC
	cycles:	3 + d

Set the specified fs field of register A to the inverse difference between itself and the corresponding field of register B. Adjusts Carry.

A=C fs - Copy C to A
----

fs = A	opcode:	DC
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	AbC
	cycles:	3 + d

Copy the fs field of register C into the corresponding field of register A. Carry is not affected.

A=DAT0 fsd - Load A from memory
----

fs = A	opcode: 142 cycles: 18
fs = B	opcode: 14A cycles: 15
fs = (P,WP,XS,X,S,M,W)	opcode: 152a cycles: 17 + d
fs = d	opcode: 15Ax (x=d-1) cycles: 16 + d

The amount of data (d nibbles) specified by fsd will be transferred from the memory address pointed to by D0 into the specified field of register A. The lowest-addressed nibble will be transferred into the lowest-order nibble of the register field, proceeding toward the higher-order nibbles. If fs = d, d nibbles are transferred into the register starting at nibble 0. See the section on "Loading Data From Memory" earlier in this chapter.

A=DAT1 fsd - Load A from memory
----

fs = A	opcode: 143 cycles: 18
fs = B	opcode: 14B cycles: 15
fs = (P,WP,XS,X,S,M,W)	opcode: 153a cycles: 17 + d
fs = d	opcode: 15Bx (x=d-1) cycles: 16 + d

The amount of data (d nibbles) specified by fsd will be transferred from the memory address pointed to by D1 into the specified field of register A. The lowest-addressed nibble will be transferred into the lowest-order nibble of the register field, proceeding toward the higher-order nibbles. If fs = d, d nibbles are transferred into the register starting at nibble 0. See the section on "Loading Data From Memory" earlier in this chapter.

A=IN - Load A with IN
----

opcode: 802
cycles: 7

Load the low-order 4 nibbles of the A register with the contents of the Input register.

A=R0 - Copy R0 to A
----

opcode: 110
cycles: 19

The contents of the scratch register R0 is copied to the working register A.

A=R1 - Copy R1 to A
----

opcode: 111
cycles: 19

The contents of the scratch register R1is copied to the working register A.

A=R2 - Copy R2 to A
----

opcode: 112
cycles: 19

The contents of the scratch register R2 is copied to the working register A.

A=R3 - Copy R3 to A
----

opcode: 113
cycles: 19

The contents of the scratch register R3 is copied to the working register A.

A=R4 - Copy R4 to A
----

opcode: 114
cycles: 19

The contents of the scratch register R4 is copied to the working register A.

ABEX fs - Exchange Registers A and B
----

fs = A — opcode: DC
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: AbC
cycles: 3 + d

Exchange the fs fields of registers of A and B. Carry is not affected.

ACEX fs - Exchange Registers A and C
----

fs = A — opcode: DE
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: AbE
cycles: 3 + d

Exchange the fs fields of registers of A and C. Carry is not affected.

ADOEX - Exchange A and D0 (nibs 0-4)
----

opcode: 132
cycles: 8

Exchange the A field of register A with Data pointer D0. Carry is not affected.

AD0XS - Exchange A and D0 short (nibs 0-3)
----

opcode: 13A
cycles: 7

Exchange the lower 4 nibbles of A with the lower 4 nibbles of Data pointer D0. Carry is not affected.

AD1EX - Exchange A and D1 (nibs 0-4)
----

opcode: 133
cycles: 8

Exchange the A field of register A with Data pointer D1. Carry is not affected.

AD1XS - Exchange A and D1 short (nibs 0-3)

---

opcode: 13B
cycles: 7

Exchange the lower 4 nibbles of A with the lower 4 nibbles of Data pointer D1. Carry is not affected.

AR0EX - Exchange A and R0

---

opcode: 120
cycles: 19

Exchange the contents of the working register A and the scratch register R0.

AR1EX - Exchange A and R1

---

opcode: 121
cycles: 19

Exchange the contents of the working register A and the scratch register R1.

AR2EX - Exchange A and R2

---

opcode: 122
cycles: 19

Exchange the contents of the working register A and the scratch register R2.

AR3EX - Exchange A and R3
----

opcode: 123
cycles: 19

Exchange the contents of the working register A and the scratch register R3.

AR4EX - Exchange A and R4
----

opcode: 124
cycles: 19

Exchange the contents of the working register A and the scratch register R4.

ASL fs - A Shift Left
----

fs = A opcode: F0
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: Bb0
cycles: 3 + d

Shift the contents of the specified fs field of register A left one nibble, without affecting the rest of the register. The nibble shifted off the left end of the field is lost. The new low-order nibble of the field is zero. The Sticky Bit (SB) is not affected.

ASLC - A Shift Left Circular
----

opcode: 810
cycles: 21

Circular shift register A left one nibble. Operates on all 16 digits. The Sticky Bit (SB) is not affected.

ASR fs - A Shift Right
----

fs = A	opcode:	F4
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Bb4
	cycles:	3 + d

Shift the contents of the specified fs field of register A right one nibble, without affecting the rest of the register. The nibble shifted off the right end of the field is lost, but the Sticky Bit (SB) is set if the nibble was non-zero. The new high-order nibble of the field is zero.

ASRB - A Shift Right Bit
----

opcode: 81C
cycles: 20

Shift register A right one bit. Operates on all 16 digits. The bit shifted off the end is lost, but the Sticky Bit (SB) is set if it was non-zero. The new high-order bit of the register is zero.

ASRC - A Shift Right Circular
----

opcode: 814
cycles: 21

Circular shift register A right one nibble. Operates on all 16 digits. The Sticky Bit (SB) is set if the nibble shifted from low-order around to high-order position was non-zero.

B=-B fs - Two's complement of B into B
----

fs = A opcode: F9
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: Bb9
cycles: 3 + d

Complement the specified fs field of B. Complement is two's complement if in HEX mode, ten's complement if in DEC mode. Carry is set if the field is not zero, else Carry is cleared.

B=-B-1 fs - One's complement of B into B
----

fs = A opcode: FD
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: BbD
cycles: 3 + d

Perform a one's complement on the specified fs field of B. Carry is always cleared.

B=0 fs - Set A equal to 0
----

fs = A	opcode: D1 cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: Ab1 cycles: 3 + d

Set the specified fs field of B to zero. Carry is not affected.

B=A fs - Copy A to B
----

fs = A	opcode: D8 cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: Ab8 cycles: 3 + d

Copy the fs field of register A into the corresponding field of register B. Carry is not affected.

B=B!A fs - B OR A into B
----

fs = A	opcode: 0EFC cycles: 4 + d
fs = (P,WP,XS,X,S,M,B,W)	opcode: 0EaC cycles: 4 + d

Set the fs field of register B to its logical OR with the corresponding field of register A. Carry is not affected.

B=B!C fs - B OR C into B
----

fs = A opcode: 0EF9
cycles: 4 + d

fs = (P,WP,XS,X,S,M,B,W) opcode: 0Ea9
cycles: 4 + d

Set the fs field of register B to its logical OR with the corresponding field of register C. Carry is not affected.

B=B&A fs - B AND A into B
----

fs = A opcode: 0EF4
cycles: 4 + d

fs = (P,WP,XS,X,S,M,B,W) opcode: 0Ea4
cycles: 4 + d

Set the fs field of register B to its logical AND with the corresponding field of register A. Carry is not affected.

B=B&C fs - B AND C into B
----

fs = A opcode: 0EF1
cycles: 4 + d

fs = (P,WP,XS,X,S,M,B,W) opcode: 0Ea1
cycles: 4 + d

Set the fs field of register B to its logical AND with the corresponding field of register C. Carry is not affected.

B=B+1 fs - Increment B
----

fs = A	opcode:	E5
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ba5
	cycles:	3 + d

Increment the specified fs field of register B by one. Adjusts Carry.

B=B+A fs - Sum of B and A into B
----

fs = A	opcode:	C8
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Aa8
	cycles:	3 + d

Set the specified fs field of register B to the sum of itself and the corresponding field of register A. Adjusts Carry.

B=B+B fs - Sum of B and B into B
----

fs = A	opcode:	C5
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Aa5
	cycles:	3 + d

Double the specified fs field of register B. Adjusts Carry.

B=B+C *fs* - Sum of B and C into B
----

fs = A	opcode: C1 cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: Aa1 cycles: 3 + d

Set the specified *fs* field of register B to the sum of itself and the corresponding field of register C. Adjusts Carry.

B=B-1 fs - Decrement B
----

fs = A	opcode: CD cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: AaD cycles: 3 + d

Decrement the specified fs field of register B by one. Adjusts Carry.

B=B-A fs - B minus A into B
----

fs = A	opcode: E8 cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: Ba8 cycles: 3 + d

Set the specified fs field of register B to the difference between itself and the corresponding field of register A. Adjusts Carry.

B=B-C fs - B minus C into B
----

fs = A	opcode:	E1
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ba1
	cycles:	3 + d

Set the specified fs field of register B to the difference between itself and the corresponding field of register C. Adjusts Carry.

B=C fs - Copy C to B
----

fs = A	opcode:	D5
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ab5
	cycles:	3 + d

Copy the fs field of register C into the corresponding field of register B. Carry is not affected.

B=C-B fs - C minus B into B
----

fs = A	opcode:	ED
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	BaD
	cycles:	3 + d

Set the specified fs field of register B to the inverse difference between itself and the corresponding field of register C. Adjusts Carry.

BAEX fs - Exchange Registers B and A

```
fs = A                        opcode:  DC
                              cycles:   7

fs = (P,WP,XS,X,S,M,B,W)      opcode:  AbC
                              cycles:   3 + d
```

Exchange the fs fields of registers of B and A. Carry is not affected.

BCEX fs - Exchange Registers B and C

```
fs = A                        opcode:  DD
                              cycles:   7

fs = (P,WP,XS,X,S,M,B,W)      opcode:  AbD
                              cycles:   3 + d
```

Exchange the fs fields of registers of B and C. Carry is not affected.

BSL fs - B Shift Left

```
fs = A                        opcode:  F1
                              cycles:   7

fs = (P,WP,XS,X,S,M,B,W)      opcode:  Bb1
                              cycles:   3 + d
```

Shift the contents of the specified fs field of register B left one nibble, without affecting the rest of the register. The nibble shifted off the left end of the field is lost. The new low-order nibble of the field is zero. The Sticky Bit (SB) is not affected.

BSLC - B Shift Left Circular
----

opcode: 811
cycles: 21

Circular shift register B left one nibble. Operates on all 16 digits. The Sticky Bit (SB) is not affected.

BSR fs - B Shift Right
----

fs = A — opcode: F5, cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: Bb5, cycles: 3 + d

Shift the contents of the specified fs field of register B right one nibble, without affecting the rest of the register. The nibble shifted off the right end of the field is lost, but the Sticky Bit (SB) is set if the nibble was non-zero. The new high-order nibble of the field is zero.

BSRB - B Shift Right Bit
----

opcode: 81D
cycles: 20

Shift register B right one bit. Operates on all 16 digits. The bit shifted off the end is lost, but the Sticky Bit (SB) is set if it was non-zero. The new high-order bit of the register is zero.

BSRC - B Shift Right Circular
----

opcode: 815
cycles: 21

Circular shift register B right one nibble. Operates on all 16 digits. The Sticky Bit (SB) is set if the nibble shifted from low-order around to high-order position was non-zero.

BUSCC - Bus Command "C"
----

opcode: 80B
cycles: 6

Enters the HP-71 bus command "C" onto the system bus (this command is reserved for later use). No other operation is performed. See the "HP-71 Hardware Specification" for more information.

C+P+1 - Increment C by One Plus P Pointer
----

opcode: 809
cycles: 8

The A field of the C register is incremented by one plus the value of the P pointer. Arithmetic is always in hex mode. Adjusts Carry.

C=-C fs - Two's complement of C into C
----

fs = A   opcode: FA
                                cycles: 7

fs = (P,WP,XS,X,S,M,B,W)   opcode: BbA
                                cycles: 3 + d

Complement the specified fs field of C. Complement is two's complement if in HEX mode, ten's complement if in DEC mode. Carry is set if the field is not zero, else Carry is cleared.

C=-C-1 fs - One's complement of C into C

fs = A	opcode:	FE
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	BbE
	cycles:	3 + d

Perform a one's complement on the specified fs field of C. Carry is always cleared.

C=0 fs - Set C equal to 0

fs = A	opcode:	D2
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ab2
	cycles:	3 + d

Set the specified fs field of C to zero. Carry is not affected.

C=A fs - Copy A to C

fs = A	opcode:	D6
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ab6
	cycles:	3 + d

Copy the fs field of register A into the corresponding field of register C. Carry is not affected.

C=A-C fs - A minus C into C
----

fs = A	opcode:	EE
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	BaE
	cycles:	3 + d

Set the specified fs field of register C to the inverse difference between itself and the corresponding field of register A. Adjusts Carry.

C=B fs - Copy B to C
----

fs = A	opcode:	D9
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ab9
	cycles:	3 + d

Copy the fs field of register B into the corresponding field of register C. Carry is not affected.

C=C!A fs - C OR A into C
----

fs = A	opcode:	0EFE
	cycles:	4 + d
fs = (P,WP,XS,X,S,M,B,W)	opcode:	0EaE
	cycles:	4 + d

Set the fs field of register C to its logical OR with the corresponding field of register A. Carry is not affected.

C=C!B fs - C OR B into C

```
fs = A                          opcode:  0EFD
                                cycles:    4 + d

fs = (P,WP,XS,X,S,M,B,W)        opcode:  0EaD
                                cycles:    4 + d
```

Set the fs field of register C to its logical OR with the corresponding field of register B. Carry is not affected.

C=C!D fs - C OR D into C

```
fs = A                          opcode:  0EFF
                                cycles:    4 + d

fs = (P,WP,XS,X,S,M,B,W)        opcode:  0EaF
                                cycles:    4 + d
```

Set the fs field of register C to its logical OR with the corresponding field of register D. Carry is not affected.

C=C&A fs - C AND A into A

```
fs = A                          opcode:  0EF2
                                cycles:    4 + d

fs = (P,WP,XS,X,S,M,B,W)        opcode:  0Ea2
                                cycles:    4 + d
```

Set the fs field of register C to its logical AND with the corresponding field of register A. Carry is not affected.

C=C&B fs - C AND B into C
----

fs = A	opcode: 0EF5 cycles: 4 + d
fs = (P,WP,XS,X,S,M,B,W)	opcode: 0Ea5 cycles: 4 + d

Set the fs field of register C to its logical AND with the corresponding field of register B. Carry is not affected.

C=C&D fs - C AND D into C
----

fs = A	opcode: 0EF7 cycles: 4 + d
fs = (P,WP,XS,X,S,M,B,W)	opcode: 0Ea7 cycles: 4 + d

Set the fs field of register C to its logical AND with the corresponding field of register D. Carry is not affected.

C=C+1 fs - Increment C
----

fs = A	opcode: E6 cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: Ba6 cycles: 3 + d

Increment the specified fs field of register C by one. Adjusts Carry.

C=C+A fs - Sum of C and A into C

```
fs = A                      opcode:  C2
                            cycles:    7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  Aa2
                            cycles:    3 + d
```

Set the specified fs field of register C to the sum of itself and the corresponding field of register A. Adjusts Carry.

C=C+B fs - Sum of C and B into C

```
fs = A                      opcode:  C9
                            cycles:    7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  Aa9
                            cycles:    3 + d
```

Set the specified fs field of register C to the sum of itself and the corresponding field of register B. Adjusts Carry.

C=C+C fs - Sum of C and C into C

```
fs = A                      opcode:  C6
                            cycles:    7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  Aa6
                            cycles:    3 + d
```

Double the specified fs field of register C. Adjusts Carry.

C=C+D fs - Sum of C and D into C
----

fs = A	opcode: CB cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: AaB cycles: 3 + d

Set the specified fs field of register C to the sum of itself and the corresponding field of register D. Adjusts Carry.

C=C-1 fs - Decrement C
----

fs = A	opcode: CE cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: AaE cycles: 3 + d

Decrement the specified fs field of register C by one. Adjusts Carry.

C=C-A fs - C minus A into C
----

fs = A	opcode: E2 cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: Ba2 cycles: 3 + d

Set the specified fs field of register C to the difference between itself and the corresponding field of register A. Adjusts Carry.

C=C-B fs - C minus B into C

```
fs = A                      opcode:  E9
                            cycles:   7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  Ba9
                            cycles:   3 + d
```

Set the specified fs field of register C to the difference between itself and the corresponding field of register B. Adjusts Carry.

C=C-D fs - C minus D into C

```
fs = A                      opcode:  EB
                            cycles:   7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  BaB
                            cycles:   3 + d
```

Set the specified fs field of register C to the difference between itself and the corresponding field of register D. Adjusts Carry.

C=D fs - Copy D to C

```
fs = A                      opcode:  DB
                            cycles:   7

fs = (P,WP,XS,X,S,M,B,W)    opcode:  AbB
                            cycles:   3 + d
```

Copy the fs field of register D into the corresponding field of register C. Carry is not affected.

C=DAT0 fsd - Load C from memory
----

fs = A	opcode:	146
	cycles:	'18
fs = B	opcode:	14E
	cycles:	15
fs = (P,WP,XS,X,S,M,W)	opcode:	156a
	cycles:	17 + d
fs = d	opcode:	15Ex (x=d-1)
	cycles:	16 + d

The amount of data (d nibbles) specified by fsd will be transferred from the memory address pointed to by D0 into the specified field of register C. The lowest-addressed nibble will be transferred into the lowest-order nibble of the register field, proceeding toward the higher-order nibbles. If fs = d, d nibbles are transferred into the register starting at nibble 0. See the section on "Loading Data From Memory" earlier in this chapter.

C=DAT1 fsd - Load C from memory
----

fs = A	opcode:	147
	cycles:	18
fs = B	opcode:	14F
	cycles:	15
fs = (P,WP,XS,X,S,M,W)	opcode:	157a
	cycles:	17 + d
fs = d	opcode:	15Fx (x=d-1)
	cycles:	16 + d

The amount of data (d nibbles) specified by fsd will be transferred from the memory address pointed to by D1 into the specified field of register C. The lowest-addressed nibble will be transferred into the lowest-order nibble of the register field, proceeding toward the higher-order nibbles. If fs = d, d nibbles are transferred into the register starting at nibble 0. See the section on "Loading Data From Memory" earlier in this chapter.

C=ID - Request chip ID
----

opcode: 806
cycles: 11

The chip which has its DAISY-IN line high and its configuration flag low will send its 5 nibble ID register to the system bus which will be loaded into the low-order 5 nibbles (A field) of the C register. See the "HP-71 Hardware Specification" for more information.

C=IN - Load C with IN
----

opcode: 803
cycles: 7

Load the low-order 4 nibbles of the C register with the contents of the Input register. See the "HP-71 Hardware Specification" for more information.

C=P n - Copy P Pointer into Nibble n of C
----

opcode: 80CN
cycles: 6

Copy P pointer into C register at digit position specified by n.

C=R0 - Copy R0 to C
----

opcode: 118
cycles: 19

The contents of the scratch register R0 is copied to the working register C.

C=R1 - Copy R1 to C
----

opcode: 119
cycles: 19

The contents of the scratch register R1 is copied to the working register C.

C=R2 - Copy R2 to C
----

opcode: 11A
cycles: 19

The contents of the scratch register R2 is copied to the working register C.

C=R3 - Copy R3 to C
----

opcode: 11B
cycles: 19

The contents of the scratch register R3 is copied to the working register C.

C=R4 - Copy R4 to C
----

opcode: 11C
cycles: 19

The contents of the scratch register R4 is copied to the working register C.

C=RSTK - Pop stack to C
----

opcode: 07
cycles: 8

Pop the top-most address off of the hardware return stack, placing the address in the lower 5 nibbles (A field) of register C. The high-order nibbles of C are unchanged. As the address is popped from the return stack, a zero address is inserted at the bottom of the stack. Compare with the RTN instruction.

C=ST - Status to C
----

opcode: 09
cycles: 6

Copy the low-order 12 bits of the status register into the low-order 12 bits (X field) of the C register.

CAEX fs - Exchange Registers C and A
----

fs = A — opcode: DE, cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: AbE, cycles: 3 + d

Exchange the fs fields of registers of C and A. Carry is not affected.

CBEX fs - Exchange Registers C and B
----

fs = A	opcode: DD cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: AbD cycles: 3 + d

Exchange the fs fields of registers of C and B. Carry is not affected.

CD0EX - Exchange C and D0 (nibs 0-4)
----

opcode: 136
cycles: 8

Exchange the A field of register C with Data pointer D0. Carry is not affected.

CD0XS - Exchange C and D0 short (nibs 0-3)
----

opcode: 13E
cycles: 7

Exchange the lower 4 nibbles of C with the lower 4 nibbles of Data pointer D0. Carry is not affected.

CD1EX - Exchange C and D1 (nibs 0-4)
----

opcode: 137
cycles: 8

Exchange the A field of register C with Data pointer D1. Carry is not affected.

CD1XS - Exchange C and D1 short (nibs 0-3)

----

opcode: 13F
cycles: 7

Exchange the lower 4 nibbles of C with the lower 4 nibbles of Data pointer D1. Carry is not affected.

CDEX *fs* - Exchange Registers C and D

----

fs = A opcode: DF
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: AbF
cycles: 3 + d

Exchange the fs fields of registers of C and D. Carry is not affected.

CLRHST - Clear Hardware Status bits

----

opcode: 82F
cycles: 3

Clears the 4 Hardware Status bits XM, SB, SR and MP. Note that the opcode is actually 82x, where x is merely a mask for which Hardware Status bits to clear, as follows:

bit 0 - External Module Missing bit (see XM=0 mnemonic) bit 1 - Sticky Bit (see SB=0 mnemonic) bit 2 - Service Request bit (see SR=0 mnemonic) bit 3 - Module Pulled bit (see MP=0 mnemonic)

For example opcode 829 clears XM and MP. Although there is no mnemonic for this, the opcode can be inserted into the code by using, for example, NIBHEX 829.

CLRST - Clear Program Status
----

opcode: 08
cycles: 6

Clear the low-order 12 bits (S0 through S11) of the Program Status register ST.

CONFIG - Configure
----

opcode: 805
cycles: 11

Copy the low-order 5 nibbles (A field) of the C register into the Configuration register of the chip which has its DAISY-IN line high and its configuration flag low. See the "HP-71 Hardware Specification" for information.

CPEX n - Exchange Nibble n of C With P Pointer
----

opcode: 80FN
cycles: 6

Exchange the P pointer with digit n of the C register.

CR0EX - Exchange C and R0
----

opcode: 128
cycles: 19

Exchange the contents of the working register C and the scratch register R0.

CR1EX - Exchange C and R1
----

opcode: 129
cycles: 19

Exchange the contents of the working register C and the scratch register R1.

CR2EX - Exchange C and R2
----

opcode: 12A
cycles: 19

Exchange the contents of the working register C and the scratch register R2.

CR3EX - Exchange C and R3
----

opcode: 12B
cycles: 19

Exchange the contents of the working register C and the scratch register R3.

CR4EX - Exchange C and R4
----

opcode: 12C
cycles: 19

Exchange the contents of the working register C and the scratch register R4.

CSL fs - C Shift Left
----

fs = A	opcode:	F2
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Bb2
	cycles:	3 + d

Shift the contents of the specified fs field of register C left one nibble, without affecting the rest of the register. The nibble shifted off the left end of the field is lost. The new low-order nibble of the field is zero. The Sticky Bit (SB) is not affected.

CSLC - C Shift Left Circular
----

opcode:	812
cycles:	21

Circular shift register C left one nibble. Operates on all 16 digits. The Sticky Bit (SB) is not affected.

CSR fs - C Shift Right
----

fs = A	opcode:	F6
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Bb6
	cycles:	3 + d

Shift the contents of the specified fs field of register C right one nibble, without affecting the rest of the register. The nibble shifted off the right end of the field is lost, but the Sticky Bit (SB) is set if the nibble was non-zero. The new high-order nibble of the field is zero.

CSRB - C Shift Right Bit
----

opcode: 81E
cycles: 20

Shift register C right one bit. Operates on all 16 digits. The bit shifted off the end is lost, but the Sticky Bit (SB) is set if it was non-zero. The new high-order bit of the register is zero.

CSRC - C Shift Right Circular
----

opcode: 816
cycles: 21

Circular shift register C right one nibble. Operates on all 16 digits. The Sticky Bit (SB) is set if the nibble shifted from low-order around to high-order position was non-zero.

CSTEX - Exchange Status
----

opcode: 0B
cycles: 6

Exchange the low-order 12 bits (S0 through S11) of the Program Status register ST with the low-order 12 bits of the C register.

D0=(2) nn - Load 2 Nibbles Into D0
----

opcode: 19nn
cycles: 4

Load the low-order two nibbles of D0 with nn. The upper nibbles of D0 remain unchanged. Any overflow is ignored. The assembled digits of nn are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

DO=(4) nnnn - Load 4 Nibbles Into DO
----

opcode: 1Annnn
cycles: 6

Load the low-order four nibbles of DO with nnnn. The upper nibble of DO remains unchanged. Any overflow is ignored. The assembled digits of nnnn are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

DO=(5) nnnnn - Load 5 Nibbles Into DO
----

opcode: 1Bnnnnn
cycles: 7

Load all five nibbles of DO with nnnnn. Any overflow is ignored. The assembled digits of nnnnn are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

DO=A - Copy A to DO (nibs 0-4)
----

opcode: 130
cycles: 8

The A field of register A is copied into Data pointer register DO. Carry is not affected.

D0=AS - Copy A to D0 short (nibs 0-3)

opcode: 138
cycles: 7

The lower 4 nibbles of A are copied into the lower 4 nibbles of Data pointer register D0. Carry is not affected.

D0=C - Copy C to D0 (nibs 0-4)

opcode: 134
cycles: 8

The A field of register C is copied into Data pointer register D0. Carry is not affected.

D0=CS - Copy C to D0 short (nibs 0-3)

opcode: 13C
cycles: 7

The lower 4 nibbles of C are copied into the lower 4 nibbles of Data pointer register D0. Carry is not affected.

D0=D0+ n - Add n to D0 (1<=n<=16)

opcode: 16x (x=n-1)
cycles: 7

Increment D0 by n. Adjusts Carry.

D0=D0- n - Subtract n from D0 (1<=n<=16)
----

opcode: 18x (x=n-1)
cycles: 7

Decrement D0 by n. Adjusts Carry.

D0=HEX hh - Load D0 with hex constant hh
----

opcode: 19hh
cycles: 4

Load the low-order two nibbles of D0 with the hex constant hh. The upper nibbles of D0 remain unchanged. The digits of hh are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D0=HEX hhhh - Load D0 with hex constant hhhh
----

opcode: 1Ahhhh
cycles: 6

Load the low-order four nibbles of D0 with the hex constant hhhh. The upper nibble of D0 remains unchanged. The digits of hhhh are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D0=HEX hhhhh - Load D0 with hex constant hhhhh

opcode: 1Bhhhhh
cycles: 7

Load all five nibbles of D0 with the hex constant hhhhh. The digits of hhhhh are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D1=(2) nn - Load 2 Nibbles Into D1

opcode: 1Dnn
cycles: 4

Load the low-order two nibbles of D1 with nn. The upper nibbles of D1 remain unchanged. Any overflow is ignored. The assembled digits of nn are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D1=(4) nnnn - Load 4 Nibbles Into D1

opcode: 1Ennnn
cycles: 6

Load the low-order four nibbles of D1 with nnnn. The upper nibble of D1 remains unchanged. Any overflow is ignored. The assembled digits of nnnn are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D1=(5) nnnnn - Load 5 Nibbles Into D1
----

opcode: 1Fnnnnn
cycles: 7

Load all five nibbles of D1 with nnnnn. Any overflow is ignored. The assembled digits of nnnnn are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D1=A - Copy A to D1 (nibs 0-4)
----

opcode: 131
cycles: 8

The A field of register A is copied into Data pointer register D1. Carry is not affected.

D1=AS - Copy A to D1 short (nibs 0-3)
----

opcode: 139
cycles: 7

The lower 4 nibbles of A are copied into the lower 4 nibbles of Data pointer register D1. Carry is not affected.

D1=C - Copy C to D1 (nibs 0-4)
----

opcode: 135
cycles: 8

The A field of register C is copied into Data pointer register D1. Carry is not affected.

D1=CS - Copy C to D1 short (nibs 0-3)
----

opcode: 13D
cycles: 7

The lower 4 nibbles of C are copied into the lower 4 nibbles of Data pointer register D1. Carry is not affected.

D1=D1+ n - Add n to D1 (1<=n<=16)
----

opcode: 17x (x=n-1)
cycles: 7

Increment D1 by n. Adjusts Carry.

D1=D1- n - Subtract n from D1 (1<=n<=16)
----

opcode: 1CX (X=n-1)
cycles: 7

Decrement D1 by n. Adjusts Carry.

D1=HEX hh - Load D1 with hex constant hh
----

opcode: 1Dhh
cycles: 4

Load the low-order two nibbles of D1 with the hex constant hh. The upper nibbles of D1 remain unchanged. The digits of hh are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D1=HEX hhhh - Load D1 with hex constant hhhh
----

opcode: 1Ehhhh
cycles: 6

Load the low-order four nibbles of D1 with the hex constant hhhh. The upper nibble of D1 remains unchanged. The digits of hhhh are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D1=HEX hhhhh - Load D1 with hex constant hhhhh
----

opcode: 1Fhhhhh
cycles: 7

Load all five nibbles of D1 with the hex constant hhhhh. The digits of hhhhh are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

D=-D fs - Two's complement of D into D
----

fs = A — opcode: FB, cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: BbB, cycles: 3 + d

Complement the specified fs field of D. Complement is two's complement if in HEX mode, ten's complement if in DEC mode. Carry is set if the field is not zero, else Carry is cleared.

D=-D-1 fs - One's complement of D into D
----

fs = A	opcode:	FF
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	BbF
	cycles:	3 + d

Perform a one's complement on the specified fs field of D. Carry is always cleared.

D=0 fs - Set D equal to 0
----

fs = A	opcode:	D3
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ab3
	cycles:	3 + d

Set the specified fs field of D to zero. Carry is not affected.

D=C fs - Copy C to D
----

fs = A	opcode:	D7
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	Ab7
	cycles:	3 + d

Copy the fs field of register C into the corresponding field of register D. Carry is not affected.

D=C-D fs - C minus D into D
----

fs = A	opcode:	ED
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	BaD
	cycles:	3 + d

Set the specified fs field of register D to the inverse difference between itself and the corresponding field of register C. Adjusts Carry.

D=D!C fs - D OR C into D
----

fs = A	opcode:	0EFF
	cycles:	4 + d
fs = (P,WP,XS,X,S,M,B,W)	opcode:	0EaF
	cycles:	4 + d

Set the fs field of register D to its logical OR with the corresponding field of register C. Carry is not affected.

D=D&C fs - D AND C into D
----

fs = A	opcode:	0EF7
	cycles:	4 + d
fs = (P,WP,XS,X,S,M,B,W)	opcode:	0Ea7
	cycles:	4 + d

Set the fs field of register D to its logical AND with the corresponding field of register C. Carry is not affected.

D=D+1 fs - Increment D
----

fs = A — opcode: E7, cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: Ba7, cycles: 3 + d

Increment the specified fs field of register D by one. Adjusts Carry.

D=D+C fs - Sum of D and C into D
----

fs = A — opcode: C3, cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: Aa3, cycles: 3 + d

Set the specified fs field of register D to the sum of itself and the corresponding field of register C. Adjusts Carry.

D=D+D fs - Sum of D and D into D
----

fs = A — opcode: C7, cycles: 7

fs = (P,WP,XS,X,S,M,B,W) — opcode: Aa7, cycles: 3 + d

Double the specified fs field of register D. Adjusts Carry.

D=D-1 fs - Decrement D
----

fs = A	opcode: CF cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: AaF cycles: 3 + d

Decrement the specified fs field of register D by one. Adjusts Carry.

D=D-C fs - D minus C into D
----

fs = A	opcode: E3 cycles: 7
fs = (P,WP,XS,X,S,M,B,W)	opcode: Ba3 cycles: 3 + d

Set the specified fs field of register D to the difference between itself and the corresponding field of register C. Adjusts Carry.

DAT0=A fsd - Load memory from A
----

fs = A	opcode: 146 cycles: 17
fs = B	opcode: 14E cycles: 14
fs = (P,WP,XS,X,S,M,W)	opcode: 156a cycles: 16 + d
fs = d	opcode: 15Ex (x=d-1) cycles: 15 + d

The amount of data (d nibbles) specified by fsd will be written to the memory address pointed to by D0 from the specified field of register A. The lowest-order nibble of the register field will be written to the lowest-addressed nibble of memory, proceeding toward the higher-order nibbles. If fs = d, d nibbles are written to memory starting from nibble 0 of the register. See the section on "Storing Data Into Memory" earlier in this chapter.

DAT0=C fsd - Store into memory from C

fs = A	opcode: 144 cycles: 17
fs = B	opcode: 14A cycles: 14
fs = (P,WP,XS,X,S,M,W)	opcode: 154a cycles: 16 + d
fs = d	opcode: 15Ax (x=d-1) cycles: 15 + d

The amount of data (d nibbles) specified by fsd will be written to the memory address pointed to by D0 from the specified field of register C. The lowest-order nibble of the register field will be written to the lowest-addressed nibble of memory, proceeding toward the higher-order nibbles. If fs = d, d nibbles are written to memory starting from nibble 0 of the register. See the section on "Storing Data Into Memory" earlier in this chapter.

DAT1=A fs - Store into memory from A

fs = A	opcode: 141 cycles: 17
fs = B	opcode: 149 cycles: 14
fs = (P,WP,XS,X,S,M,W)	opcode: 151a cycles: 16 + d
fs = d	opcode: 159x (x=d-1) cycles: 15 + d

The amount of data (d nibbles) specified by fsd will be written to the memory address pointed to by D1 from the specified field of register A. The lowest-order nibble of the register field will be written to the lowest-addressed nibble of memory, proceeding toward the higher-order nibbles. If fs = d, d nibbles are written to memory starting from nibble 0 of the register. See the section on "Storing Data Into Memory" earlier in this chapter.

DAT1=C fsd - Store into memory from C
----

fs = A	opcode:	145
	cycles:	17
fs = B	opcode:	14D
	cycles:	14
fs = (P,WP,XS,X,S,M,W)	opcode:	155a
	cycles:	16 + d
fs = d	opcode:	15Dx (x=d-1)
	cycles:	15 + d

The amount of data (d nibbles) specified by fsd will be written to the memory address pointed to by D1 from the specified field of register C. The lowest-order nibble of the register field will be written to the lowest-addressed nibble of memory, proceeding toward the higher-order nibbles. If fs = d, d nibbles are written to memory starting from nibble 0 of the register. See the section on "Storing Data Into Memory" earlier in this chapter.

DCEX fs - Exchange Registers D and C
----

fs = A	opcode:	DF
	cycles:	7
fs = (P,WP,XS,X,S,M,B,W)	opcode:	AbF
	cycles:	3 + d

Exchange the fs fields of registers of D and C. Carry is not affected.

DSL fs - D Shift Left
----

fs = A opcode: F3
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: Bb3
cycles: 3 + d

Shift the contents of the specified fs field of register D left one nibble, without affecting the rest of the register. The nibble shifted off the left end of the field is lost. The new low-order nibble of the field is zero. The Sticky Bit (SB) is not affected.

DSLC - D Shift Left Circular
----

opcode: 813
cycles: 21

Circular shift register D left one nibble. Operates on all 16 digits. The Sticky Bit (SB) is not affected.

DSR fs - D Shift Right
----

fs = A opcode: F7
cycles: 7

fs = (P,WP,XS,X,S,M,B,W) opcode: Bb7
cycles: 3 + d

Shift the contents of the specified fs field of register D right one nibble, without affecting the rest of the register. The nibble shifted off the right end of the field is lost, but the Sticky Bit (SB) is set if the nibble was non-zero. The new high-order nibble of the field is zero.

DSRB - D Shift Right Bit

opcode: 81F
cycles: 20

Shift register D right one bit. Operates on all 16 digits. The bit shifted off the end is lost, but the Sticky Bit (SB) is set if it was non-zero. The new high-order bit of the register is zero.

DSRC - D Shift Right Circular

opcode: 817
cycles: 21

Circular shift register D right one nibble. Operates on all 16 digits. The Sticky Bit (SB) is set if the nibble shifted from low-order around to high-order position was non-zero.

GOC label - Go relative on carry

opcode: 4aa (Carry=0)
cycles: 10 (GO)
3 (NO)

Short relative jump to label if Carry is set. label must be in the range:

addr - 128 <= label <= addr + 127

where addr is the address of the second nibble of the opcode. The address offset aa is in two's complement form and is relative to addr.

GOLONG label - Go Long

opcode: 8Caaaa
cycles: 14

Long relative jump to label unconditionally. label must be in the range:

addr - 32768 <= label <= addr + 32767

where addr is the address of the third nibble of the opcode. The address offset aaaa is in two's complement form and is relative to addr.

GONC label - Go relative on no carry

opcode: 5aa (Carry=1)
cycles: 10 (GO)
3 (NO)

Short relative jump to label if Carry is clear. label must be in the range:

addr - 128 <= label <= addr + 127

where addr is the address of the second nibble of the opcode. The address offset aa is in two's complement form and is relative to addr.

GOSBVL label - Gosub very long to label

opcode: 8Faaaaa
cycles: 15

Absolute subroutine jump to aaaaa, which is the absolute address of label. See the GOSUB mnemonic.

GOSUB label - Gosub to label

opcode: 7aaa
cycles: 12

Relative subroutine jump to label. label must be in the range:

addr - 2048 <= label <= addr + 2047

where addr is the starting address of the next instruction. The address offset aaa is in two's complement form and is relative to addr.

As with all subroutine jumps, the address (addr) of the instruction following the gosub opcode is pushed onto the hardware return stack, so that when a corresponding return is executed, control resumes with the instruction at address addr.

As the return address is pushed onto the return stack, the bottom-most address on the stack is discarded. Therefore, the return stack always contains 8 addresses, and if pushes exceed pops by 8 levels, the bottom-most return addresses are lost. Since the interrupt system requires one level to process interrupts, only 7 levels of the return stack can be used by code which must execute when interrupts are enabled. See the RTN mnemonic for further information.

GOSUBL label - Gosub long to label

opcode: 8Eaaaa
cycles: 15

Long relative subroutine jump to label. label must be in the range:

addr - 32768 <= label <= addr + 32767

where addr is the starting address of the next instruction. The address offset aaaa is in two's complement form and is relative to addr. See the GOSUB mnemonic.

GOTO label - Jump relative
----

opcode: 6aaa
cycles: 11

Relative jump to label unconditionally. label must be in the range:

addr - 2048 <= label <= addr + 2047

where addr is the address of the second nibble of the opcode. The address offset aaa is in two's complement form and is relative to addr.

GOVLNG label - Jump very long
----

opcode: 8Daaaaa
cycles: 14

Unconditional jump to aaaaa, which is the absolute address of label.

GOYES label - Jump if Test is True
-----

opcode: yy
cycles: included in the accompaning Test mnemonic cycle time.

GOYES is a mnemonic to specify part of a CPU test opcode. GOYES must always follow a test mnemonic. If the condition of the test is met, a jump is performed to label with Carry set. label must be in the range

addr - 128 <= label <= addr + 127

where addr is the starting address of the jump offset yy. If the test condition is not met, Carry is cleared and control passes to the next instruction. Compare with RTNYES.

INTOFF - Interrupt Off
----

opcode: 808F
cycles: 5

Disable the keyboard interrupt system.

INTON - Interrupt On
----

opcode: 8080
cycles: 5

Enable the keyboard interrupt system. See the "HP-71 Hardware Specification" for more information.

LC(m) n..n - Load C with constant (1<=m<=6)
----

opcode: 3xn..n (x=m-1)
cycles: 3+m

Load m digits of the expression n..n to the C register beginning at the P pointer position, and proceeding toward higher-order nibbles, with the ability to wrap around the register. See the section on "Loading Data From Memory" earlier in this chapter.

LCASC \A..A\ - Load C with ASCII constant
----

opcode: 3mc..c
(m = 2*(# of chars)-1;
c..c = ASCII codes)
cycles: 3+2*(# of chars)

Load up to 8 ASCII characters to the C register beginning at the P pointer position, and proceeding toward higher-order nibbles, with the ability to wrap around the register. Each A represents an

ASCII character. The ASCII characters are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

LCHEX h..h - Load C with hex constant
----

opcode: 3nh..h (n=# of digits-1)
cycles: 4+n

Load up to 16 hex digits into the C register beginning at the P pointer position, and proceeding toward higher-order nibbles, with the ability to wrap around the register. The hex digits are stored in the opcode in reverse order so that when the instruction is executed the data will be loaded into the register with the intended orientation. See the section on "Loading Data From Memory" earlier in this chapter.

MP=0 - Clear Module Pulled bit (MP)
----

opcode: 828
cycles: 3

Clears the Module Pulled bit (MP) and pulls the Module Pulled Interrupt line low. See CLRHST mnemonic.

NOP3 - Three nibble No-op
----

opcode: 420
cycles: 10 (GO/RTNYES)
3 (NO)

This mnemonic generates a GOC or a GONC to the next instruction, effectively skiping three nibbles.

NOP4 - Four nibble No-op
----

opcode: 6300
cycles: 11

This mnemonic generates a GOTO to the next instruction, efectively skiping four nibbles.

NOP5 - Five nibble No-op
----

opcode: 64000
cycles: 11

This mnemonic generates a relative GOTO to +4 nibbles. The fifth nibble in the opcode is a place holder and is jumped over. The mnemonic effectively skips five nibbles.

OUT=C - Load 3 nibbles of OR
-----

opcode: 801
cycles: 6

All nibbles of the Output register are loaded with the low-order three nibbles of C (X field).

OUT=CS - Load 1 nibble of OR
------

opcode: 800
cycles: 4

The least significant nibble of the Output register is loaded with the least significant nibble of the C register.

P=C n - Copy P pointer into C at Nibble n
----

opcode: 80DN
cycles: 6

Copy nibble n of register C into the P pointer.

P=P+1 - Increment P Pointer
----

opcode: 0C
cycles: 3

Increment the P pointer. If P is incremented past F it will automatically wrap around to 0. Adjusts Carry.

P=P-1 - Decrement P Pointer
----

opcode: 0D
cycles: 3

Decrement the P pointer. If P is decremented past 0 it automatically wraps around to F. Adjusts Carry.

P= n - Set P Pointer to n
----

opcode: 2n
cycles: 2

Set the P pointer to n.

R0=A - Copy A to register R0
----

opcode: 100
cycles: 19

The contents of the working register A is copied to the scratch register R0.

R0=C - Copy C to register R0
----

opcode: 108
cycles: 19

The contents of the working register C is copied to the scratch register R0.

R1=A - Copy A to register R1
----

opcode: 101
cycles: 19

The contents of the working register A is copied to the scratch register R1.

R1=C - Copy C to register R1
----

opcode: 109
cycles: 19

The contents of the working register C is copied to the scratch register R1.

R2=A - Copy A to register R2
----

opcode: 102
cycles: 19

The contents of the working register A is copied to the scratch register R2.

R2=C - Copy C to register R2
----

opcode: 10A
cycles: 19

The contents of the working register C is copied to the scratch register R2.

R3=A - Copy A to register R3
----

opcode: 103
cycles: 19

The contents of the working register A is copied to the scratch register R3.

R3=C - Copy C to register R3
----

opcode: 10B
cycles: 19

The contents of the working register C is copied to the scratch register R3.

R4=A - Copy A to register R4
----

opcode: 104
cycles: 19

The contents of the working register A is copied to the scratch register R4.

R4=C - Copy C to register R4
----

opcode: 10C
cycles: 19

The contents of the working register C is copied to the scratch register R4.

RESET - System reset
----

opcode: 80A
cycles: 6

The System Reset Bus Command is issued with all chips performing a local reset. The reset function will vary according to the chip type. See the "HP-71 Hardware Specification" for more information.

RSTK=C - Push C to Return Stack
----

opcode: 06
cycles: 8

Push the low-order 5 nibbles (A field) of the C register onto the Return Stack. See the GOSUB mnemonic.

RTI - Return from interrupt
----

opcode: 0F
cycles: 9

Return and re-enable the interrupt system. See the RTN mnemonic.

RTN - Return
----

opcode: 01
cycles: 9

Return control to the top address on the hardware return stack. The top address on the hardware return stack is popped off and placed in the program counter PC. As the address is popped off the stack, a zero address is inserted at the bottom of the stack.

Therefore the the hardware return stack always contains 8 addresses, and if more pops (returns) than pushes (gosubs) are performed, zeros will be read off the stack. Such an attempt to "return" to address 0 results in a memory reset, since the memory reset code of the operating system resides at address 0.

RTNC -Return on carry
----

opcode: 400
cycles: 10 (RTN)
3 (NO)

Return if Carry is set. See RTN mnemonic.

RTNCC - Return, clear carry
----

opcode: 03
cycles: 9

Return and set Carry. See RTN mnemonic.

RTNNC - Return on no carry
----

opcode: 500 (Carry=1)
cycles: 10 (RTN)
3 (NO)

Return if Carry is not set. See RTN mnemonic.

RTNSC - Return, set carry
----

opcode: 02
cycles: 9

Return and set Carry. See RTN mnemonic.

RTNSXM - Return, set External Module Missing bit (XM)
----

opcode: 00
cycles: 9

Return and set the External Module Missing bit (XM). See the RTN instruction.

RTNYES - Return if Test is True
----

opcode: 00
cycles: included in the accompaning mnemonic cycle time.

If the test condition is not met, Carry is cleared and control passes to the next instruction. Compare with RTNYES. RTNYES is a mnemonic to specify part of a CPU test opcode. RTNYES must always follow a test mnemonic. If the test condition is met, Carry is set and a return is executed. If the test condition is not met, control passes to the instruction following the RTNYES. Compare with the RTN and GOYES mnemonics.

SB=0 - Clear Sticky Bit (SB)
----

opcode: 822
cycles: 3

Clear the Sticky Bit (SB). See CLRHST mnemonic.

SETDEC - Set decimal
----

opcode: 05
cycles: 3

Set CPU arithmetic mode to decimal.

SETHEX - Set hexadecimal mode
----

opcode: 04
cycles: 3

Set CPU arithmetic mode to hexadecimal.

SHUTDN - System Shutdown
----

opcode: 807
cycles: 5

When this instruction is executed the CPU sends out the Shutdown Bus Command and stops its clock. See the "HP-71 Hardware Specification" for more information.

SR=0 - Clear Service Request bit (SR)
----

opcode: 824
cycles: 3

Clear the Service Request bit (SR). See the CLRHST instruction.

SREQ? - Service Request
----

opcode: 80E
cycles: 7

This instruction sets the Service Request bit (SR) if any chip on the system bus requests service. When this instruction is executed, a Service Request Bus Command is issued on the system bus to poll all chips for a Service Request. If any chip requests service, a bus line will be pulled high during the next strobe following the Service Request Bus Command. This value of the bus will be latched into the least significant nibble of the C register. The bus line pulled high determines the device type (Timer, HPIL, et cetera). If any bus line is high, the Service Request bit (SR) will be set. See the "HP-71 Hardware Specification" for more information. See also the ?SREQ and SR=0 mnemonics.

ST=0 n - Clear Program Status bit n
----

opcode: 84n
cycles: 4

Clear the Program Status bit selected by n.

ST=1 n - Set Program Status bit n
----

opcode: 85n
cycles: 4

Set the Program Status bit selected by n.

ST=C - C to Status
----

opcode: 0A
cycles: 6

Copy the low-order 12 bits of the Status register (X field) into the low-order 12 bits of the C register.

UNCNFG - Unconfigure
----

opcode: 804
cycles: 12

Load the low-order 5 nibbles (A field) of the C register into each Data pointer with the device addressed by the Data pointer unconfiguring.

XM=0 - Clear External Module Missing bit (XM)
----

opcode: 821
cycles: 3

Clear the External Module Missing bit (XM). This bit is set by the RTNSXM instruction. See the CLRHST instruction.

```
+--------------------------------------------------+------------------+
|                                                  |                  |
|   HP-71 CODE EXAMPLES                            |  CHAPTER   17    |
|                                                  |                  |
+--------------------------------------------------+------------------+
```

17.1 Machine Code Packing Techniques

1. Take full advantage of existing subroutines, or create beneficial new ones (even short ones).

2. Use A field instead of B field when possible and when speed doesn't matter.

3. Remove unnecessary P=0 instructions. (Most routines exit with pointer set to 0.)

4a. If two subroutines have common ending, then end one of them with a GOTO to the common ending.

4b. If common code precedes subroutine calls, move that common code to the front end of the subroutine.

5. A RTN should generally not follow a GOSUB instruction.

6. Shorten error messages or delete redundant ones.

7. Remove unnecessary long branches within modules.

8. Code for optimum space, not speed, if speed loss is not significant.

9. Centralize the loading of C with the same Error Number.

10. Setting a data pointer to the same 5 nibble value can be shortened using a GOSUB to set it.

11. Using an LC(5) to load a 1 or 2 nibble constant into C(A) can be shortened with:

```
        C=0     A
        LC(2)   =symbol
```

12. Using load listing, search for routines that are

never referenced, or only referenced once.

13. Using load listing as an aid, search for common sequences of subroutine calls.

14. If the state of the carry is predictably the same for all paths through a section of code, any GOTO instruction to a nearby label can be replaced by a GOC or GONC instruction depending on whether the carry is known to be set or clear, respectively. This type of branch saves 1 nibble, and is referred to as a "Branch Every Time," often abbreviated as "BET" or "B.E.T." in the comment field of the instruction. Such branches should be used with caution and should be clearly commented.

17.2 Mainframe File Type Table

The mainframe file type table is as follows. For an explanation of the format of this table, see the "Table Formats" chapter.

```
=FTYPE
*** DATA FILE (Interchange DATA File)
       NIBHEX 110
       CON(2) =oDAsod
       NIBASC \DATA \
       CON(1) 2
       CON(4) =fDATA
       CON(4) (=fDATA)+1     Secure DATA file
*** BASIC FILE
       NIBHEX 001
       CON(2) =oBSsod
       NIBASC \BASIC\
       CON(1) 4
       CON(4) =fBASIC
       CON(4) (=fBASIC)+1    Secure BASIC
       CON(4) (=fBASIC)+2    Private BASIC
       CON(4) (=fBASIC)+3    Secure, private BASIC
*** KEY FILE
       NIBHEX 000
       CON(2) =oKYsod
       NIBASC \KEY  \
       CON(1) 2
       CON(4) =fKEY
       CON(4) (=fKEY)+1      Secure KEYS
*** TEXT FILE
       NIBHEX 440
       CON(2) =oTXsod
       NIBASC \TEXT \
```

```
        CON(1) 2
        CON(4) 1
        CON(4) #E0D1         Secure TEXT
*** LIF1 FILE (same as TEXT)
        NIBHEX 440
        CON(2) =oTXsod
        NIBASC \LIF1 \
        CON(1) 1
        CON(4) 1
*** SDATA FILE (Series 40 Data File)
        NIBHEX 220
        CON(2) =o41sod
        NIBASC \SDATA\
        CON(1) 1
        CON(4) #E0D0
*** BIN FILE (Binary File)
        NIBHEX 001
        CON(2) =oBNsod
        NIBASC \BIN  \
        CON(1) 4
        CON(4) =fBIN
        CON(4) (=fBIN)+1     Secure BIN
        CON(4) (=fBIN)+2     Private BIN
        CON(4) (=fBIN)+3     Secure, private BIN
*** LEX FILE (Langauge Extension File)
        NIBHEX 001
        CON(2) =oLXsod
        NIBASC \LEX  \
        CON(1) 4
        CON(4) =fLEX
        CON(4) (=fLEX)+1     Secure LEX
        CON(4) (=fLEX)+2     Private LEX
        CON(4) (=fLEX)+3     Secure, private LEX
****
*
        NIBHEX FF             Terminates Table
*
        END
```

17.3 LEX File Implementing Statements and Functions

This LEX file is taken from the HP-71 Editor ROM. It implements the statements INSERT#, REPLACE#, and DELETE# for TEXT files, and extends the LIST and PLIST statements to include TEXT files. In addition, a number of functions are also implemented to examine and search TEXT files, to detect the pressing of scroll keys, and to aid the parsing of Editor commands.

```
        TITLE Titan EDITOR Lexfile <840101.1823>
        REL    #8
*
*   TTTTT    III     &      EEEEE    DDDD     TTTTT
*     T       I     & &     E         D  D      T
*     T       I     & &     E         D  D      T
*     T       I      &      EEEE      D  D      T
*     T       I     & & &   E         D  D      T
*     T       I     &   &   E         D  D      T
*     T      III     && &   EEEEE    DDDD       T
*
************************************************************
*  Set assembler flag1 = 0 to assemble the complete
*  Text Editor, with Formatter.
*
*  Set assembler flag1 = 1 to assemble the short
*  Text Editor, without Formatter.
************************************************************
*
        RDSYMB SB%RAM
        RDSYMB TI%EQU
        NIBASC \EDLEX   \       File Name
        CON(4) =fLEX            File Type
        NIBHEX 00               Flags
        NIBHEX 1441             Time
        NIBHEX 412138           Date
        REL(5) FILEND           File Length
*
        NIBHEX 0F               Id
        CON(2)   1              Lowest Token
        CON(2)   7              Highest Token
        REL(5) SCRLEX           End of lex table chain
*
        NIBHEX F                Speed table omitted
        CON(4) (TxTbSt)+1-(*)   Offset to text table
        REL(4)  MSGTBL          Offset to message table
        REL(5)  POLHND          Offset to poll handler
        STITLE M a i n   T a b l e
*  Main Table
=xromF0
*
        CON(3)    0             01  DELETE#
        REL(5) DELETE
        NIBHEX D
*
        CON(3)   34             02  EDTEXT
        REL(5) EDTEXT
        NIBHEX D
*
        CON(3)   49             03  FILESZR
        REL(5) FILSZR
```

```
        NIBHEX F
*
        CON(3)   66              04   INSERT#
        REL(5) INSERT
        NIBHEX D
*
        CON(3)   81              05   REPLACE#
        REL(5) REPLCE
        NIBHEX D
*
        CON(3)   98              06   SEARCH
        REL(5) SEARCH
        NIBHEX F
*
        CON(3)   15              07   EDPARSE$
        REL(5) EDPARS
        NIBHEX F
        STITLE T e x t   T a b l e
*  Text Table
TxTbSt                          Text table start
*
        NIBHEX B                 DELETE#
        NIBASC \DELETE\
        NIBHEX 10
*
        NIBHEX F                 EDPARSE$
        NIBASC \EDPARSE$\
        NIBHEX 70
*
        NIBHEX B                 EDTEXT
        NIBASC \EDTEXT\
        NIBHEX 20
*
        NIBHEX D                 FILESZR
        NIBASC \FILESZR\
        NIBHEX 30
*
        NIBHEX B                 INSERT#
        NIBASC \INSERT\
        NIBHEX 40
*
        NIBHEX D                 REPLACE#
        NIBASC \REPLACE\
        NIBHEX 50
*
        NIBHEX B                 SEARCH
        NIBASC \SEARCH\
        NIBHEX 60
TxTbEn  NIBHEX 1FF              Text termination
*
        STITLE Editor messages
```

```
*
 MSGTBL

 frmt-  IF     1                 Short msg table w/o formatter

* -------------- merge MB&EDS here ---------------
*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*!!!!!!!!! Message number 5 is placed first because of the    !!
*!!!!!!!!! requirement to have a 0 nibble following the       !!
*!!!!!!!!! range field.  If message number 5 changes,         !!
*!!!!!!!!! you must select another message to put in the      !!
*!!!!!!!!! first slot!  (any message with a length which      !!
*!!!!!!!!! is a multiple of 16)                               !!
        CON(2)   1        Min message #
        CON(2)  11        Max message #
*
=eLINE  EQU      5           Line
        CON(2)  16
        CON(2)   5           Message number   5
        CON(1)   4
        NIBASC \Line \
        CON(1)  12
*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*
=eEOF   EQU      1           Eof
        CON(2)  12
        CON(2)   1           Message number   1
        CON(1)   2
        NIBASC \Eof\
        CON(1)  12
*
=ecmds  EQU      2           CDEFHILMPRST
        CON(2)  31
        CON(2)   2           Message number   2
        CON(1)  11
        CON(1)  11
        NIBASC \CDEFHILM\
        NIBASC \PRST\
        CON(1)  12
*
=eUNKNW EQU      3           ? Cmd:
        CON(2)  13
        CON(2)   3           Message number   3
        CON(1)   1
        NIBASC \? \
        CON(1)  13
        CON(2) =eCMD
        CON(1)  12
*
=eFLNM  EQU      4           Filename:
        CON(2)  19
```

```
        CON(2)   4           Message number   4
        CON(1)  14
        CON(2) =eFILE
        CON(1)   4
        NIBASC \name:\
        CON(1)  12
*
* Message number 5 is found at the top of the table.
*
*
=eCMD   EQU      6           Cmd:
        CON(2)  14
        CON(2)   6           Message number   6
        CON(1)   3
        NIBASC \Cmd:\
        CON(1)  12
*
=eOKDLT EQU      7           OK to Delete? Y/N:
        CON(2)  43
        CON(2)   7           Message number   7
        CON(1)  10
        NIBASC \OK to De\
        NIBASC \let\
        CON(1)   6
        NIBASC \e? Y/N:\
        CON(1)  12
*
=eYNQ   EQU      8           YNQ
        CON(2)  12
        CON(2)   8           Message number   8
        CON(1)   2
        NIBASC \YNQ\
        CON(1)  12
*
=eINVCM EQU      9           Invalid Cmd Strg
        CON(2)  25
        CON(2)   9           Message number   9
        CON(1)  14
        CON(2) =eINVLD
        CON(1)   7
        NIBASC \Cmd Strg\
        CON(1)  12
*
=eWRKG  EQU     10           Working...
        CON(2)  26
        CON(2)  10           Message number  10
        CON(1)   9
        NIBASC \Working.\
        NIBASC \..\
        CON(1)  12
*
```

```
*eDONE  EQU      11           Done
        CON(2)   14
        CON(2)   11           Message number  11
        CON(1)    3
        NIBASC \Done\
        CON(1)   12
*
*
        NIBHEX FF             Table terminator

 frmt-  ELSE                  Short msg table w/o formatter

*--------------- merge MB&EDM here -------------
*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*!!!!!!!!! Message number 5 is placed first because of the      !!
*!!!!!!!!! requirement to have a 0 nibble following the         !!
*!!!!!!!!! range field.  If message number 5 changes,           !!
*!!!!!!!!! you must select another message to put in the        !!
*!!!!!!!!! first slot!  (any message with a length which        !!
*!!!!!!!!! is a multiple of 16)                                 !!
        CON(2)   1       Min message #
        CON(2)  53       Max message #
*
*eLINE  EQU      5            Line
        CON(2)  16
        CON(2)   5            Message number   5
        CON(1)   4
        NIBASC \Line \
        CON(1)  12
*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*
*eEOF   EQU      1            Eof
        CON(2)  12
        CON(2)   1            Message number   1
        CON(1)   2
        NIBASC \Eof\
        CON(1)  12
*
*ecmds  EQU      2            CDEFHILMPRST
        CON(2)  31
        CON(2)   2            Message number   2
        CON(1)  11
        CON(1)  11
        NIBASC \CDEFHILM\
        NIBASC \PRST\
        CON(1)  12
*
*eUNKNW EQU      3            ? Cmd:
        CON(2)  13
        CON(2)   3            Message number   3
        CON(1)   1
```

```
        NIBASC \? \
        CON(1)  13
        CON(2) =eCMD
        CON(1)  12
*
=eFLNM  EQU      4            Filename:
        CON(2)  19
        CON(2)   4            Message number   4
        CON(1)  14
        CON(2) =eFILE
        CON(1)   4
        NIBASC \name:\
        CON(1)  12
*
* Message number 5 is found at the top of the table.
*
*
=eCMD   EQU      6            Cmd:
        CON(2)  14
        CON(2)   6            Message number   6
        CON(1)   3
        NIBASC \Cmd:\
        CON(1)  12
*
=eOKDLT EQU      7            OK to Delete? Y/N:
        CON(2)  43
        CON(2)   7            Message number   7
        CON(1)  10
        NIBASC \OK to De\
        NIBASC \let\
        CON(1)   6
        NIBASC \e? Y/N:\
        CON(1)  12
*
=eYNQ   EQU      8            YNQ
        CON(2)  12
        CON(2)   8            Message number   8
        CON(1)   2
        NIBASC \YNQ\
        CON(1)  12
*
=eINVCM EQU      9            Invalid Cmd Strg
        CON(2)  25
        CON(2)   9            Message number   9
        CON(1)  14
        CON(2) =eINVLD
        CON(1)   7
        NIBASC \Cmd Strg\
        CON(1)  12
*
=eWRKG  EQU     10            Working...
```

```
        CON(2)  26
        CON(2)  10              Message number  10
        CON(1)   9
        NIBASC \Working.\
        NIBASC \..\
        CON(1)  12
*
=eDONE  EQU     11              Done
        CON(2)  14
        CON(2)  11              Message number  11
        CON(1)   3
        NIBASC \Done\
        CON(1)  12
*
=efrmt  EQU     12
        CON(2)  68
        CON(2)  12              Message number  12
        CON(1)  10
        NIBASC \ESPNPASP\
        NIBASC \ADF\
        CON(1)  10
        NIBASC \ICOCEJUM\
        NIBASC \ASK\
        CON(1)   7
        NIBASC \TADLPLME\
        CON(1)  12
*
=e2MFL  EQU     13              Merge > 5 Files
        CON(2)  32
        CON(2)  13              Message number  13
        CON(1)   9
        NIBASC \Merge > \
        NIBASC \5 \
        CON(1)  14
        CON(2)  =eFILE
        CON(1)   0
        NIBASC \s\
        CON(1)  12
*
=eMULT  EQU     14              Multiple Distribution Lists
        CON(2)  41
        CON(2)  14              Message number  14
        CON(1)  10
        NIBASC \Mult Dis\
        NIBASC \tr \
        CON(1)   5
        NIBASC \Lists:\
        CON(1)  12
*
=ePLSIN EQU     15              Insert Page...
        CON(2)  35
```

```
        CON(2)  15           Message number  15
        CON(1)  11
        CON(1)  13
        NIBASC \Insert P\
        NIBASC \age...\
        CON(1)  12
*
*
******* HELP list ************************
*                            Copy: [b[e]] C [<file>]
*
=eCOPY  EQU     16
        CON(2)  17
        CON(2)  16           Message number  16
        CON(1)  13
        CON(2) =ebes
        CON(1)   0
        NIBASC \C\
        CON(1)  13
        CON(2) =e<file
        CON(1)   0
        NIBASC \]\
        CON(1)  12
*                            Delete: [b[e]] D [<file>[+]]
*
=eDELT  EQU     17
        CON(2)  23
        CON(2)  17           Message number  17
        CON(1)  13
        CON(2) =ebes
        CON(1)   0
        NIBASC \D\
        CON(1)  13
        CON(2) =e<file
        CON(1)   3
        NIBASC \[+]]\
        CON(1)  12
*                            Exit: E
*
=eEXIT  EQU     18
        CON(2)   8
        CON(2)  18           Message number  18
        CON(1)   0
        NIBASC \E\
        CON(1)  12
*                            Format: F [n][G$]
*
=eFORMT EQU     19
        CON(2)  24
        CON(2)  19           Message number  19
        CON(1)   8
```

```
        NIBASC \F [n][G$\
        NIBASC \]\
        CON(1)  12
*                                   Help: H
*
=eHELP  EQU     20
        CON(2)  8
        CON(2)  20                  Message number  20
        CON(1)  0
        NIBASC \H\
        CON(1)  12
*                                   Insert: [1] I
*
=eINSRT EQU     21
        CON(2)  16
        CON(2)  21                  Message number  21
        CON(1)  4
        NIBASC \[1] I\
        CON(1)  12
*                                   List: [b[e]] L [n][N]
*
=eLIST  EQU     22
        CON(2)  14
        CON(2)  22                  Message number  22
        CON(1)  13
        CON(2)  =ebes
        CON(1)  0
        NIBASC \L\
        CON(1)  13
        CON(2)  =enN
        CON(1)  12
*                                   Move: [b[e]] M [<file>]
*
=eMOVE  EQU     23
        CON(2)  17
        CON(2)  23                  Message number  23
        CON(1)  13
        CON(2)  =ebes
        CON(1)  0
        NIBASC \M\
        CON(1)  13
        CON(2)  =e<file
        CON(1)  0
        NIBASC \]\
        CON(1)  12
*                                   Print: [b[e]] P [n][N]
*
=ePRINT EQU     24
        CON(2)  14
        CON(2)  24                  Message number  24
        CON(1)  13
```

```
        CON(2) =ebes
        CON(1)   0
        NIBASC \P\
        CON(1)  13
        CON(2) =enN
        CON(1)  12
*
*                                  Replace: [b[e]][?] R/str1/str2[/]
=eREPLC EQU     25
        CON(2)  38
        CON(2)  25                 Message number  25
        CON(1)  13
        CON(2) =ebe?
        CON(1)  11
        CON(1)  13
        NIBASC \R/str1/s\
        NIBASC \tr2[/]\
        CON(1)  12
*
*                                  Search: [b[e]][?] S/str[/]
=eSEARC EQU     26
        CON(2)  25
        CON(2)  26                 Message number  26
        CON(1)  13
        CON(2) =ebe?
        CON(1)   7
        NIBASC \S/str[/]\
        CON(1)  12
*
*                                  Text: [1] T
=eTEXT  EQU     27
        CON(2)  16
        CON(2)  27                 Message number  27
        CON(1)   4
        NIBASC \[1] T\
        CON(1)  12
*                                  ^ad: advance page
**************************************************
* Building blocks
*
*                                  [b[e]]
=ebes   EQU     50
        CON(2)  20
        CON(2)  50                 Message number  50
        CON(1)   6
        NIBASC \[b[e]] \
        CON(1)  12
*
*                                  [<file>
=e<file EQU     51
        CON(2)  22
```

```
        CON(2)  51              Message number  51
        CON(1)   7
        NIBASC \ [<file>\
        CON(1)  12
*
*                               [b[e]][?]
=ebe?   EQU     52
        CON(2)  26
        CON(2)  52              Message number  52
        CON(1)   9
        NIBASC \[b[e]][?\
        NIBASC \] \
        CON(1)  12
*
*                               [n][N]
=enN    EQU     53
        CON(2)  20
        CON(2)  53              Message number  53
        CON(1)   6
        NIBASC \ [n][N]\
        CON(1)  12
*
        NIBHEX FF               Table terminator

 frmt-  ENDIF                   Short msg table w/o formatter.

* Poll handler goes here.  Handler for VER$ poll is
* provided
*
 POLHND ?B=0    B               VER$ poll?
        GOYES   hVER$0          Yes.
        GONC    hVER$2          No.  To hVER$2 w/carry clear.
 hVER$0 C=R3
        D1=C
        A=R2
        D1=D1- (VER$en)-(VER$st)-2
        CD1EX
        ?A>C    A
        GOYES   hVER$1
        D1=C
        R3=C
*
**!! LCASC text to be returned for VER$ here
* Include a leading blank!!
*
 VER$st LCASC  \ EDT:A\
 VER$en DAT1=C (VER$en)-(VER$st)-2
 hVER$1 RTNSXM
*
**!! Continue poll handler here: Carry is clear, VER$ poll
*  has been handled.
```

```
*
 hVER$2 LC(2)  =pLIST2
        ?B=C   B
        GOYES  LIST00
        RTNSXM
*
 LIST00 LC(5)  =fTEXT
        ?A=C   A
        GOYES  list01
        RTNSXM
 list01 GOTO   LIST01
*
**!! LEXFILE code goes here
*
        STITLE EDTEXT Keyword Execute
*****************************************************************
*****************************************************************
**
** Name:         EDTEXT  -  EDTEXT Keyword Execute
**
** Category:     STEXEC
**
** Purpose:
**       Executes EDTEXT keyword
**
** Entry:
**       P       =  0
**       DO past tEDTEXT (at tLITRL)
**
** Exit:
**       P       =  0
**
** Calls:        See CALL statement execute
**
** Uses.......
**               See CALL statement execute
**
** Stk lvls:     See CALL
**
** History:
**
**    Date      Programmer               Modification
**  --------    ----------    ----------------------------------
**  09/28/83    S.W.          Added documentation
**
*****************************************************************
*****************************************************************
*
        REL(5) EDTXTd          Offset to EDTEXT decompile
        REL(5) EDTXTp          Offset to EDTEXT parse
 EDTEXT GOVLNG =CALL
```

```
*
        STITLE FILESZR Function Execute
****************************************************************
****************************************************************
**
** Name:        FILSZR  -  FILSZR Function Execute
**
** Category:    FNEXEC
**
** Purpose:
**      FILSZR locates the specified TEXT file in and returns
**      the number of records in the file.  The syntax is:
**
**          FILSZR ( <file specifier string> )
**
**      The returned value is:
**
**          If >= 0   Number of records in TEXT file
**          If <  0   Negative of the error number.  Possible
**                      errors are:
**                         Invalid File Spec
**                         File Not Found
**                         Invalid File Type
**                         File Protect
**                         Illegal Access
**
** Entry:
**      String specifying file is on stack
**      P      = 0
**
** Exit:
**      P      = 0
**
** Calls:       FILXQ$,FINDF+,POSTXT,HDFLT,D1MSTK,PSHSTK,FNRTN1
**
** Uses.......
**  Exclusive: A,B(A),B(S),C,D,R0,R1,D0,D1,sPRBLM,sEOF,sBADRC
**  Inclusive: A-D,R0-R3,D0,D1,S11-S0,STMTR1,STMTD1,Function
**             Scratch
**
** Stk lvls:   6 (FILXQ$)
**
** History:
**
**    Date      Programmer           Modification
**  --------    ----------    ---------------------------------
**  09/29/83    FH            Designed and coded
**
****************************************************************
****************************************************************
sPRBLM EQU    4
```

```
        NIBHEX 411              Parameter descriptor: One string
FILSZR  CD0EX                   Save D0 in RSTKBF
        RSTK=C                   .
        GOSBVL =R<RSTK           .
        GOSUB  ave=d1           Save string start in AVMEME
        GOSBVL =FILXQ$          Pop file name into A
        GOC    FILS10           Branch if valid file name
        GOSUB  d1mstk           Restore poor old D1
        GOSBVL =POP1S           Move D1 past string header
        CD1EX                   Move D1 past string contents
        C=C+A  A                 .
        D1=C                     .
        GOSUB  ave=d1           Save D1 in AVMEME
        LC(4)  =eFSPEC          Load error code
FILSer  ST=1   sPRBLM           Set error flag
        CSL    A                Isolate error code in A
        CSR    A                 .
        A=C    A                 .
        GOTO   FILS20           Go return negative error code
FILS10  GOSUB  ave=d1           Save D1 in AVMEME (string popped)
        GOSBVL =FINDF+          Find file
        GOC    FILSer           Error if not found
        A=0    A                Search for unreachable record
        A=A-1  A                 . which is -1
        R1=A                     .
        AD1EX                   A = File header addr
        GOSUB  POSTXT           Position file, always error return
        ST=0   sPRBLM
        ?C#0   A                Real error, not just EOF?
        GOYES  FILSer            .
        A=R0                    Compute # records in file
        A=A+1  A                 .
FILS20  GOSUB  d1mstk           Restore D1
        GOSBVL =HDFLT           Convert to decimal
        ?ST=0  sPRBLM           No error?
        GOYES  FILS30
        A=-A-1 S                Make number negative
FILS30  SETHEX                  Return to hex mode
        D1=D1- 16               Write number to stack
        DAT1=A W
        GOSBVL =RSTK<R          Restore D0
        C=RSTK                   .
        CD0EX                    .
expr    GOVLNG =EXPR            Return

        STITLE DELETE# Keyword Execute

        REL(5) DELETd           Off set to DELETE# decompile
        REL(5) DELETp           Off set to DELETE# parse
DELETE  ST=0   sINS             Set up DELETE# status
```

```
        ST=1    sDEL
        GOTO    REPL10

        STITLE INSERT# Keyword Execute

        REL(5) INSRTd          Offset to INSERT# decompile
        REL(5) INSRTp          Offset to INSERT# parse
INSERT  ST=1   sINS            Set up INSERT# status
        ST=0   sDEL
        GOTO   REPL10

        STITLE REPLACE# Keyword Execute
**************************************************************
**************************************************************
**
** Name: REPLCE  -  REPLACE# Statement Execute
**
** Category:   STEXEC
**
** Purpose:
**      Execute REPLACE# statement for the HP-71 EDITOR ROM.
**
** Entry:
**      P       =  0
**      DO      @  After starting token
**
** Exit:
**      P       =  0
**      To NXTSTM
**
** Calls:       OBCOLL,STATSV,GETCH#,MGOSUB,EXPEX-,D1MSTK,POP1R,
**              FLTDH,STATRS,REVPOP,SWPBYT,MOVEU0,POSFIL,BSERR,
**              ?PRFI+,RPLLIN,FIBADR,NXTSTM
**
** Uses.......
**  Exclusive: A-D,R1,R3,sINS,sDEL,STMTR1,STMTD1,CHN#SV
**  Inclusive: A-D,R0-R4,S11-S0,Statement and Function scratch,
**             CHN#SV
**
** Stk lvls:   6 (GETCH#)
**
** Detail:
**   Status usage:       sINS     sDEL
**
**      REPLACE#          0        0
**      INSERT#           1        0
**      DELETE#           1        1
**
**   Statement Scratch usage:
**
**      STMTR0(15-14)         Channel number
```

```
**      STMTRO(9-5)             Record number
**
**
**
** Algorithm:
**      Set up respective status bits (see detail above)
**      Collapse output buffer
**      Evaluate channel number, exit if error
**      Skip comma token
**      Evalute record number, exit if error
**      Save record number in STMTRO
**      If not DELETE#, then
**        Skip ";" token
**        Evaluate string expression
**        Move string from MTHSTK to Output Buffer
**      If file is not in RAM, or is secure or private then error
**      If file is not TEXT copy code, then error
**      Space file out to <record number> giving START of line
**      If INSERT# then
**        Set LENGTH to 0
**      Else
**        Compute line length giving LENGTH
**      Call RPLLIN to edit file
**      Update FIB end-of-data field
**      Collapse output buffer
**      Go to NXTSTM
**
** History:
**
**     Date       Programmer           Modification
**   --------     ----------     -------------------------------------
**   09/12/83     F. Hall        Designed and coded
**
*****************************************************************
*****************************************************************

***
*  Status Symbols
*
 sINS   EQU     11
 sDEL   EQU     9
 sBADRC EQU     8
*sI/OBF EQU     10                  External symbol
*
 ave=d1 GOVLNG  =AVE=D1
 dimstk GOVLNG  =D1MSTK
 swpbyt GOVLNG  =SWPBYT
 mgosub GOVLNG  =MGOSUB
*
 invarg LC(4)   =eIVARG
        GOTO    bserr
```

```
        REL(5) REPLCd          Offset to REPLACE# decompile
        REL(5) REPLCp          Offset to REPLACE# parse
 REPLCE ST=0   sINS            Set up REPLACE# status
        ST=0   sDEL
*
**      Collapse output buffer
**      Evaluate channel number, exit if error
**      Skip comma token
**      Evalute record number, exit if error
*
 REPL10 GOSBVL =OBCOLL         Collapse output buffer
        D1=(5) =S-R0-1         Save status in S-R0-1
        GOSBVL =STATSV          .
        GOSUB  mgosub          Get channel #, save in CHN#SV
        CON(5) =GETCH#          .
        D0=D0+ 2               Skip comma
        GOSUB  mgosub          Get record number
        CON(5) =EXPEX-          .
        GOSUB  dlmstk           . (D1 not valid after MGOSUB here)
        GOSUB  poplr            .
        GOSUB  fltdh            .
        GONC   invarg          Branch if out of range
*
**      If not DELETE#, then
**        Skip ";" token
**        Evaluate string expression
**        Move string from MTHSTK to Output Buffer
*
        D1=(5) =S-R0-1         Restore status momentarily
        GOSBVL =STATRS          .
        ?ST=1  sDEL            DELETE# ?
        GOYES  REPL20           .
        D1=D1- 5               Store record # in S-R0-0
        DAT1=A A                .
        D0=D0+ 2               Skip over ;
        GOSUB  mgosub          Evaluate string expression
        CON(5) =EXPEX-
        GOSUB  dlmstk          D1 @ Av mem end
        GOSBVL =REVPOP         Pop record string (must be reversed)
        B=0    W               B = # bytes of data
        B=A    A                .
        BSRB                    .
        CD1EX                  C = stg start
        D1=(5) =AVMEME         Move AVMEME beyond end of string
        A=A+C  A                .
        DAT1=A A                .
        D0=C                   D0 @ Line header = SOURCE START
        D0=D0- 4                .
        A=B    A               Write line header
        GOSUB  swpbyt           .
```

```
        DAT0=A 4
        B=B+1  A              .
        BSRB                  Round up to even bytes for LIF std
        B=B+1  A              .
        B=B+B  A              Add 2 bytes for line header
        B=B+B  A              Convert to nibs
        D1=D1- (AVMEME)-(AVMEMS) Update AVMEMS to new end
        A=DAT1 A              . (B = BLOCK LENGTH)
        A=A+B  A                .
        DAT1=A A                .
        A=A-B  A              D1 = DEST START
        D1=A                  .
        GOSBVL =MOVEU0        Move string down to output buffer
        D1=(5) =S-R0-0        Recall record number
        A=DAT1 A              .
        D1=D1+ 5              Restore status
        GOSBVL =STATRS
*
**      Space file out to <record number> giving START of line
**      If file is not in RAM, or is secure or private then error
**      If file is not TEXT copy code, then error
*
REPL20 R1=A                   Store record number
       D1=(5) =CHN#SV         Recall channel #
       A=DAT1 B               .
       GOSUB  POSFIL          Position file to requested record
       GONC   REPL30          Branch if no error
       ?C#0   A               Problem not simply EOF?
       GOYES  bserr           .
REPL25 LC(4)  =eEOFIL         Error out "End of File"
       GONC   bserr           . (BET)
REPL30 A=B    S               Check if file secure
       GOSBVL =?PRFI+         .
       GOC    bserr           .
       D=D-1  S               File in MAIN?
       GOC    REPL40
       D=D-1  S               File in IRAM?
       GOC    REPL40
       LC(4)  =eFACCS         "Invalid Access"
bserr  GOVLNG =BSERR
REPL40 ?ST=1  sBADRC          Refuse to edit bad record
       GOYES  REPL25
*
**      If INSERT# then
**        Set LENGTH to 0
**      Else
**        Compute line length giving LENGTH
**      Call RPLLIN to edit file
**      Update FIB end-of-data field
**      Collapse output buffer
**      Go to NXTSTM
```

```
*
        CD1EX                   C,RSTK = Start of line
        RSTK=C                   .
        AD0EX                   A = Start of NEXT line
        ?ST=0   sINS            Not INSERT# ?
        GOYES   REPL50
        A=C     A               Preset LENGTH = 0
 REPL50 C=A-C   A               R3 = LENGTH of previous line
  *     R3=C                     .
        D1=(5)  =STMTD1         Set C = File header address
        C=DAT1  A                .
        D1=C                     .
        D1=D1+  oFBEGb           .
        C=DAT1  A                .
        GOSUB   mgosub          Replace line
        CON(5)  =RPLLIN          .
        GONC    REPL60          Branch if no error
        B=C     A               Pop stack, protecting error code
        C=RSTK                   .
        C=B     A                .
        GOTO    bserr
 REPL60 D1=(5)  =CHN#SV         Update FIB's current position
        A=DAT1  A                .
        GOSBVL  =FIBADR          .
        D1=D1+  16               .
        D1=D1+  (oDBEGb)-16      .
        A=DAT1  A                .
        C=RSTK                   . recall abs address of line start
        C=C-A   A                . make relative to data start
        D1=D1+  16               .
        D1=D1+  (oCPOSb)-(oDBEGb)-16 .
        DAT1=C  A                .
        D1=D1+  (oDLENb)-(oCPOSb) Update data length
        C=DAT1  A                 . C = data length
        A=R3                      . A = offset
        A=A+C   A                 .
        DAT1=A  A                 .
        GOSBVL  =OBCOLL         Collapse output buffer
        GOLONG  nxtstm          Exit to next statement
LIST    STITLE LIST Statement Execute
*****************************************************************
*****************************************************************
**
** Name:        LISTTX  -  LIST of TEXT files
**
** Category:    STEXEC
**
** Purpose:
**         Handles POLL to LIST a TEXT file
**
** Entry:
```

```
**      P       = 0
**      B(B) contains poll#
**      A(A) contains file type#
**      D1 points to file header
**      D0 past file specifier
**
** Exit:
**      P       = 0
**
** Calls:       DECHEX, FRCRDn, RCDSKP, FILSKP, PRPSND, POPUPD
**
** Uses.......
**  Exclusive: A-D, D1,D0, R0-R3, S-R1-1, OUTBS
**
** Stk lvls:    6
**
** History:
**
**    Date      Programmer              Modification
**  --------    ----------    ------------------------------------
**  09/14/83    S.W.          Wrote routine
**
*************************************************************
*************************************************************
*
 LIST01 C=0     W
        A=0     W
        C=C+1   A
        R1=C
        LCHEX   1048575         Biggest# = 5 hex digits
        R3=C
        A=DATO  B
        LC(2)   =tCOMMA
        ?A#C    B
        GOYES   LIST07          No parms specified?
        D0=D0+  1
        A=DATO  A
        ASR     A
        D0=D0+  5
        R1=A                    Write over default parm
        A=DATO  B
        LC(2)   =tCOMMA
        ?A#C    B
        GOYES   LIST06          No 2nd parm specified?
*
 LIST05 D0=D0+  2
        A=DATO  4
        R3=A
        C=R1
        ?C<=A   A               parm1<=parm2 ?
        GOYES   LIST07
```

```
        LC(4)  =eIVARG
        RTNSC
*
LIST06 A=R1
        R3=A
* BCD line#s in R1 & R3
* D1 positioned to start of file
* Convert R1 & R3 to HEX
LIST07 A=R1
        GOSBVL =DECHEX          A(A) contains HEX
        R1=A
        A=R3
        GOSBVL =DECHEX
        R3=A
*
        AD1EX
        GOSBVL =FILSK+
* D1 at file length field
* C(A) at end of file
        D=C     A               File end
*
        D1=D1+ 5                Step over file length field
        CD1EX
        R2=C                    Ptr to SOD (Start of Data)
*
        A=R1
        GOSUB   FRCRDn          Find 1st record to list
        GONC    LIST30          Record found?
*
        XM=0
        RTNCC
* D1 pointing to 1st record to list
LIST30 D1=D1+ 4                 Point to data
        AD1EX
        AR3EX                   A(A)=end rec#;R3=list start
        C=R2                    SOD
        GOSUB   FRCRDn
        CD1EX
        D1=C                    Copy D1 into C(A)
        GOC     LIST40          C(A) at EOF or EOD
* Position past last record to list
        GOSUB   RCDSKP
LIST40 D0=(5) =S-R1-1           Write out for PRPSND
        DAT0=C A                Save ptr past last recrd to list
        C=D     A               Save ptr to EOF
        RSTK=C
* Pop update address off GOSUB stack which was put there by POLL
        GOSBVL =POPUPD
* R3 contains list start/ S-R1-1 contains list end
        C=R3                    Ptr to data start
LIST50 D1=(5) =OUTBS
```

```
        DAT1=C A                    Start of buffer for PRPSND
        D1=C
        C=RSTK
        D=C    A                    Restore ptr to EOF
        GOSUB  RCDSK+
* List B bytes, starting at D1; C(A) contains ptr to next record
        R0=C
        C=D    A
        RSTK=C                      Save ptr to end of file
        GOSBVL =PRPSND
* C(A)=Ptr to next record
        GOSBVL =CK"ON"              Allow ATTN to interrupt LIST
        D1=C
        D1=D1+ 4                    Step over 2 bytes at record start
        CD1EX
        GONC   LIST50               (B.E.T.)
*

        STITLE EDTEXT Keyword Parse
*****************************************************************
*****************************************************************
**
** Name:        EDTXTp  -  EDTEXT parse
**
** Category:    STPARS
**
** Purpose:
**      Parses EDTEXT statement
**
** Entry:
**      P       = 0
**      D1 past tEDIT
**
** Exit:
**      P       = 0
**
** Calls:       FSPECp, COMCKO, EOLCK, RESPTR, OUTBYT, EXPPAR,
**              R3=D10, D1C=R3, GNXTCR, COMCK, CLRPRM
**
** Uses:        A-C, D(15-5), D1,D0, R0-R3, S0-S3, S7, S10, XM
**              FUNCDO, F-R0-0, F-R0-1
**
** Detail:      EDTEXT <filename> [,<command string>]
**
** Algorithm:
**              This statement is tokenized as a CALL:
**              tEDTEXT tLITRL EDTEXT tPRMST <string> tCVAL ...
**              ... <string> tCVAL tPRMEN
**
** Stk lvls:    6
**
```

```
**
** History:
**
**     Date      Programmer              Modification
**   --------   ----------     ----------------------------------
**   09/12/83   S.W.           Wrote routine
**   10/26/83   S.W.           Added check to disallow U.D.F.'s
**   11/16/83   S.W.           Added code to disallow imbedded
**                              quotes in a command stream that is
**                              not a string expression.
**
**************************************************************
**************************************************************
*
*
*
 fspece GOVLNG =FSPECe
*
 EDTXTp LCHEX  F3545845544445C4
        GOSBVL =OUTC15          tLITRL EDTEXT tPRMST
* Call FSPECp
* Save D1/D0 in safe place - (R3 not reliable)
        CD0EX
        D0=(5) =FUNCR0
        DAT0=C A
        D0=D0+ 5
        AD1EX
        DAT0=A A
        D0=C
        D1=A
*
        GOSBVL =FSPECp
        GOC    fspece           Invalid file specifier?
* Legal file specifier - ensure it's followed by stmt end or comma
        GOSBVL =EOLCK
        GOC    EDTp05           Stmt end found ?
        GOSUB  comck+           Error exit if no comma
* Restore D1/D0
 EDTp05 D0=(5) =FUNCR0
        C=DAT0 A
        D0=D0+ 5
        A=DAT0 A
        D0=C
        D1=A
*
        GOSUB  EDTpSB
* Optional ,<string expr>
        GOSBVL =COMCK
        GONC   EDTp55           Comma not found?
        ST=1   9
        GOSUB  EDTpSB
```

```
        GONC    EDTp57          (B.E.T.)
*
EDTp55  GOSUB   EDTp60           Output another null & tCVAL
EDTp57  LC(2)   =tPRMEN
        GOTO    outbyt
*
EDTp60  GOSUB   resptr
        GOSUB   "out             Output null string
        GOSUB   "out
CVAL    LC(2)   =tCVAL
        GOTO    outbyt
*
EDTpSB  GOSBVL  =R3=D10
        GOSBVL  =CLRPRM          Clear PRMCNT nibble
        GOSBVL  =EXPPAR
        ?ST=1   3                Not a Legal string expr?
        GOYES   EDTp15
* Valid string expression found & output
* Now check for & disallow user-defined functions
* D1 no longer needed - either restored from R3 or LEXPTR
        D1=(5)  =PRMCNT
        A=DAT1  1
        ?A#0    P                User-defined function found?
        GOYES   EDTp15
        GOSUB   resptr
cval    GONC    CVAL             (B.E.T.)
*
EDTp15  GOSBVL  =D1C=R3          Restore D1/D0
        D0=C
*
        GOSUB   "out             Output leading "
        GOSBVL  =GNXTCR
        LCHEX   0D
        ?A=C    B
        GOYES   msgprm
*
EDTp20  LCASC   \"\
        ?A=C    B
        GOYES   msgprm           Don't allow imbedded quotes
        LC(1)   \'\
        ?A=C    B
        GOYES   msgprm
*
        GOSBVL  =OUT1T+
        A=DAT1  B
        LCHEX   0D
        ?A=C    B
        GOYES   EDTp25
        ?ST=1   9                Command string parse?
        GOYES   EDTp20
* File specifier parse
```

```
        LCASC  \ \
        ?A=C   B
        GOYES  EDTp25
        LCASC  \,\
        ?A#C   B
        GOYES  EDTp20
EDTp25  GOSUB  "out
        GONC   cval              (B.E.T.)
*
"out    LCASC  \"\               Output trailing "
outbyt  GOVLNG =OUTBYT
*
resptr  GOVLNG =RESPTR
*
numck   GOVLNG =NUMCK
*
errxl   ST=1   4
syntxe  GOVLNG =SYNTXe            Syntax error
*
msgprm  ST=1   4
        GOVLNG =IVPARe           Invalid Parm
*
comck+  GOSBVL =COMCK+
        RTNC
        GONC   syntxe            (B.E.T.)
        STITLE REPLACE# Keyword Parse
**************************************************************
**************************************************************
**
** Name:        REPLCp  -  REPLACE#, DELETE#, INSERT# Parse
**
** Category:    STPARS
**
** Purpose:
**      Parses REPLACE#, DELETE#, and INSERT# statements
**
** Entry:
**      P       =  0
**      D1 past tREPLC, tINSRT, or tDELET
**
** Exit:
**      P       =  0
**
** Calls:       #CK, NUMCK, COMCKO, OUT1TK, STRGCK
**
** Uses:        A-C, D1,D0, S0-S3,S7,S8, R3
**
** Detail:      REPLACE# <channel#>,<record#>;<string expr>
**              INSERT# has same syntax as REPLACE#
**
**              DELETE# <channel#>,<record#>
```

```
**
** Stk lvls:   5
**
** History:
**
**     Date      Programmer              Modification
**   --------    ----------    ----------------------------------
**   09/12/83    S.W.          Wrote routine.
**
*****************************************************************
*****************************************************************
*
 DELETp ST=1   8
*
 INSRTp
 REPLCp GOSBVL =#CK
        GOC    errx1           No # ?
        D1=D1+ 2               Step over #
        GOSUB  numck           Parse channel no.
        GOSUB  comck+          Output tCOMMA; error if not found
        GOSUB  numck           Parse record#
        ?ST=1  8               DELETE# parse?
        GOYES  resptr
        LC(2)  =tSEMIC
        ?A#C   B
        GOYES  syntxe
        GOSBVL =OUT1TK         Output tSEMIC
        GOVLNG =STRNGP
*
*
        STITLE EDTEXT Keyword Decompile
*****************************************************************
*****************************************************************
**
** Name:        EDTXTd   -   EDTEXT decompile
**
** Category:    STDCMP
**
** Purpose:
**        Decompiles EDTEXT statement
**
** Entry:
**        P       =  0
**        D1 past tEDTEXT
**        D(A) contains end of available memory (AVMEME)
**
** Exit:
**        P       =  0
**        via OUTEL1
**
** Calls:       OUTBYT, EXPRDC
```

```
**
** Uses:        A-C, R0-R2, D1,D0, S0,S3,S8,S10,S11
**
** Stk lvls:    5
**
** History:
**
**    Date      Programmer              Modification
**  --------    ----------    ---------------------------------
**  09/12/83    S.W.          Wrote routine.
**
*********************************************************************
*********************************************************************
*
EDTXTd
        D1=D1+ 16           Step tLITRL, \EDTEXT\, tPRMST
        GOSBVL =EXPRDC
        D1=D1+ 2            Step over tCVAL
        LCASC  \,\
        GOSUB  outbyt
        GOSBVL =EXPRDC
        D1=D1+ 4            Step over tCVAL, tPRMEN
        GOVLNG =OUTEL1
*
        STITLE REPLACE# Keyword Decompile
*********************************************************************
*********************************************************************
**
** Name:        REPLCd  -  REPLACE#, INSERT#, DELETE# decompile
**
** Category:    STDCMP
**
** Purpose:
**      Decompiles REPLACE#, INSERT#, DELETE# statements
**
** Entry:
**      P      =  0
**      D1 past tREPLC, tINSRT, or tDELET
**      D(A) contains end of available memory (AVMEME)
**
** Exit:
**      P      =  0
**      via FIXDC
**
** Calls:       OUTBYT, EXPRDC
**
** Uses:        A-C, D1,D0, R0-R2, S0,S3,S8,S10,S11
**
** Stk lvls:    5
**
** History:
```

```
**
**    Date      Programmer              Modification
**  --------   ----------    ------------------------------------
**  09/12/83   S.W.          Wrote routine
**
***************************************************************
***************************************************************
*
 DELETd
 INSRTd
 REPLCd LCASC  \#\
        GOSUB  outbyt
 SCRLLd GOVLNG =FIXDC
*
***************************************************************
***************************************************************
**
** Name: POSFIL, POSTXT  -  Position Memory Text File to Record n
**
** Category:    FILUTL
**
** Purpose:
**       Position memory text file to given record.  File is
**       indicated by channel number (POSFIL), or file header
**       (POSTXT).
**
** Entry:
**       A(B)    =  Channel number      (POSFIL only)
**       A(A)    =  File header address (POSTXT only)
**       R1(A)   =  Desired line number (first line = line 0)
**       P       =  0
**
** Exit:
**    HARD ERROR EXIT if Channel # not open ("File Not Found")
**    ELSE:
**       sBADRC =  Set if D1 is positioned at a bad record
**       R1     =  Entry condition.
**       P      =  0
**      Carry clear: Desired record found
**       D1     @  Abs address of start of line
**       D0     @  Abs address of start of NEXT line
**       R0     =  Record number of last record in file
**       B(S)   =  File protection nib from FIB
**       D(A)   =  Abs address of EOF
**       D(S)   =  Device code of file (POSFIL only)
**       STMTD1 =  Fib address (POSFIL only)
**      Carry set:  Desired record NOT found
**       sEOF   =  Set if D1 is positioned at EOF as defined
**                    by file chain
**       C(A)   =  Error code:
**                    File is not in memory (POSFIL only)
```

```
**                   File is private
**                   File is not TEXT file
**                   Channel number not found
**                   Premature EOF ("End of File")
**                = 0 if requested line is not in file.  D1 is
**                   positioned at EOD or EOF.  D1, D, and R1
**                   exit conditions are valid.
**
** Calls:       LOCFIL, FILSK+, FRCRDr
**
** Uses.......
**  Inclusive: A, B, C, D, R0, D0, D1, sEOF, sI/OBF, sBADRC
**             STMTD1 (POSFIL only)
**
** Stk lvls:   3
**
** Algorithm:
**      Locate file FIB, return error if channel # not found
**      Verify that file is in memory
**      Fetch file header
**      Verify that file type is TEXT
**      Verify that file is not private
**      Compute file start, EOF
**      Call FRCRDn to locate record
**      Set up exit conditions
**
** History:
**
**    Date     Programmer               Modification
**  --------   ----------   ------------------------------------
**  09/16/83   F. Hall      Designed and coded
**
****************************************************************
****************************************************************

sI/OBF EQU    10
*
**      Locate file FIB, return error if channel # not found
**      Verify that file is in memory
**      Fetch file header
**      Verify that file type is TEXT
**      Verify that file is not private
**      Compute file start, EOF
**      Call FRCRDn to locate record
**      Set up exit conditions
*
POSFIL GOSBVL =FIBADR         Find FIB address (or error out)
       D1=D1+ =oPROTb         Read protection nib
       A=DAT1 S                   . into B(S)
       B=A    S                   .
       GOSBVL =?PRFIL         Private file?
```

```
        RTNC                        . RTNYES
        D1=D1+  (oDEVCb)-(oPROTb) Read device code
        C=DAT1  S                   .
        D=C     S                   .
        C=C+C   S                  Error out if external file
        GOC     POSF40              .
        D1=D1+  (oFBEGb)-(oDEVCb) Read file address
        A=DAT1  A
POSTXT  D1=A                       Check file type
        D1=D1+  oFTYPh              .
        C=0     A                   .
        C=DAT1  4                   .
        C=C-1   A                   .
        ?C#0    A                  Not TEXT file?
        GOYES   POSF60
        D1=D1+  (oFLAGh)-(oFTYPh) Read protection nib
        GOSBVL  =FILSK+            Compute EOF into D
        D=C     A                   .
        D1=D1+  5                  Compute data start into C
        CD1EX                       .
        A=R1                       Recall desired record #
        GOSUB   FRCRDr             Position to desired record
        RTNNC                      Return if record found
        C=0     A
        ?ST=0   sBADRC             Was the problem EOF or EOD?
        RTNYES
        LC(2)   =eEOFIL            "End of file"
        RTNSC
POSF40  LC(4)   =eFACCS            "Invalid access"
        RTNSC
POSF60  LC(4)   =eFTYPE            "Invalid File Type"
        RTNSC

**************************************************************
**************************************************************
**
** Name:        FRCRDn, FRCRDr  -  Find Given TEXT Record
**
** Category:    FILUTL
**
** Purpose:
**      Given TEXT file record #n (n>0), or #r (r>=0), it locates
**      that record.
**
** Entry:
**      A(A)    =  Desired record number.  First record is
**                   1 for FRCRDn, 0 for FRCRDr.
**      C(A)    =  Abs address of file start of data
**      D(A)    =  Abs address of EOF according to file chain
**      P       =  0
**
```

```
** Exit:
**      R0      =  Record number we are positioned at (FFFFF if
**                   no records in file; end of data mark is not
**                   counted as a record)
**      R1      =  Desired record number (>=0)
**      B(A)    =  Number of bytes of data in line according to
**                   line length header (FFFFF if incomplete
**                   header in corrupt record)line
**      sEOF    =  Set iff D1 is positioned at EOF according to
**                   file chain
**      sBADRC =   Set if current record extends beyond EOF.
**                   This indicates file is corrupt, can occur
**                   for two reasons:
**                   a) Only 1 byte left in file (line header
**                        requires 2 bytes)
**                   b) Line header present but record length
**                        extends beyond EOF
**      P       =  0
**    Carry clr:   Desired record found
**      D1      @  Start of desired record
**      D0      @  Start of NEXT record
**    Carry set:   Desired record NOT found
**      D1      @  EOF or EOD mark, or start of last record in
**                   file if sBADRC set
**
** Calls:       PRSREC
**
** Uses:        A, B(A), C, R0, R1, D0, D1, sEOF
**
** Stk lvls:    2
**
** Algorithm:
**      Save current record = -1
**      Save current record address
**      Clear sEOF, sBADRC
**  1.0 Parse record header, return "Not found" if no record
**      Increment current record #
**      If current record # = desired record number, then
**        Return "Found"
**      If sBADRC is clear, then
**        Go to 1.0
**      Else
**        Return "Not found"
**
** History:
**
**    Date      Programmer              Modification
**  --------    ----------   ----------------------------------
**  09/14/83    S.W.         Wrote routine.
**
***************************************************************
```

```
*****************************************************************
*
 FRCRDn A=A-1  A                Convert line # to record #
 FRCRDr R1=A                    Save desired record number
        A=0    W                Save current record number = -1
        A=A-1  A                 .
        R0=A                     .
        ST=0   sEOF             Clear status
        ST=0   sBADRC            .
 FRCR10 GOSUB  PRSREC           Parse record
        RTNC                    Return if no such record
        D0=C                    D0 = start of next line
        A=R0                    Increment current record number
        A=A+1  A                 .
        R0=A                     .
        C=R1                    Are we at desired record number?
        ?A=C   A                 .
        GOYES  rtncc             . return "Found" if so
        ?ST=1  sBADRC           Return "Not found" if bad record
        RTNYES
        CD0EX                   C = start of next line
        GONC   FRCR10           Loop again (BET)

*****************************************************************
*****************************************************************
**
** Name:        PRSREC  - Parse Text Record Header
**
**
** Category:    FILUTL
**
** Purpose:
**      Examine the line length header of a TEXT file record to
**      determine line length for normal record, or presence of
**      end-of-data (EOD) mark, or presenceof end-of-file (EOF),
**      or absence of complete line header (corrupt file).
**
** Entry:
**      C(A)    =  Starting address of record
**      D(A)    =  EOF from file chain
**      P       =  0
**
** Exit:
**      D1      @  Starting address of record
**      D(A)    =  EOF from file chain
**      P       =  0
**    Carry clear: Record exists
**      B(A)    =  Number of bytes of data in record
**      C(A)    =  Starting address of next record
**      sBADRC  =  Set if line goes beyond EOF, else unchanged
**    Carry set:  No record present (at EOF, EOD, or no header)
```

```
**      B(A)    =  0  if at EOF or EOD
**              =  -1 if no line length header present
**      sBADRC =  Set if no header present, else unchanged
**      sEOF    =  Set if at EOF, else unchanged
**
** Calls:       SWPBYT
**
** Uses.......
**  Inclusive: A, B(A), C, D1, sEOF, sBADRC
**
** Stk lvls:    1
**
** Algorithm:
**      Set #Bytes = 0
**      If current position = EOF then
**        Set sEOF
**        Return "Not found"
**      If line header is incomplete, then
**        Set sBADRC
**        Set #Bytes = -1
**        Return "Not found"
**      If line header = EOD mark (FFFF), then
**        Return "Not found"
**      Compute #Bytes in line
**      Compute start of next line
**      If start of next line > EOF, then
**        Set sBADRC
**      Return "Found"
**
** History:
**
**    Date      Programmer              Modification
**  --------    ----------    ---------------------------------
**  09/19/83    FH            Adapted from code by SW
**
*****************************************************************
*****************************************************************
PRSREC B=0     A              Preset #Bytes = 0
       D1=C                   D1 = start of line
       ?C>=D   A              At EOF?
       GOYES   PRSR10
       D1=D1+  4              Check if line header present
       CD1EX                   .
       ?C>D    A              Line header missing?
       GOYES   PRSR20          .
       A=DAT1  4              Read line header
       GOSUB   swpbyt         Compute B = #Bytes of data
       P=      3               .
       B=A     WP              .
       C=B     A              Test for EOF, compute #Bytes
       B=B+1   WP
```

```
        P=      0
        RTNC                        Return "Not found" if EOD
        BCEX    A                   Restore B = #Bytes, C = #Bytes+1
        CSRB                        Round to even #bytes (LIF stndrd)
        C=C+1   A                   Compute total # nibs in record
        C=C+C   A                    . #bytes + 2 for header
        C=C+C   A                    . #nibs   + 4 for header
        AD1EX                       Compute C = start of next line
        D1=A                         .
        C=A+C   A                    .
        ?C<=D   A                   NOT corrupt record?
        GOYES   rtncc
        ST=1    sBADRC              Set "Bad record"
rtncc   RTNCC                       Return "Found"
*
PRSR10  ST=1    sEOF                Set EOF flag
        RTNSC                       Return "Not found"
*
PRSR20  B=B-1   A                   Set #bytes = -1
        ST=1    sBADRC              Set "Bad record"
        RTNSC                       Return "Not found"
*
*****************************************************************
*****************************************************************
**
** Name:        RCDSKP  -  Record Skip
**
** Category:    FILUTL
**
** Purpose:
**      Skips over a TEXT file record.
**
** Entry:
**      D1      @  Record start (prior to 2 byte length field)
**      D(A)    =  EOF from file chain
**      P       =  0
**
** Exit:
**      P       =  0
**     Carry clr =>
**      D1      @  Current record first character (after
**                   2 byte length field)
**      B(A)    =  Number of bytes of data in record
**      C(A)    =  Address of next record
**      sEOF    =  0
**      sBADRC  =  Set iff current record extends beyond EOF
**     Carry set => No record to skip
**      D1      =  Entry condition, which points either at EOF,
**                   end of data (FFFF), or at an incomplete
**                   line header
**      sEOF    =  Set iff D1 points to EOF
```

```
**       sBADRC =  Set iff header incomplete, else 0
**
** Calls:      PRSREC
**
** Uses:       A, B(A), C(A), D1
**
** Stk lvls:   2
**
** History:
**
**    Date      Programmer                Modification
**  --------    ----------     ----------------------------------
**  09/14/83    SW             Wrote routine.
**  09/19/83    FH             Adapted for FILESZR.
**
*************************************************************
*************************************************************
=RCDSK+ D1=D1- 4              Point to start of record
=RCDSKP CD1EX                 C = start of line
        ST=0   sEOF           Preset status
        ST=0   sBADRC          .
        GOSUB  PRSREC         Parse record
        RTNC                  Return if no record to skip
        D1=D1+ 4              Move to first character of record
        RTNCC
        STITLE SEARCH# Function Execute
*10 CALL SC @ END
*20 SUB SC
*30 DIM S9$[96],S$[96],T$[96],T1$[96]
*40 DATA \^.@$
*50 DATA ABC
*60 R=0
*70 READ S9$ @ DISP "Pattern: ";S9$
*80 ON ERROR GOTO 270
*90 READ T$ @ DISP "Target: ";T$
*100 IF S9$[LEN(S9$)]="\" AND S9$[LEN(S9$)-1]#"\\" THEN
*           S9$=S9$[1,LEN(S9$)-1]
*110 S$=S9$ @ C$=S$[1,1] @ I=1 @ A0=0 @ R=0
*120 IF C$="" THEN 270
*125 IF C$<>"\" THEN 190
*127 IF S$[2]#"$" THEN 140
*128 IF A0 AND T$<>"" THEN 270 ELSE I=LEN(T$)+1 @ T1$="" @ GOTO 280
*130 IF C$<>"\" THEN 190
*140 IF S$[2,2]="\" THEN 240
*150 S$=S$[2] @ C$=S$[1,1]
*160 R=NOT R
*170 IF C$="^" AND R AND NOT A0 THEN
*           A0=1 @ S$=S$[2] @ C$=S$[1,1] @ GOTO 130
*190 IF R AND (C$="." OR C$="@" OR C$="$") OR C$="\" THEN 240
*200 IF C$="" THEN T1$="" @ GOTO 280
*210 FOR I=I TO LEN(T$) @ IF C$=T$[I,I] THEN 240
```

```
*220 IF A0 THEN 90
*230 NEXT I @ GOTO 90
*240 T1$=T$[I] @ CALL SCN(S$,T1$,(R),M)
*250 IF M THEN 280
*260 I=I+1 @ IF NOT A0 AND I<=LEN(T$) THEN 190
*270 DISP "not found" @ GOTO 300
*280 DISP "found: ";T1$
*290 DISP "Start:";I;"   Length:";LEN(T1$)
*300 END SUB
*310 SUB SCN(S$,T$,R,M)
*320 DISP S$;"    ";T$;R
*330 DIM S1$[96],T1$[96]
*340 S1=1 @ T1=1 @ S3=LEN(S$) @ T3=LEN(T$)
*350 C$=S$[S1,S1]
*360 IF C$="" THEN T$="" @ GOTO 640
*370 IF C$#"\" THEN 405
*380 S1=S1+1 @ C$=S$[S1,S1]
*390 IF C$="" THEN T1=T1-1 @ GOTO 'EXIT'
*400 IF C$#"\" THEN R=NOT R
*405 IF NOT R THEN 440
*410 IF C$="." THEN 475
*420 IF C$="@" THEN S1=S1+1 @ GOTO 530
*430 IF S$[S1]="$" THEN 510
*440 IF C$#T$[T1,T1] THEN 500
*450 S1=S1+1 @ IF S1>S3 THEN
*                 BEEP 0 @ 'EXIT': T$=T$[1,T1] @ GOTO 640
*460 T1=T1+1 @ IF T1>T3 THEN 480
*470 GOTO 350
*475 IF T1>T3 THEN 500 ELSE 450
*480 IF R AND S$[S1,S1]="@" THEN S1=S1+1 @ GOTO 480
*485 IF R AND S$[S1]="$" THEN 640
*490 IF NOT R AND S$[S1]="\" THEN R=1 @ S1=S1+1 @ GOTO 480
*500 M=0 @ GOTO 650
*510 IF T1>T3 THEN 'EXIT' ELSE 500
*520 T1=T1+1 @ S1=S1+1 @ IF T1>T3 THEN 500
*530 IF S1>S3 THEN 640
*540 IF NOT R THEN 580
*550 IF S$[S1,S1]="@" THEN S1=S1+1 @ GOTO 530
*560 IF S$[S1,S1]="." THEN 520
*570 IF S$[S1]="$" THEN 640
*580 IF S$[S1,S1]="\" THEN 610
*590 IF S$[S1,S1]=T$[T3,T3] THEN 610
*600 T3=T3-1 @ IF T3<T1 THEN 500 ELSE 580
*610 S1$=S$[S1] @ T1$=T$[T3]
*620 CALL SCN(S1$,T1$,(R),M) @ IF NOT M THEN 600
*630 T$=T$[1,T3-1]&T1$
*640 M=1
*650 END SUB
        EJECT
RegExp EQU    0
TopLvl EQU    7
```

```
 Short  EQU    6
 Match  EQU    9
 First  EQU    5
 Anchor EQU    11
 BackSl EQU    92                 Backslash character
*
* 2F89B  FUNCR0      --  Start of pattern
* 2F8A0  FUNCR0+5    --  End line #
* 2F8A5  FUNCR0+10   --  Start line # (Current Line #)
* 2F8AA  FUNCR0+15   --  Start column # - 1
* 2F8AF  FUNCR0+20   --  Temp save of STMTD1, End of File
* 2F8B4  FUNCR1+25   --  Current record pointer
* 2F8B9  FUNCD0-2    --  Start of target
* 2F8C0  FUNCD1      --  PC
 pophex GOSUB  poplr
        D1=D1+ 16
 fltdh  GOVLNG =FLTDH
*
        NIBHEX 8                  5th parm numeric -- channel #
        NIBHEX 8                  4th parm numeric -- end line
        NIBHEX 8                  3rd parm numeric -- start line
        NIBHEX 8                  2nd parm numeric -- start column
        NIBHEX 4                  1st parm string  -- search string
        NIBHEX 55                 Requires 5 parameters
 SEARCH CD0EX
        D0=(5) =FUNCD1
        DAT0=C A                  Save D0 in FUNCD0
        GOSUB  pophex
        R0=A                      R0=Channel #
        GOSUB  pophex
        D0=(2) (=FUNCR0)+5
        DAT0=A A                  (FUNCR0+5)=end line #
        GOSUB  pophex
        D0=D0+ 5
        DAT0=A A                  (FUNCR0+10)=start line#
        R1=A                      Save start line# in R1 for POSFIL
        GOSUB  pophex
        A=A-1  A                  Is column = zero?
        GONC   SEAR05             No, then okay
        A=0    A                  Yes, then treat it like 1
 SEAR05 ST=0   First
        D0=D0+ 5
        DAT0=A A                  (FUNCR0+15)=start column-1
        A=A-1  A                  Is column = one?
        GOC    SEAR10             No, then don't set First flag
        ST=1   First              Yes, then enable anchoring ,
 SEAR10 CD1EX
        R2=C                      R2=stack pointer
        D0=D0+ 5
        D1=(5) =STMTD1
        C=DAT1 A
```

```
        DAT0=C A               (FUNCR0+20)=STMTD1
        A=R0                   Recall channel #
        GOSUB  POSFIL          Find start line #
        GOC    err?
        D0=(5) (=FUNCR0)+20
        C=DAT0 A               Recall value for STMTD1
        D0=(2) =STMTD1
        DAT0=C A               Restore STMTD1
        C=D    A
        D0=(2) (=FUNCR0)+20
        DAT0=C A               (FUNCR0+20)=End of file
        C=R2                   Recall stack pointer
        CD1EX                  D1=stack pointer
        D0=D0+ 5
        DAT0=C A               (FUNCR0+25)=Current record pointer
        GOSUB  SCNPRP          Prepare to SCAN
        GOC    nomtch
        D0=(5) (=FUNCR0)+10    Point to FUNCR0+10
        C=DAT0 A               Read Current record #
        GONC   loop            (B.E.T.)  Start loop
 mferr
badrec  GOTO   bserr
 err?   ?C#0   A
        GOYES  mferr
        C=R2                   Recall stack pointer
        D1=C
        GOSBVL =POPMTH         Discard pattern string from stack
        AD1EX                  A(A)=Stack pointer
        ST=0   Short
        R2=A
 nomtch C=0    W
        GOTO   return          Return result = 0
*
 pop1r  GOVLNG =POP1R
*
*       C(A)=Current record number, D0=FUNCR0+10
 loop   D0=D0- 5               Point to FUNCR0+5
        A=DAT0 A               Read End record #
        ?A<C   A               Past last record to be searched?
        GOYES  nomtch          Yes, then report no match
        D0=D0+ 15              Point to FUNCR0+20
        C=DAT0 A               Read end of file
        D=C    A               D=End of file
        D0=D0+ 5               Point to FUNCR0+25
        C=DAT0 A               Read current record pointer
        GOSUB  PRSREC          Parse record length
        GOC    nomtch          If EOF then report no match
        ?ST=1  sBADRC          Pointing at a bad record
        GOYES  badrec          Yes, then error out
        DAT0=C A               Update current record pointer to nxt ??
        D1=D1+ 4               Point past record length
```

```
       CD1EX                   C=Start of target
       D0=D0+ 5                Point to FUNCR0+30
       DAT0=C A                Remember start of target
       D=C    A                D points to start of target
       C=C+B  A
       C=C+B  A                Point past end of data in record
       R3=C                    R3 points to end of target
       D0=D0- 15               Point to FUNCR0+15
       ?ST=0  First            Is this the first record?
       GOYES  SEAR30           No, then don't skip any columns
       C=DAT0 A                Read start column - 1
       ?B<=C  A                Start column > last column?
       GOYES  nxtrec           Yes, then skip to next record
       D=D+C  A
       D=D+C  A                Point to starting column in target
SEAR30 D0=D0- 15               Point to FUNCR0
       C=DAT0 A                Read pointer to start of pattern
       D1=C                    Free space starts here
       B=C    A                B(A)=Pointer to start of pattern
       GOSUB  SCAN             Scan for pattern in target
       GONC   fndmtc           If found, then return result
nxtrec ST=0   First            No longer First
       D0=(5) (=FUNCR0)+10     Point to FUNCR0+10
       C=DAT0 A                Read current record number
       C=C+1  A                Increment current record number
       DAT0=C A                Update current record number
       GOTO   loop             Loop back to check another record

*************************

fndmtc D0=(5) (=FUNCR0)+10     Point to FUNCR0+10
       C=0    W
       C=DAT0 A                Read current record number
       GOSUB  hxdcw            Convert to decimal
       SETHEX
       R3=C                    Save record number
       D0=(2) (=FUNCR0)+30     Point to FUNCR0+30
       C=0    W
       C=DAT0 A                Read start of target
       CDEX   A                C(A)=Start of match,D(A)=Start of tar??
       A=R1
       D=C-D  A                D(A)=First char of match
       C=A-C  A                C(W)=Length of match
       CSRB                    C(W)=Length of match in bytes
       GOSUB  hxdcw            Convert to decimal
       SETHEX
       CDEX   A                D(A)=Length of match in decimal,
                               C(A)=First char of match in hex nibs
       CSRB                    C(W)=First char of match in bytes
       C=C+1  A                Convert to option base 1
```

```
        GOSUB  hxdcw           Convert to decimal
        C=R3                   Recall record number
        CSL    W
        CSL    W
        CSL    W               Make room for start col
        C=A    X               Copy in start col
        CSL    W
        CSL    W
        CSL    W               Make room for match length
        LCHEX  008             Initial exponent before normalization
        CDEX   X               Copy in match length, D(X)=Exponent
        P=     14
NRM00   CSL    W               Shift one digit
        D=D-1  X               Decrement exponent
        ?C=0   P               Is number normalized?
        GOYES  NRM00           No, then keep shifting
        C=D    X               Yes, then copy exponent back
        SETHEX
return  A=R2
        ?ST=0  Short
        GOYES  retrn1
        A=A+1  A
        A=A+1  A
retrn1  D1=A
        D1=D1- 16
        DAT1=C W
        D0=(5) =FUNCD1
        A=DAT0 A
        D0=A
        GOLONG expr
*
*
*  SCNPRP -- Pops the pattern string off stack (D1 points to string??
*            Exit:  R2 points to end of string
*                   Short set iff R2 has been adjusted because
*                      of a trailing backslash
*                   R0 is (AVMEMS)+21
*                   D1 and (FUNCR0) = Start of pattern
*                   Carry set iff pattern was "" or "\"
*
*
*           Uses:  A(A),C(A),D0,R2,ST(Short)
*
SCNPRP ST=0    Short
       GOSBVL  =REVPOP         Reverses string and pops it
       CD1EX
       D1=C
       D0=(5)  =FUNCR0
       DAT0=C  A               FUNCR0=Start of pattern
       C=C+A   A
       R2=C                    R2=End of pattern
```

```
        ?A=0    A           Is pattern the null string?
        RTNYES              Yes, then no match found
        D0=C
        D0=D0-  2
        A=DAT0  B           Read last char of pattern
        LC(2)   BackSl
        ?A#C    B           Is the last char a backslash?
        GOYES   SCNP10      No, then skip
        D0=D0-  2           Back up to next to last char
        AD1EX
        D1=A                A=Start of pattern
        CD0EX               C=End of pattern - 4
        ?A>=C   A           Is the string at least 2 chars?
        RTNYES              No, then no match found ("\" is illeg??
        D0=C                Point to penultimate char
        A=DAT0  B           Read penultimate char
        LC(2)   BackSl
        ?A=C    B           Is it a backslash?
        GOYES   SCNP10      Yes, then leave pattern alone
        C=R2                No, then delete trailing backslash
        C=C-1   A
        C=C-1   A
        R2=C                Shorten pattern to eliminate
                            backslash
        ST=1    Short       Remember that it was shortened
SCNP10  D0=(5)  =AVMEMS
        A=DAT0  A
        C=0     A
        LC(2)   21
        A=A+C   A           Calculate (AVMEMS)+21 for
                            available memory checks later
        R0=A                Save this in R0
        RTNCC

*
*  SCAN is the search driver, it will try to find the pattern strin??
*       in the specified target string
*  Entry:  B(A) = Start of pattern
*          R2 = End of pattern
*          D(A) = Start of target
*          R3 = End of target
*          D1 = stack pointer (high end of available memory)
*          R0 = (AVMEMS)+21
*          First should be set only if anchor should cause
*              no match (ie first line of search and not first
*              column in target line)
*  Exit:   No match found:  Carry set
*          Match found:  Carry clear and
*            D(A) =  Start of match
*            R1 = End of match
*
```

```
*
 SCAN    ST=0    Anchor         Not anchored to start of line
         ST=0    RegExp         Regular expressions off
         GOSUB   PATCHR         Get first pattern character
         LC(2)   BackSl
         ?A#C    B              Is it a backslash?
         GOYES   L190j          No, then skip
L125.1   D0=D0+  2              Point to second character
         A=DAT0  B              Read second character
*                               There must be a second character
*                               since SCNPRP would not have allowed
*                               just backslash.
         LCASC   \$\
         ?A#C    B              Is second character a $?
         GOYES   L140           No, then continue
         D0=D0+  2
         AD0EX
         C=R2
         ?A#C    B              Is second character the last?
         GOYES   L140           No, then okay
         C=R3                   Yes, then "\$" returns .LLL000
                                where LLL is the target string
                                length plus 1.
         ?ST=0   Anchor         Are we anchored?
         GOYES   L125.2         No, then match eol
         ?C#D    A              Is start = end?
         RTNYES                 No, then \^$ doesn't match
L125.2   D=C     A              Start of match = Past end of string
                                End is same
MATCH+   R1=C                   Point D1 to end of string
         RTNCC                  Return indicating success
L130     GOSUB   PATCH+         Move to next char and read it
         LC(2)   BackSl
L190j    ?A#C    B              Is it a backslash?
         GOYES   L190           No,then skip checking special chars
L140     A=B     A
         D0=A
         D0=D0+  2              Point to next char (temporarily)
         A=DAT0  B              Read next char
         LC(2)   BackSl
         ?A=C    B              Is it a second backslash?
         GOYES   L240j          Yes, then call SCANSB
L150     B=B+1   A              Move to next character
         B=B+1   A
L160     GOSUB   RETOGL         Toggle regular expressions flag
L170     ?ST=0   RegExp         Are regular expressions active?
         GOYES   L190.2         No, then skip looking for special cha??
         ?ST=1   Anchor         Has anchor been specified already?
         GOYES   L190           Yes, then treat ^ like any other char
         GOSUB   PATCHR         Get current pattern char
         LCASC   \^\            No, then check for ^
```

```
        ?A#C    B          Is it an ^?
        GOYES   L190.1     No, then check for other special char??
        ?ST=1   First      Is anchoring allowed?
        RTNYES             No, then return indicating no match
        ST=0    RegExp     Clear regular expression flag, it wil??
                           be turned back on later
        ST=1    Anchor     Now anchored
        GOTO    L125.1     Loop back to start
L190    ?ST=0   RegExp     Are regular expressions active?
        GOYES   L190.2     No, then skip checking for spec. char??
        GOSUB   PATCHR     Get current pattern string
L190.1  LCASC   \.\
        ?A=C    B          Is current char a .?
        GOYES   L240       Yes, then call SCANSB
        LC(1)   \$\
        ?A=C    B          Is it a $?
        GOYES   L240       Yes, then call SCANSB
        LC(2)   \@\
L240j   ?A=C    B          Is it an @?
        GOYES   L240       Yes, then call SCANSB
L190.2  GOSUB   PATCHR     Read current pattern char
        LC(2)   BackSl
        ?A=C    B          Is it a backslash?
        GOYES   L240       Yes, then call SCANSB
L200    C=R2               Recall ptr to end of pattern
        ?B<C    A          At end of pattern?
        GOYES   L210       No, then continue looking
        C=D     A          Yes, then match up to this point
        GOTO    MATCH+
L210    C=R3               Recall ptr to end of target
        ?C<=D   A          At end of target?
        RTNYES             Yes, then return indicating no match
        C=D     A
        D0=C               Point to target character
        C=DAT0  B          Read target character
        ?A=C    B          Does pattern match target char?
        GOYES   L240       Yes, then call SCANSB
        ?ST=1   Anchor     No, then is pattern anchored?
        RTNYES             Yes, then return indicating no match
        D=D+1   A          No, then move to next target characte??
        D=D+1   A
        GOTO    L210       See if this target char matches patte??
L240    ST=1    TopLvl     Calling SCANSB from top level
        GOSUB   SCANSB
        GOC     RTNCC      Return if match found
        ?ST=1   Anchor     Is anchor set?
        RTNYES             Yes, then return indicating no match
        D=D+1   A
        D=D+1   A          No, then move to next target characte??
        C=R3               Recall ptr to end of target string
        ?C<=D   A          At end of target?
```

```
       RTNYES                      Yes, then return indicating no match
       GONC    L190                (B.E.T.) No, then see if next char ma??
 RTNCC RTNCC
*
* SCANSB is a recursive subroutine.
*
* Register usage:
*  E     ST(TopLvl) = Set if called at top level
*  E     R0 = (AVMEMS)+21
*    S   R1 = Pointer past end of matched string.   (T3)
*  E     R2 = Pointer past end of search string.
*  E     R3 = Pointer past end of target string.
*  E S   B  = Current position in search string.    (S1)
*  E S   D  = Current position in target string.    (T1+T2)
*  E     D1 = Stack pointer.
*  E S   RSTK = Return address
*  E S   RegExp = Set iff regular expressions are active
*
* In the table above, lines with an E are entry conditions
* and lines with an S are stacked for each recursion
*
* Exit:
*    Match(S9) and Carry set iff match found
*    R1 = Points past match string (if matched)
*         Not changed if no match found
*    ST(TopLvl) clear
*
* Uses:  A,C,R1,D0,S0,S7,S9,S10,S11,available memory
*
*
*
*
memerr GOVLNG =MEMERR             Report insufficient memory
SCANSB
       CD1EX
       D1=C                       Copy stack pointer to C
       A=R0                       Recall limit of avail mem
       ?A>C    A                  Enough memory?
       GOYES   memerr             No, then error
       D1=D1-  5
       C=R1
       DAT1=C  A                  1 <- R1
       D1=D1-  5
       C=B     A
       DAT1=C  A                  2 <- B(A)
       D1=D1-  5
       C=D     A
       DAT1=C  A                  3 <- D(A)
       ?ST=1   TopLvl
       GOYES   SCNSB1
       C=R1
```

```
        D=C     A
        D=D-1   A
        D=D-1   A               D(A)=Ptr to start of target for sub
SCNSB1  ST=0    TopLvl
        D1=D1-  5
        C=RSTK
        DAT1=C  A               4 <- RSTK
        D1=D1-  1
        C=ST
        DAT1=C  P               5 <- RegExp (S0)
L340    C=R3
        R1=C                    Copy end of target to end of match
L350
L360    C=R2                    Recall end of pattern
        ?C>B    A               At end of pattern string?
        GOYES   L370            No, then continue
        C=D     A               Yes, then target up to this point...
        R1=C                    ...has been matched
        GOTO    L640            Return and indicate success
*
PATCH+  B=B+1   A               Increment pattern pointer
        B=B+1   A
PATCHR  A=B     A               Copy pattern pointer to A
        D0=A                    Then to D0
        A=DAT0  B               Read the pattern character
        RTNCC                   Return
*
*   RETOGL  Toggles regular expressions on/off
RETOGL  ?ST=1   RegExp          Is the RegExp bit set now?
        GOYES   RETOGO          Yes, then clear it
        ST=1    RegExp          No, then set it
        RTN                     Return
RETOGO  ST=0    RegExp          Clear RegExp bit
        RTNCC                   Return


*
L370    GOSUB   PATCHR          Get the current pattern char
        LC(2)   BackSl
        ?A#C    B               Is it a backslash?
        GOYES   L405            No, then continue
L380    GOSUB   PATCH+          Skip backslash and read next char
L390    C=R2                    Recall end of pattern
        ?C>B    A               At end of pattern?
        GOYES   L400            No, then continue
        D=D-1   A               Yes, then have matched
        D=D-1   A               not counting current target char
        GOTO    EXIT            Return indicating match
*
L400    LC(2)   BackSl
        ?A=C    B               Is it a second backslash?
```

```
        GOYES  L405         Yes, then don't toggle RegExp
        GOSUB  RETOGL       No, then toggle RegExp
 L405   ?ST=0  RegExp       Are regular expressions active?
        GOYES  L440         No, then skip looking for
                            special characters
 L410   LCASC  \.\
        ?A#C   B            Is it a .?
        GOYES  L420         No, then continue
 L475   C=R1                Yes, then recall end of target
        ?D<C   A            Is there a character to skip?
        GOYES  L450         Yes, then okay
        GOTO   L500         No, then indicate match not found
*
 L420   LCASC  \@\
        ?A#C   B            Is it an @?
        GOYES  L430         No, then continue
        GOTO   L550.1       Yes, then process it
 L430   LCASC  \$\
        ?A#C   B            Is it a $?
        GOYES  L440         No, then continue
        C=R1                Recall end of target
        ?D<C   A            At end of target string?
        GOYES  L500         No, then report failure
        GOTO   EXIT         Yes, then report success
*
 L440   C=D    A            Copy target string pointer to C
        D0=C                then to D0
        C=DAT0 B            Read current target char
        ?A=C   B            Does this match pattern char?
        GOYES  L450         Yes, then advance to next
                            No, then report failure
 L500   ST=0   Match        Indicate match not found
        GOTO   SCNRTN       Return
*
 L450   B=B+1  A            Advance pattern ptr to next char
        B=B+1  A
        C=R2                Recall end of pattern
        ?B<C   A            Past end of pattern?
        GOYES  L460         No, then continue
 EXIT   C=D    A            Copy current char ptr to C
        C=C+1  A
        C=C+1  A            Move past current char
        R1=C                Set this as end of match
 L640   ST=1   Match        Indicate match found
        GOTO   SCNRTN       Return
*
 L460   D=D+1  A            Advance target pointer
        D=D+1  A
        C=R1                Recall end of target
        ?D>=C  A            Past end of target?
        GOYES  L480         Yes, then check for end of pattern
```

```
        GOTO   L350         No, then continue processing pattern
*
 L480   GOSUB  PATCHR       Recall current pattern character
        B=B+1  A            Increment to next pattern char
        B=B+1  A
        ?ST=0  RegExp       Are regular expressions active?
        GOYES  L490NR       No, then check for \
        LCASC  \@\
        ?A#C   B            Is it an @?
        GOYES  L480.2       No, then look for $
        C=R2                Recall ptr to start of pattern
        ?B>=C  A            At end of pattern?
        GOYES  L640         Yes, then report success
        GONC   L480         (B.E.T.) No, then loop back to
                            check for more @'s or $
 L480.2 LCASC  \$\          Yes, then check for $
        ?A#C   B            Is it a $?
        GOYES  L500         No, then no match found
        C=R2                Yes, then check if its the end of pat??
        ?B>=C  A            At end of pattern?
        GOYES  L640         Yes, then report success
                            target string matchs
        GONC   L500         (B.E.T.) No, then no match found
*
 L490NR LC(2)  BackSl       Check first for backslash
        ?A#C   B            Is it a backslash?
        GOYES  L500         No, then report no match found
        ST=1   RegExp       Yes, then turn on regular expressions
        GONC   L480         (B.E.T.) Now check if @ or $ follows
*
 L520   D=D+1  A            Increment target ptr to next char
        D=D+1  A
        C=R1                Recall end of target
        ?D>=C  A            Past end of target?
        GOYES  L500j        Yes, then no match found
 L550.1 B=B+1  A            Increment pattern ptr to next char
        B=B+1  A
 L530   C=R2                Recall end of pattern
        ?B>=C  A            Past end of pattern?
        GOYES  L640         Yes, then report match
 L540   ?ST=0  RegExp       Are regular expressions active?
        GOYES  L580         No, then skip checking for special ch??
 L550   GOSUB  PATCHR       Recall current pattern character
        LCASC  \@\
        ?A#C   B            Is it an @?
        GOYES  L560         No, then continue
        GONC   L550.1       Yes, then ignore it
                            (Two @'s in a row are same as one).
*
 L560   LCASC  \.\
        ?A=C   B            Is it a .?
```

```
        GOYES  L520          Yes, then skip a target char
 L570   LCASC  \$\
        ?A#C   B             Is it a $?
        GOYES  L580.1        No, then continue
        C=R2                 Yes, then recall end of pattern
        C=C-1  A             Calculate addr of last char in patter??
        C=C-1  A
        ?B<C   A             Is this the last char in pattern?
        GOYES  L580.1        No, then continue
        GOTO   L640.         Yes, then report match found
*
 L580   GOSUB  PATCHR        Recall current pattern character
 L580.1 LC(2)  BackSl
        ?A=C   B             Is it a backslash?
        GOYES  L610          Yes, then do recursion
 L590   C=R1                 Recall end of target
        D0=C                 Point past end of target
        D0=D0- 2             Back up to last char in target
        C=DAT0 B             Read last char in target
        ?A=C   B             Does this match the first pattern cha??
        GOYES  L610          Yes, then do recursion
 L600   C=R1                 Recall end of target
        C=C-1  A
        C=C-1  A             Move it back one character
        R1=C                 Save this as new end of target
        ?D<C   A             Is the target pointer past end?
        GOYES  L580          No, then keep looking for a match
                             with this shorter @ match field
 L500j  GOTO   L500          Yes, then no match found
*
 L610                        Ready for recursion
 L620   GOSUB  SCANSB        Make recursive call
        GONC   L600          Resume search
        GOTO   L640          Report success


 SCNRTN C=ST
        C=DAT1 P
        ST=C                 5 -> RegExp (S0)
        D1=D1+ 1
        C=DAT1 A
        RSTK=C               4 -> RSTK
        D1=D1+ 5
        C=DAT1 A
        D=C    A             3 -> D(A)
        D1=D1+ 5
        C=DAT1 A
        B=C    A             2 -> B(A)
        D1=D1+ 5
        C=DAT1 A
        D1=D1+ 5
```

```
        ?ST=1  Match          Was a match found?
        RTNYES                Yes, then skip restoring R1
                              Note: R1 was left pointing at end
                              of match by subroutine
        R1=C                  1 -> R1
        RTNCC
 hxdcw  GOVLNG =HXDCW
*
        EJECT
*
*       The string returned is in the following format on the stack??
*
*
*                    21 20 19 18  13 12   7 6    1
*  +---------------+--+--+--+------+------+------+
*  |      P6       |  |P5|P4| P3   |  P2  |  P1  |
*  +---------------+--+--+--+------+------+------+
*                  ^  ^  ^
*                  |  |  |
*                  |  |  Command
*                  |  Option char
*                  Error code

 Comma  EQU    11
 NEXT   ST=0   Comma
 NEXT+  B=0    A
 NEXT00 CD1EX
        D1=C
        ?C<=D  A              At eol?
        RTNYES                Yes, then char type=0
        D1=D1- 2              Point to next char
        A=DAT1 B              Read next char
        LC(2)  \ \
        ?A=C   B              Is it a blank?
        GOYES  NEXT00         Yes, then ignore it
        ?ST=1  Comma          Already had a comma
        GOYES  NEXT05         Yes, then don't allow another
        LC(1)  \,\
        ST=1   Comma          Now have a comma or don't care anymor??
        ?A=C   B              Is it a comma?
        GOYES  NEXT00         Yes, then ignore it
 NEXT05 B=B+1  A              Char type=1?
        GOSBVL =DRANGE        Is it a digit?
        GONC   NEXTDG         Yes, then char type=1
        B=B+1  A              Char type=2?
        LC(2)  \.\
        ?A=C   B              Is it a .?
        RTNYES                Yes, then char type=2
        LC(1)  \#\
        ?A=C   B              Is it a #?
        RTNYES                Yes, then char type=2
```

```
        B=B+1  A           Char type=3?
        LC(1)  \+\
        ?A=C   B           Is it a +?
        RTNYES             Yes, then char type=3
        B=B+1  A           Char type=4?
        LC(2)  \?\
        ?A=C   B           Is it a "?"?
        RTNYES             Yes, then char type=4
NEXT10  B=B+1  A           Char type=5
        RTNSC
NEXTDG  GOSUB  ZEROS
ZerPrm  EQU    10
        ST=0   ZerPrm
        C=A    B
        B=C    W
NEXTD1  CD1EX
        D1=C
        ?C<=D  A
        GOYES  NXTD3.
        D1=D1- 2
        A=DAT1 B
        GOSBVL =DRANGE
        GOC    NEXTD3
        BSLC
        BSLC
        ?B=0   P
        GOYES  NEXTD2
        GOSUB  NINES
        B=C    W
        A=B    A
NEXTD2  B=A    B
        GOC    NEXTD1      (B.E.T.)
NINES   LCASC  \999999\
        RTNSC
NEXTD3  D1=D1+ 2           Reinclude this character
NXTD3.  GOSUB  ZEROS
        A=B    W
        ?A#C   W
        GOYES  NEXTD4
        ST=1   ZerPrm
NEXTD4  GOSUB  NINES
        ?A<=C  W
        GOYES  NEXTD5
        A=C    W
NEXTD5  B=0    A
        GOTO   NEXT10
*
*
ZEROS   LCASC  \00000000\
        RTNCC
*
```

```
        NIBHEX 411              One string parameter
EDPARS  CD0EX
        D0=(5) (=FUNCD0)-5
        DAT0=C A                Save PC
        CD1EX
        R3=C                    Save stack pointer in R3
        LC(4)  #F0~ecmds        Message number of cmd letters
        R0=C
        GOSBVL =FPOLL           Poll will change R0 to some other
        CON(2) =pTRANS          message number if translation occurs
        D0=(5) =SCRTCH          Put cmd letters in SCRTCH memory
        C=R0
        GOSBVL =TBMSG$          Initialize SCRTCH to cmd letters
        C=R3
        D1=C                    Restore stack pointer
        GOSBVL =POP1S           Get string from stack
        CD1EX
        D0=C   A                D0=Start of source
        C=A    A                C(A)=Length
        GOSBVL =D1@AVS          A=(AVMEMS)
        D1=(4) =FUNCD0
        DAT1=A A                Initialize FUNCD0 as end of option st??
        D1=D1+ 5
        DAT1=A A                Initialize FUNCD1 as start of opt str??
        D1=A                    D1=Start of dest
        GOSBVL =MOVEU3
        CD0EX
        D0=(5) =AVMEME
        DAT0=C A                Update (AVMEME) to stack pointer
                                (Parameter has been popped off)
        GOSUB  ZEROS
        R1=C
        R2=C
        R3=C                    Initialize parameters P1,P2,P3
        C=0    S
        LCASC  \ \
        R0=C                    Initialize parameters P4,P5,Error
        GOSBVL =D=AVMS
        CLRST                   Clear status bits
STATE1  ST=1   Comma
        GOSUB  NEXT+
        ?ST=1  ZerPrm
        GOYES  ZEROP1
        GOSUB  TYPJMP
        REL(3) EDPERR           Eol
        REL(3) SV1N-2           Digit
        REL(3) SV1.-2           . or #
        REL(3) EDPERR           +
        REL(3) SV5-5            ?
        REL(3) SV4-4            Letter
*
```

```
*
 ZEROP3 P=P+1
 ZEROP2 P=P+1
 ZEROP1 P=P+1
        C=R0
        CPEX   15
        GOTO   EDPER?
*
*
 SV1.-2 GOSUB  ZEROS
        C=A    B
        A=C    W
 SV1N-2 R1=A
        ST=1   0          First parameter found
 STATE2 GOSUB  NEXT
        ?ST=1  ZerPrm
        GOYES  ZEROP2
        GOSUB  TYPJMP
        REL(3) STATE9     Eol
        REL(3) SV2N-3     Digit
        REL(3) SV2.-3     . or #
        REL(3) EDPERR     +
        REL(3) SV5-5      ?
        REL(3) SV4-4      Letter
*
*
 SV3-6  ?ST=1  ZerPrm     Is parameter zero?
        GOYES  ZEROP3     Yes, then error
        R3=A
        ST=1   2          Third parameter found
 STATE6 GOSUB  NEXT
        ?B=0   P
        GOYES  STAT9j
        LCHEX  5
        ?B#C   P
        GOYES  EDPERR
 SV5-8  ?ST=1  4          Option already specified?
        GOYES  EDPERR     Yes, then error
        GOSUB  SV5
 STATE8 GOSUB  NEXT
        ?B#0   P          Is it an Eol?
        GOYES  EDPERR     No, then error
 STAT9j GOTO   STATE9
*
 EDPERR C=R0
        P=     15
        LCHEX  7
 EDPER? R0=C
        P=     0
        GOTO   EDP80
*
```

```
*
 SV5    B=A    A
        BSLC
        BSLC
        A=R0
        B=A    B
        A=B    A
        R0=A
        ST=1   4              Fifth parameter found
        RTNCC
*
*
 SV2.-3 GOSUB  ZEROS
        C=A    B
        A=C    W
 SV2N-3 R2=A
        ST=1   1              Second parameter found
 STATE3 GOSUB  NEXT
        LCHEX  4
        ?B=C   P
        GOYES  SV5-5
        ?B<C   P
        GOYES  EDPERR
*     Fall into SV4-4
*
*
* Translate from CDEFHILMPRST to
*                ABCDEFGHIJKL
 SV4-4  GOSBVL =CONVUC
        D0=(5) =SCRTCH
        B=A    A
        LC(2)  \A\
        P=     11
 SV4-41 A=DAT0 B
        ?A=B   B
        GOYES  SV4-42
        C=C+1  A
        D0=D0+ 2
        P=P-1
        GONC   SV4-41         Loop back for next possible cmd
        C=R0
        C=B    B              Return invalid command letter
        P=     15
        LCHEX  4              Error in parameter 4 (command)
        GOTO   EDPER?
*
 SV5-5  GOSUB  SV5
 STATE5 GOSUB  NEXT
        LCHEX  5
        ?B=C   P
        GOYES  SV4-4
```

```
        GOTO   EDPERR
*
*
*
 SETP6S CD1EX
        D1=C
        D0=(5) =FUNCD1
        DAT0=C A
        ST=1   5              Parameter 6 found
        RTN
*
 SV4-43 GOSUB  SETP6S         Set start of P6 to (D1)
        D0=D0- 5              Point at FUNCD0
        C=DAT0 A              Read end of 6th parameter
        D1=C                  This is new current loc
*       ST=1   5              Sixth parameter found
*  Do we need to indicate that this parm has been found????
        GOTO   STATE9
 SV4-42 A=R0
        A=C    B
        R0=A
        P=     0
        LCASC  \J\            Is it a replace command?
        ?A=C   B
        GOYES  SV4-43
        C=C+1  A
        ?A=C   B              Is it a search command?
        GOYES  SV4-43
 STATE4 GOSUB  NEXT
        GOSUB  TYPJMP
        REL(3) STATE9         Eol
        REL(3) SV3-6          Digit
        REL(3) EDPERR         . or #
        REL(3) EDPERR         +
        REL(3) EDPERR         ?
        REL(3) SV6-7          L
*
*
 SV6-7  D1=D1+ 2              Include this char in parm 6
        A=R0
        LCASC  \B\
        ?A#C   B              Is it a delete command?
        GOYES  SV4-43         No, then P6 is rest of line
        GOSUB  SETP6S         Set start of P6 to (D1)
 SV6-71 D1=D1- 2
        CD1EX
        D1=C
        ?C<D   A
        GOYES  STATE7
        A=DAT1 B
        LC(2)  \ \
```

```
        ?A=C    B
        GOYES   SV6-72
        LC(1)   \+\
        ?A=C    B
        GOYES   SV6-72
        LC(1)   \,\
        ?A#C    B
        GOYES   SV6-71
 SV6-72 D1=D1+  2                 Don't include ",", "+", or " " in P6
        CD1EX
        D1=C
        D0=D0-  5                 Point to FUNCD0
        DAT0=C  A                 P6 ends here
 STATE7 GOSUB   NEXT
        LCHEX   3
        ?B#C    P
        GOYES   STAT71
        GOTO    SV5-8
 STAT71 ?B=0    P
        GOYES   STATE9
        GOTO    EDPERR
 STATE9
        GOSUB   PARMTB
*
*   A bit set in the following table indicates that the
*   corresponding parameter may not be specified
*
*              Parameter #        123456
        NIBHEX 63                 011011 blank (Goto)       @
        NIBHEX 41                 001010 Copy               A
        NIBHEX 40                 001000 Delete             B
        NIBHEX 73                 111011 Exit               C
        NIBHEX 30                 110000 Format             D
        NIBHEX 30                 110000 Help               E
        NIBHEX 63                 011011 Insert             F
        NIBHEX 00                 000000 List               G
        NIBHEX 40                 001010 Move               H
        NIBHEX 00                 000000 Print              I
        NIBHEX 40                 001000 Replace            J
        NIBHEX 40                 001000 Search             K
        NIBHEX 63                 011011 Text               L
 PARMTB C=RSTK
        A=R0
*     Note that LSD of ASCII blank is 0.
        B=0     A
        B=A     P
        B=B+B   A
        C=B+C   A
        D0=C
        A=DAT0  B
        C=ST
```

```
        A=A&C  B
        ?A=0   B
        GOYES  EDP80
        C=0    A
        SB=0
 EDP70  C=C+1  A
        ASRB
        ?SB=0
        GOYES  EDP70
        A=R0
        CSRC
        A=C    S
        R0=A
*
*       Now its time to build stack entry.
*
 EDP80  GOSBVL =D1=AVE          D1&C(A) = (AVMEME)
        CR1EX                   R1=Start of stack item
        D1=D1- 12
        DAT1=C 12               Write out P1
        C=R2
        D1=D1- 12
        DAT1=C 12               Write out P2
        C=R3
        D1=D1- 12
        DAT1=C 12               Write out P3
        C=R0                    Recall P4,P5
        D1=D1- 2
        DAT1=C B                Write out P4
        CSR    A
        CSR    A
        D1=D1- 2
        DAT1=C B                Write out P5
        LCHEX  3
        CSLC
        D1=D1- 2
        DAT1=C B                Write out error code
        D0=(5) =FUNCD0
        C=DAT0 A                Read end of last parameter
                                C(A)=Start of source
        D0=D0+ 5
        A=DAT0 A                Read start of last parameter
                                A(A)=End of source
        GOSBVL =MOVED2          Move final string onto stack
        D0=(5) (=FUNCD0)-5
        C=DAT0 A
        D0=C                    Restore PC
        ST=0   0                Don't return from ADHEAD
        GOVLNG =ADHEAD
*
 TYPJMP C=B    A
```

```
        GOVLNG =TBLJMC
*
        EJECT
        NIBHEX 811              one argument: numeric.
 MSG$   GOSUBL pop1r            Pop; error if cplx or string.
        GOSUBL fltdh            De-normalize, round.
        GONC   MSG$07           NC= neg real; null message.
        ?XM=0                   Real>1E6, NaN or Inf?
        GOYES  MSG$15           No.
*                               Yes. Null message.
 MSG$07 A=0    A                Null message.
 MSG$09 R0=A                    Store msg# in R0.
        SETHEX                  (DEC mode from MSG$15.)
        D1=D1+ 16               D1 past stack item.
        GOSBVL =R3=D10
        GOSBVL =FPOLL           Poll for translation.
        CON(2) =pTRANS
*
        GOSBVL =D0=AVS          Set D0= AvMemSt.
        C=R0                    Fetch message number.
        GOSBVL =TBMSG$          Build msg in avail mem.
        GOVLNG =ERRM$f          Put it on stack, exit.
*                  ------- EXIT
*
 MSG$15 GOSBVL =HEXDEC          Arg back to decimal.
        LCHEX  00256
        ?A>=C  X                Msg number>256?
        GOYES  MSG$07           Yes. Null msg.
        R0=A                    Save msg number.
        ASR    W                LEX ID# to A(X).
        ASR    A
        ASR    A
        ?A>=C  A                LES ID# > 256?
        GOYES  MSG$07           Yes. Null msg.
        GOSBVL =A-MULT          Multiply LEX ID# by 256.
        C=R0                    Fetch msg number.
        C=0    M                C(A)= msg#.
        A=A+C  A
        GOSBVL =DECHEX          e.g., converts 17025 to 1119.
        GOC    MSG$09           (BET)
*
        EJECT
 SCRLEX NIBHEX 25                Id
        CON(2)   2               Lowest Token
        CON(2)   3               Highest Token
        NIBHEX 00000             End of lex table chain
*
        NIBHEX F                 Speed table omitted
        CON(4) (L2TbSt)+1-(*)    Offset to text table
        CON(4) 0                 No message table
        CON(5) 0                 No poll handler
```

```
        STITLE M a i n   T a b l e
*  Main Table
*
        CON(3)    11              02  SCROLL
        REL(5) SCROLL
        NIBHEX D
*
        CON(3)     0              03  MSG$
        REL(5) MSG$
        NIBHEX F
        STITLE T e x t   T a b l e
*  Text Table
 L2TbSt                           Text table start
*
        NIBHEX 7                  MSG$
        NIBASC \MSG$\
        NIBHEX 30
*
        NIBHEX B                  SCROLL
        NIBASC \SCROLL\
        NIBHEX 20
 L2TbEn NIBHEX 1FF                Text termination
*
 SCRLLp GOVLNG =FIXP
*
        REL(5) SCRLLd
        REL(5) SCRLLp
 SCROLL GOSUBL mgosub             Evaluate expression
        CON(5) =EXPEXC
        GOSBVL =D1=AVE
        A=DAT1 A
        GOSUBL pophex             Pop hex number off stack
        A=A-1  A                  Convert to option base 0
        GONC   SCRL10             If non-zero then skip
        A=0    A                  Use zero
 SCRL10 B=0    A
        B=A    B                  Copy to B(A)
        D0=(5) =WINDLN
        A=DAT0 B                  Read window length
        C=0    A
        LC(2)  95
        C=C-A  B                  C(B)=Start of last window
        ?B>C   B                  Is specified start>last start?
        GOYES  SCRL20             No, then okay
        C=B    A                  Yes, then just use last start
 SCRL20 A=C    B
 SCRL30 LC(5)  =DSPBFS
        C=C+A  A
        C=C+A  A                  Calculate address of character
        D0=C                      Point to this character
        C=DAT0 B                  Read character at this spot
```

```
       ?C#0   B               Is it a null?
       GOYES  SCRL40          No, then okay
       A=A-1  A               Yes, then look one char
       GONC   SCRL30          If not at start of buf, then loop bac??
       A=0    A               Otherwise, just use 0
SCRL40 D0=(4) =FIRSTC
       DAT0=A B               Write out calculated FIRSTC
       D0=D0- (FIRSTC)-((DSPSTA)+3)
       C=DAT0 A
       CSTEX
BitsOk EQU    1
       ST=0   BitsOk
       CSTEX
       DAT0=C A
       GOSBVL =SCRLLR
nxtstm GOVLNG =NXTSTM
*
*
* End of LEXFILE
*
FILEND
       END
```

17.4 LEX File Showing Use of Speed Table

Following is a small sample LEX file with a speed table. This example is simply for illustration, since speed tables are appropriate for lex files with a very large number of tokens, which we have omitted here for space considerations.

This LEX file defines the following tokens:

Token Number	Token	LEX File Token Symbol	Description
1	FUNCT	FUNCx	A function
2	BAT	BATx	A statement
3	BATTER	BATRx	A longer statement
4	TOKEN	xTOKEN	An arbitrary token
5	QUIT	QUITx	A non-programmable command

This LEX file includes the necessary external references to the poll handler address, the various execution addresses, and the end-of-file. This example contains a SPEED table which for so few keywords is wasteful and probably wouldn't be used if this were a real LEX file.

```
       TITLE  Lexical Analyzer Tables--ID=FE
*  This file was generated on Wed Dec 15, 1982   2:58 pm
*  File Header
```

```
        NIBASC \TESTFILE\       File Name
        CON(4) =fLEX            File Type
        NIBHEX 00               Flags
        NIBHEX 8541             Time
        NIBHEX 512128           Date
        REL(5) =FILEND          File Length
*
        NIBHEX EF               Id
        CON(2)   1              Lowest Token
        CON(2)   5              Highest Token
        NIBHEX 00000            End of lex table chain
*
*  Speed Table
        NIBHEX 0                 Speed table exists
        CON(3) (TxTbEn)-(TxTbSt)  A
        CON(3)    0               B
        CON(3) (TxTbEn)-(TxTbSt)  C
        CON(3) (TxTbEn)-(TxTbSt)  D
        CON(3) (TxTbEn)-(TxTbSt)  E
        CON(3)   24               F
        CON(3) (TxTbEn)-(TxTbSt)  G
        CON(3) (TxTbEn)-(TxTbSt)  H
        CON(3) (TxTbEn)-(TxTbSt)  I
        CON(3) (TxTbEn)-(TxTbSt)  J
        CON(3) (TxTbEn)-(TxTbSt)  K
        CON(3) (TxTbEn)-(TxTbSt)  L
        CON(3) (TxTbEn)-(TxTbSt)  M
        CON(3) (TxTbEn)-(TxTbSt)  N
        CON(3) (TxTbEn)-(TxTbSt)  O
        CON(3) (TxTbEn)-(TxTbSt)  P
        CON(3)   37               Q
        CON(3) (TxTbEn)-(TxTbSt)  R
        CON(3) (TxTbEn)-(TxTbSt)  S
        CON(3)   48               T
        CON(3) (TxTbEn)-(TxTbSt)  U
        CON(3) (TxTbEn)-(TxTbSt)  V
        CON(3) (TxTbEn)-(TxTbSt)  W
        CON(3) (TxTbEn)-(TxTbSt)  X
        CON(3) (TxTbEn)-(TxTbSt)  Y
        CON(3) (TxTbEn)-(TxTbSt)  Z
        NIBHEX 0                Speed table exists
        CON(4) (TxTbSt)+1-(*)   Offset to text table
        CON(4) 0                No message table
        REL(5) =POLHND          Offset to poll handler
        STITLE M a i n   T a b l e
*  Main Table
=xromFE
*
        CON(3)   24             01  A function
        REL(5) =FUNCx
        NIBHEX F
```

```
*
        CON(3)   15             02  A statement
        REL(5) =BATx
        NIBHEX D
*
        CON(3)    0             03  A longer statement
        REL(5) =BATRx
        NIBHEX D
*
=xTOKEN EQU     #04
        CON(3)   48             04  A token
        NIBHEX 00000
        NIBHEX 0
*
        CON(3)   37             05  A non-programmable command
        REL(5) =QUITx
        NIBHEX 1
        STITLE T e x t   T a b l e
*  Text Table
 TxTbSt                        Text table start
*
        NIBHEX B                A longer statement
        NIBASC \BATTER\
        NIBHEX 30
*
        NIBHEX 5                A statement
        NIBASC \BAT\
        NIBHEX 20
*
        NIBHEX 9                A function
        NIBASC \FUNCT\
        NIBHEX 10
*
        NIBHEX 7                A non-programmable c
        NIBASC \QUIT\
        NIBHEX 50
*
        NIBHEX 9                A token
        NIBASC \TOKEN\
        NIBHEX 40
 TxTbEn NIBHEX 1FF             Text termination
        END
```

17.5 Foreign Language Translation of Messages

See the chapter titled "Message Handling" for a complete descriptio?? of the construction and implementation of message tables. Language translators are LEX files with one purpose: to translate messages from master LEX files. These messages are displayed for errors, warnings, and system messages, for the ERRM$ and MSG$ (MSG$ is foun?? in LEX file #82), and for the g-1ERRM keystroke.

17.5.1 One-shot Mainframe Translator

This Spanish translator for mainframe messages would ALWAYS produce Spanish translations, as long as it is present in memory; hence the term "one-shot". To disable the translation, it must be purged from memory.

```
        TITLE LEXFILE<840101.1823>
*
*  This file was generated on Wed Oct 19, 1983   9:46 am
*  File Header
        NIBASC \ESP001  \  File Name (for lack of better one...)
        CON(4) =fLEX          File Type
        NIBHEX 00             Flags
        NIBHEX 6490           Time
        NIBHEX 910138         Date
        REL(5) FILEND         File Length
*
        NIBHEX 10             Id
        CON(2) 255            Lowest Token
        CON(2)   0            Highest Token
        NIBHEX 00000          End of lex table chain
*
        NIBHEX F              Speed table omitted
        CON(4) (TxTbSt)+1-(*) Offset to text table
        REL(4) MSGTBL       Offset to message table
        REL(5) POLHND         Offset to poll handler
        STITLE M a i n   T a b l e
*  Main Table
=xrom01
        STITLE T e x t   T a b l e
*  Text Table
 TxTbSt                      Text table start
 TxTbEn NIBHEX 1FF           Text termination
        STITLE Mainframe Messages: Espanol
*
```

```
* NOTE ! NOTE ! NOTE ! NOTE ! NOTE ! NOTE ! NOTE ! NOTE ! NOTE
*  +---------------------------------------------------------+
*  | The following Spanish messages are not meant to be      |
*  | the official translations of the mainframe messages.    |
*  | Please excuse the attempt at translation -- this is     |
*  | only meant to be an example of a complete translator    |
*  | LEX file.                                               |
*  +---------------------------------------------------------+
*
 MSGTBL
        CON(2)   1       Min message #
        CON(2) 249       Max message #
*
* -- Note that message 00 need not be included because
*    message 00 from the mainframe is a null message.
*    I.e., MSG$(0) does not have to be translated.
*
*------------------- Math messages ---------------------------
*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
* Message number 8 is placed first because the first   !!
* nibble past the range field MUST be 0 !!  Message    !!
* number 8 has a total length of 16; if this is        !!
* changed, another message with length=16 (or a        !!
* multiple of 16) MUST be placed first.                !!
*                              /Zero                   !!
=sZRDIV EQU     8              /Cero                   !!
        CON(2)  16                                     !!
        CON(2)   8             Message number   8      !!
        CON(1)   4                                     !!
        NIBASC \/Cero\                                 !!
        CON(1)  12                                     !!
*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*                              Underflow
*
=sUNFLW EQU     1              Valor Menudo
        CON(2)  23
        CON(2)   1             Message number   1
        CON(1)  13
        CON(2) =sVALOR
        CON(1)   6
        NIBASC \ Menudo\
        CON(1)  12
*                              Overflow
*
=sOVFLW EQU     2              Valor Rebosado
        CON(2)  11
        CON(2)   2             Message number   2
        CON(1)  13
        CON(2) =sVALOR
        CON(1)  13
        CON(2) =sREBOS
```

```
        CON(1)  12
*                               EXPONENT(0)
*
=sEXPO  EQU      3              EXPONENT(0)
        CON(2)   8
        CON(2)   3              Message number   3
        CON(1)  14
        CON(2) =eEXPO
        CON(1)  12
*                               TAN=Inf
*
=sTNINF EQU      4              TAN=Inf
        CON(2)   8
        CON(2)   4              Message number   4
        CON(1)  14
        CON(2) =eTNINF
        CON(1)  12
*                               0^neg
*
=s0^NEG EQU      5              0^neg
        CON(2)   8
        CON(2)   5              Message number   5
        CON(1)  14
        CON(2) =e0^NEG
        CON(1)  12
*                               0^0
*
=s0^0   EQU      6              0^0
        CON(2)   8
        CON(2)   6              Message number   6
        CON(1)  14
        CON(2) =e0^0
        CON(1)  12
*                               0/0
*
=sZRO/0 EQU      7              0/0
        CON(2)   8
        CON(2)   7              Message number   7
        CON(1)  14
        CON(2) =eZRO/0
        CON(1)  12
*
* message number 8 is found at the top of the table
*
*                               Neg^Non-int
*
=sNEG^X EQU      9              Neg^(Nro ni Entero)
        CON(2)  45
        CON(2)   9              Message number   9
        CON(1)  10
        NIBASC \Neg^(Nro\
```

```
        NIBASC \ ni\
        CON(1)   7
        NIBASC \ Entero)\
        CON(1)  12
*                               SQR(neg)
*
=sSQR-  EQU     10              SQR(neg)
        CON(2)   8
        CON(2)  10              Message number  10
        CON(1)  14
        CON(2) =eSQR-
        CON(1)  12
*                               Invalid Arg
*
=sIVARG EQU     11              Operacion Prohibida
        CON(2)  14
        CON(2)  11              Message number  11
        CON(1)  13
        CON(2) =sOPERA
        CON(1)  13
        CON(2) =sPROHI
        CON(1)   0
        NIBASC \a\
        CON(1)  12
*                               LOG(0)
*
=sLN0   EQU     12              LOG(0)
        CON(2)   8
        CON(2)  12              Message number  12
        CON(1)  14
        CON(2) =eLN0
        CON(1)  12
*                               LOG(neg)
*
=sLOG-  EQU     13              LOG(neg)
        CON(2)   8
        CON(2)  13              Message number  13
        CON(1)  14
        CON(2) =eLOG-
        CON(1)  12
*                               Inf/Inf
*
=sIF/IF EQU     14              Inf/Inf
        CON(2)   8
        CON(2)  14              Message number  14
        CON(1)  14
        CON(2) =eIF/IF
        CON(1)  12
*                               Inf-Inf
*
=sIF-IF EQU     15              Inf-Inf
```

```
        CON(2)  8
        CON(2)  15              Message number  15
        CON(1)  14
        CON(2)  =eIF-IF
        CON(1)  12
*                               Inf*0
*
=sIF*ZR EQU     16              Inf*0
        CON(2)  8
        CON(2)  16              Message number  16
        CON(1)  14
        CON(2)  =eIF*ZR
        CON(1)  12
*                               1^Inf
*
=s1^INF EQU     17              1^Inf
        CON(2)  8
        CON(2)  17              Message number  17
        CON(1)  14
        CON(2)  =e1^INF
        CON(1)  12
*                               Inf^0
*
=sINF^0 EQU     18              Inf^0
        CON(2)  8
        CON(2)  18              Message number  18
        CON(1)  14
        CON(2)  =eINF^0
        CON(1)  12
*                               Signaled Op
*
=sSIGOP EQU     19              Operacion de Senal
        CON(2)  22
        CON(2)  19              Message number  19
        CON(1)  13
        CON(2)  =sOPERA
        CON(1)  13
        CON(2)  =sDE
        CON(1)  4
        NIBASC \Senal\
        CON(1)  12
*                               Unordered
*
=sUNORC EQU     20              Sin Orden
        CON(2)  24
        CON(2)  20              Message number  20
        CON(1)  8
        NIBASC \Sin Orde\
        NIBASC \n\
        CON(1)  12
*                               Inexact
```

```
*
=sINX    EQU      21              Inexacto
         CON(2)   11
         CON(2)   21              Message number  21
         CON(1)   14
         CON(2)  =eINX
         CON(1)    0
         NIBASC \o\
         CON(1)   12
*
*
*---------------- System Errors -----------------------
*                                 Low Battery
*
=sLOBAT  EQU      22              Pilas Descargadas
         CON(2)   41
         CON(2)   22              Message number  22
         CON(1)   10
         NIBASC \Pilas De\
         NIBASC \sca\
         CON(1)    5
         NIBASC \rgadas\
         CON(1)   12
*                                 System Error
*
=sSYSER  EQU      23              Error de Sistema
         CON(2)   34
         CON(2)   23              Message number  23
         CON(1)    4
         NIBASC \Error\
         CON(1)   13
         CON(2)  =sDE
         CON(1)    6
         NIBASC \Sistema\
         CON(1)   12
*                                 Insufficient Memory
*
=sMEM    EQU      24              Memoria Insuficiente
         CON(2)   47
         CON(2)   24              Message number  24
         CON(1)   10
         NIBASC \Memoria \
         NIBASC \Ins\
         CON(1)    8
         NIBASC \uficient\
         NIBASC \e\
         CON(1)   12
*                                 Module Pulled
*
=sMPI    EQU      25              Enchufe Arrancado
         CON(2)   41
```

```
        CON(2)  25          Message number  25
        CON(1)  10
        NIBASC \Enchufe \
        NIBASC \Arr\
        CON(1)   5
        NIBASC \ancado\
        CON(1)  12
*                           Configuration
*
=s2MROM EQU     26          Configuracion
        CON(2)  33
        CON(2)  26          Message number  26
        CON(1)  11
        CON(1)  12
        NIBASC \Configur\
        NIBASC \acion\
        CON(1)  12
*                           Invalid AF
*
=sAF    EQU     27          AF Invalido
        CON(2)  13
        CON(2)  27          Message number  27
        CON(1)   1
        NIBASC \AF\
        CON(1)  13
        CON(2) =sINV-0
        CON(1)  12
*
*
*----------------- Program Errors -----------------------
*                           Subscript
*
=sSUBSC EQU     28          Suscripto
        CON(2)  24
        CON(2)  28          Message number  28
        CON(1)   8
        NIBASC \Suscript\
        NIBASC \o\
        CON(1)  12
*                           Record Ovfl
*
=sRECOR EQU     29          Registro Rebosado
        CON(2)  25
        CON(2)  29          Message number  29
        CON(1)   7
        NIBASC \Registro\
        CON(1)  13
        CON(2) =sREBOS
        CON(1)  12
*                           Stmt Not Found
*
```

```
=sSTMNF EQU     30              Se Falta Planteo
        CON(2)  23
        CON(2)  30              Message number  30
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   6
        NIBASC \Planteo\
        CON(1)  12
*                               Data Type
*
=sDATTY EQU     31              Data Invalida
        CON(2)  17
        CON(2)  31              Message number  31
        CON(1)   3
        NIBASC \Data\
        CON(1)  13
        CON(2) =sINV-A
        CON(1)  12
*                               No Data
*
=sNODAT EQU     32              Se Falta Dato
        CON(2)  17
        CON(2)  32              Message number  32
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   3
        NIBASC \Dato\
        CON(1)  12
*                               FN Not Found
*
=sFNNtF EQU     33              Se Falta FN
        CON(2)  13
        CON(2)  33              Message number  33
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   1
        NIBASC \FN\
        CON(1)  12
*                               XFN Not Found
*
=sXFNNF EQU     34              Se Falta XFN
        CON(2)  15
        CON(2)  34              Message number  34
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   2
        NIBASC \XFN\
        CON(1)  12
*                               XWORD Not Found
*
=sXWORD EQU     35              Se Falta XWORD
```

```
        CON(2)  19
        CON(2)  35              Message number  35
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   4
        NIBASC \XWORD\
        CON(1)  12
*                               Parameter Mismatch
*
=sPRMIS EQU     36              Valores Sin Parejos
        CON(2)  31
        CON(2)  36              Message number  36
        CON(1)  13
        CON(2) =sVALOR
        CON(1)   1
        NIBASC \es\
        CON(1)  13
        CON(2) =sSIN
        CON(1)   6
        NIBASC \Parejos\
        CON(1)  12
*                               String Ovfl
*
=sSTROV EQU     37              Letrero Rebosado
        CON(2)  23
        CON(2)  37              Message number  37
        CON(1)   6
        NIBASC \Letrero\
        CON(1)  13
        CON(2) =sREBOS
        CON(1)  12
*                               Numeric Input
*
=sNUMIN EQU     38              Asiento Numerico
        CON(2)  27
        CON(2)  38              Message number  38
        CON(1)  13
        CON(2) =sASIEN
        CON(1)   8
        NIBASC \ Numeric\
        NIBASC \o\
        CON(1)  12
*                               Too Many Inputs
*
=sTOOMI EQU     39              Asientos Demasiados
        CON(2)  21
        CON(2)  39              Message number  39
        CON(1)  13
        CON(2) =sASIEN
        CON(1)   1
        NIBASC \s \
```

```
        CON(1)  13
        CON(2)  =sDEMAS
        CON(1)   1
        NIBASC \os\
        CON(1)  12
*                                 Too Few Inputs
*
=sTOOFI EQU     40                Asientos Tan Pocos
        CON(2)  31
        CON(2)  40                Message number  40
        CON(1)  13
        CON(2)  =sASIEN
        CON(1)  10
        NIBASC \s Tan Po\
        NIBASC \cos\
        CON(1)  12
*                                 Chnl# Not Found
*
=sCHNL# EQU     41                Asignacion de Canal
        CON(2)  40
        CON(2)  41                Message number  41
        CON(1)   9
        NIBASC \Asignaci\
        NIBASC \on\
        CON(1)  13
        CON(2)  =sDE
        CON(1)   4
        NIBASC \Canal\
        CON(1)  12
*                                 FOR w/o NEXT
*
=sFwoNX EQU     42                FOR Sin NEXT
        CON(2)  24
        CON(2)  42                Message number  42
        CON(1)   2
        NIBASC \FOR\
        CON(1)  13
        CON(2)  =sSIN
        CON(1)   3
        NIBASC \NEXT\
        CON(1)  12
*                                 NEXT w/o FOR
*
=sNXwoF EQU     43                NEXT Sin FOR
        CON(2)  24
        CON(2)  43                Message number  43
        CON(1)   3
        NIBASC \NEXT\
        CON(1)  13
        CON(2)  =sSIN
        CON(1)   2
```

```
        NIBASC \FOR\
        CON(1)  12
*                               RTN w/o GOSUB
*
=sRwoGS EQU     44              RTN Sin GOSUB
        CON(2)  26
        CON(2)  44              Message number  44
        CON(1)   2
        NIBASC \RTN\
        CON(1)  13
        CON(2) =sSIN
        CON(1)   4
        NIBASC \GOSUB\
        CON(1)  12
*                               Invalid IMAGE
*
=sINVIM EQU     45              IMAGE Invalido
        CON(2)  19
        CON(2)  45              Message number  45
        CON(1)   4
        NIBASC \IMAGE\
        CON(1)  13
        CON(2) =sINV-O
        CON(1)  12
*                               Invalid USING
*
=sINVUS EQU     46              USING Invalido
        CON(2)  19
        CON(2)  46              Message number  46
        CON(1)   4
        NIBASC \USING\
        CON(1)  13
        CON(2) =sINV-O
        CON(1)  12
*                               IMAGE Ovfl
*
=sIMGOV EQU     47              IMAGE Rebosado
        CON(2)  19
        CON(2)  47              Message number  47
        CON(1)   4
        NIBASC \IMAGE\
        CON(1)  13
        CON(2) =sREBOS
        CON(1)  12
*                               Invalid TAB
*
=sIVTAB EQU     48              TAB Invalido
        CON(2)  15
        CON(2)  48              Message number  48
        CON(1)   2
        NIBASC \TAB\
```

```
        CON(1)  13
        CON(2) =sINV-O
        CON(1)  12
*                               Sub Not Found
*
=sSPGNF EQU     49              Se Falta Subprograma
        CON(2)  31
        CON(2)  49              Message number  49
        CON(1)  13
        CON(2) =sFALTA
        CON(1)  10
        NIBASC \Subprogr\
        NIBASC \ama\
        CON(1)  12
*                               Var Context
*
=sVCNTX EQU     50              Contexto Invalido
        CON(2)  25
        CON(2)  50              Message number  50
        CON(1)   7
        NIBASC \Contexto\
        CON(1)  13
        CON(2) =sINV-O
        CON(1)  12
*                               Invalid Stat Array
*
=sIVSAR EQU     51              Matriz de Estadisticos
        CON(2)  21
        CON(2)  51              Message number  51
        CON(1)   5
        NIBASC \Matriz\
        CON(1)  13
        CON(2) =sDESTA
        CON(1)  12
*                               Invalid Statistic
*
=sIVSTA EQU     52              Estadistica Invalida
        CON(2)  14
        CON(2)  52              Message number  52
        CON(1)  13
        CON(2) =sESTAD
        CON(1)   0
        NIBASC \a\
        CON(1)  13
        CON(2) =sINV-A
        CON(1)  12
*                               Invalid Stat Op
*
=sIVSOP EQU     53              Operacion de Estadisticos
        CON(2)  11
        CON(2)  53              Message number  53
```

```
        CON(1)  13
        CON(2) =sOPERA
        CON(1)  13
        CON(2) =sDESTA
        CON(1)  12
*                               End of File
*
=sEOFIL EQU     54              Fin de Archivo
        CON(2)  18
        CON(2)  54              Message number  54
        CON(1)   2
        NIBASC \Fin\
        CON(1)  13
        CON(2) =sDE
        CON(1)  13
        CON(2) =sARCHI
        CON(1)  12
*                               Invalid Transform
*
=sILTFM EQU     55              Transform Invalida
        CON(2)  11
        CON(2)  55              Message number  55
        CON(1)  14
        CON(2) =eTFM
        CON(1)  13
        CON(2) =sINV-A
        CON(1)  12
*                               Transform Failed
*
=sTFFLD EQU     56              Se Fallo la Transform
        CON(2)  18
        CON(2)  56              Message number  56
        CON(1)  13
        CON(2) =sFALLO
        CON(1)   2
        NIBASC \la \
        CON(1)  14
        CON(2) =eTFM
        CON(1)  12
*
*
*------------------ File and Device Errors ------------------
*                               File Not Found
*
=sFnFND EQU     57              Archivo Desconocido
        CON(2)  14
        CON(2)  57              Message number  57
        CON(1)  13
        CON(2) =sARCHI
        CON(1)  13
        CON(2) =sDESCO
```

```
        CON(1)   0
        NIBASC \o\
        CON(1)  12
*                               Invalid Filespec
*
=sFSPEC EQU     58              Archivo Especificacion
        CON(2)  40
        CON(2)  58              Message number  58
        CON(1)  13
        CON(2) =sARCHI
        CON(1)  11
        CON(1)  14
        NIBASC \ Especif\
        NIBASC \icacion\
        CON(1)  12
*                               File Exists
*
=sFEXST EQU     59              Archivo Existe
        CON(2)  23
        CON(2)  59              Message number  59
        CON(1)  13
        CON(2) =sARCHI
        CON(1)   6
        NIBASC \ Existe\
        CON(1)  12
*                               Illegal Access
*
=sFACCS EQU     60              Acceso Prohibido
        CON(2)  24
        CON(2)  60              Message number  60
        CON(1)   5
        NIBASC \Acceso\
        CON(1)  13
        CON(2) =sPROHI
        CON(1)   0
        NIBASC \o\
        CON(1)  12
*                               File Protect
*
=sFPROT EQU     61              Archivo Protegido
        CON(2)  29
        CON(2)  61              Message number  61
        CON(1)  13
        CON(2) =sARCHI
        CON(1)   9
        NIBASC \ Protegi\
        NIBASC \do\
        CON(1)  12
*                               File Open
*
=sFOPEN EQU     62              Archivo Abierto
```

```
        CON(2)  25
        CON(2)  62              Message number  62
        CON(1)  13
        CON(2) =sARCHI
        CON(1)   7
        NIBASC \ Abierto\
        CON(1)  12
*                               Invalid File Type
*
=sFTYPE EQU     63              Tipo Invalido de Archivo
        CON(2)  23
        CON(2)  63              Message number  63
        CON(1)   3
        NIBASC \Tipo\
        CON(1)  13
        CON(2) =sINV-O
        CON(1)  13
        CON(2) =sDE
        CON(1)  13
        CON(2) =sARCHI
        CON(1)  12
*                               Device Not Found
*
=sDVCNF EQU     64              Accesorio Desconocido
        CON(2)  30
        CON(2)  64              Message number  64
        CON(1)   8
        NIBASC \Accesori\
        NIBASC \o\
        CON(1)  13
        CON(2) =sDESCO
        CON(1)   0
        NIBASC \o\
        CON(1)  12
*                               Line Too Long
*
=sL2LNG EQU     65              Enunciado Rebosado
        CON(2)  27
        CON(2)  65              Message number  65
        CON(1)   8
        NIBASC \Enunciad\
        NIBASC \o\
        CON(1)  13
        CON(2) =sREBOS
        CON(1)  12
*
*
*--------------  Card Reader Errors  ---------------------------
*                               Write Protected
*
=sPROTD EQU     66              Prot Contra Escribir
```

```
        CON(2)  47
        CON(2)  66              Message number  66
        CON(1)  10
        NIBASC \Prot Con\
        NIBASC \tra\
        CON(1)   8
        NIBASC \ Escribi\
        NIBASC \r\
        CON(1)  12
*                               Not This File
*
=sNOTIN EQU     67              Archivo Equivocado
        CON(2)  31
        CON(2)  67              Message number  67
        CON(1)  13
        CON(2) =sARCHI
        CON(1)  10
        NIBASC \ Equivoc\
        NIBASC \ado\
        CON(1)  12
*                               Verify Fail
*
=sVFYER EQU     68              Se Fallo la Verificacion
        CON(2)  40
        CON(2)  68              Message number  68
        CON(1)  13
        CON(2) =sFALLO
        CON(1)  11
        CON(1)  14
        NIBASC \la Verif\
        NIBASC \icacion\
        CON(1)  12
*                               Unknown Card
*
=sUNKCD EQU     69              Carta Desconocida
        CON(2)  22
        CON(2)  69              Message number  69
        CON(1)   4
        NIBASC \Carta\
        CON(1)  13
        CON(2) =sDESCO
        CON(1)   0
        NIBASC \a\
        CON(1)  12
*                               R/W Error
*
=sRWERR EQU     70              Se Fallo el Traslado
        CON(2)  31
        CON(2)  70              Message number  70
        CON(1)  13
        CON(2) =sFALLO
```

```
        CON(1)  10
        NIBASC \el Trasl\
        NIBASC \ado\
        CON(1)  12
*                               Too Fast
*
=sTUFAS EQU     71              Demasiado Rapido
        CON(2)  25
        CON(2)  71              Message number  71
        CON(1)  13
        CON(2) =sDEMAS
        CON(1)   7
        NIBASC \o Rapido\
        CON(1)  12
*                               Too Slow
*
=sTUSLO EQU     72              Demasiado Despacio
        CON(2)  29
        CON(2)  72              Message number  72
        CON(1)  13
        CON(2) =sDEMAS
        CON(1)   9
        NIBASC \o Despac\
        NIBASC \io\
        CON(1)  12
*                               Wrong Name
*
=sWRGNM EQU     73              Nombre Desconocido
        CON(2)  24
        CON(2)  73              Message number  73
        CON(1)   5
        NIBASC \Nombre\
        CON(1)  13
        CON(2) =sDESCO
        CON(1)   0
        NIBASC \o\
        CON(1)  12
*                               File Too Big
*
=sF2BIG EQU     74              Archivo Rebosado
        CON(2)  11
        CON(2)  74              Message number  74
        CON(1)  13
        CON(2) =sARCHI
        CON(1)  13
        CON(2) =sREBOS
        CON(1)  12
*
*
*------------------ Syntax Errors ------------------------------??
*                                Syntax
```

```
*
=sSYNTX EQU     75              Sintaxis
        CON(2)  22
        CON(2)  75              Message number  75
        CON(1)   7
        NIBASC \Sintaxis\
        CON(1)  12
*                               ) Expected
*
=sPRNEX EQU     76              Se Falta Parentesis
        CON(2)  29
        CON(2)  76              Message number  76
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   9
        NIBASC \Parentes\
        NIBASC \is\
        CON(1)  12
*                               Quote Expected
*
=sQUOEX EQU     77              Se Falta Comillas
        CON(2)  25
        CON(2)  77              Message number  77
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   7
        NIBASC \Comillas\
        CON(1)  12
*                               Excess Chars
*
=sEXCHR EQU     78              Letras Demasiadas
        CON(2)  28
        CON(2)  78              Message number  78
        CON(1)   6
        NIBASC \Letras \
        CON(1)  13
        CON(2) =sDEMAS
        CON(1)   1
        NIBASC \as\
        CON(1)  12
*                               Illegal Context
*
=sILCNT EQU     79              Contexto Prohibido
        CON(2)  28
        CON(2)  79              Message number  79
        CON(1)   7
        NIBASC \Contexto\
        CON(1)  13
        CON(2) =sPROHI
        CON(1)   0
        NIBASC \o\
```

```
        CON(1)  12
*                                  Invalid Expr
*
=sILEXP EQU     80                 Expresion Invalida
        CON(2)  27
        CON(2)  80                 Message number  80
        CON(1)   8
        NIBASC \Expresio\
        NIBASC \n\
        CON(1)  13
        CON(2) =sINV-A
        CON(1)  12
*                                  Invalid Parm
*
=sILPAR EQU     81                 Valor Invalido
        CON(2)  11
        CON(2)  81                 Message number  81
        CON(1)  13
        CON(2) =sVALOR
        CON(1)  13
        CON(2) =sINV-O
        CON(1)  12
*                                  Missing Parm
*
=sMSPAR EQU     82                 Se Falta Valor
        CON(2)  11
        CON(2)  82                 Message number  82
        CON(1)  13
        CON(2) =sFALTA
        CON(1)  13
        CON(2) =sVALOR
        CON(1)  12
*                                  Invalid Var
*
=sILVAR EQU     83                 Variable Invalida
        CON(2)  25
        CON(2)  83                 Message number  83
        CON(1)   7
        NIBASC \Variable\
        CON(1)  13
        CON(2) =sINV-A
        CON(1)  12
*                                  Precedence
*
=sPRCER EQU     84                 Precedencia
        CON(2)  28
        CON(2)  84                 Message number  84
        CON(1)  10
        NIBASC \Preceden\
        NIBASC \cia\
        CON(1)  12
```

```
*                                  Invalid Key
*
=sILKEY EQU     85                 Tecla Invalida
        CON(2)  19
        CON(2)  85                 Message number  85
        CON(1)   4
        NIBASC \Tecla\
        CON(1)  13
        CON(2) =sINV-A
        CON(1)  12
*                                  Operand Expected
*
=sROWRN EQU     86                 Se Falta Operando
        CON(2)  25
        CON(2)  86                 Message number  86
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   7
        NIBASC \Operando\
        CON(1)  12
*                                  Operator Expected
*
=sR1WRN EQU     87                 Se Falta Operario
        CON(2)  25
        CON(2)  87                 Message number  87
        CON(1)  13
        CON(2) =sFALTA
        CON(1)   7
        NIBASC \Operario\
        CON(1)  12
*                                  TFM WRN L###: <msg>
*
=sTFWRN EQU     88                 TFM WRN L###: <msg>
        CON(2)  31
        CON(2)  88                 Message number  88
        CON(1)   8
        NIBASC \TFM WRN \
        NIBASC \L\
        CON(2)  47
        CON(1)   0
        NIBASC \:\
        CON(2)  31
        CON(1)  12
*
*
*-------------- Card Reader Messages -------------------------
*                                  Pull ### of ###
*
=sPLLC# EQU     89                 Saque ### de ###
        CON(2)  11
        CON(2)  89                 Message number  89
```

```
        CON(1)  13
        CON(2) =sSAQU
        CON(1)  13
        CON(2) =s#de#
        CON(1)  12
*                                 Pull Card
*
=sPLLC  EQU     90                Saque Carta
        CON(2)  21
        CON(2)  90                Message number  90
        CON(1)  13
        CON(2) =sSAQU
        CON(1)   5
        NIBASC \ Carta\
        CON(1)  12
*                                 Wrt: Align then ENDLN
*
=sWALGN EQU     91                Esc: Alinee y ENDLN
        CON(2)  15
        CON(2)  91                Message number  91
        CON(1)   2
        NIBASC \Esc\
        CON(1)  13
        CON(2) =sALGN
        CON(1)  12
*                                 Vfy: Align then ENDLN
*
=sVALGN EQU     92                Ver: Alinee y ENDLN
        CON(2)  15
        CON(2)  92                Message number  92
        CON(1)   2
        NIBASC \Ver\
        CON(1)  13
        CON(2) =sALGN
        CON(1)  12
*                                 Read: Align then ENDLN
*
=sRALGN EQU     93                Leer: Alinee y ENDLN
        CON(2)  17
        CON(2)  93                Message number  93
        CON(1)   3
        NIBASC \Leer\
        CON(1)  13
        CON(2) =sALGN
        CON(1)  12
*                                 Prot: Align then ENDLN
*
=sPALGN EQU     94                Prot: Alinee y ENDLN
        CON(2)  17
        CON(2)  94                Message number  94
        CON(1)   3
```

```
        NIBASC \Prot\
        CON(1)  13
        CON(2) =sALGN
        CON(1)  12
*                                Unpr: Align then ENDLN
*
=sUALGN EQU     95               Dsprot: Alinee y ENDLN
        CON(2)  21
        CON(2)  95               Message number  95
        CON(1)   5
        NIBASC \Dsprot\
        CON(1)  13
        CON(2) =sALGN
        CON(1)  12
*                                Cat: Align then ENDLN
*
=sCALGN EQU     96               Cat: Alinee y ENDLN
        CON(2)  15
        CON(2)  96               Message number  96
        CON(1)   2
        NIBASC \Cat\
        CON(1)  13
        CON(2) =sALGN
        CON(1)  12
*                                Trk ### Done
*
=sTRKDN EQU     97               Pista ### Acabado
        CON(2)  35
        CON(2)  97               Message number  97
        CON(1)   5
        NIBASC \Pista \
        CON(2)  63
        CON(1)   6
        NIBASC \Acabado\
        CON(1)  12
*
***********************************************************
***********************************************************
*
**** Building Block words for messages.
*
*                                (trk ### of ###)
*
=sTRKOF EQU    229               (pista ### de ###)
        CON(2)  26
        CON(2) 229               Message number 229
        CON(1)   6
        NIBASC \ (pista\
        CON(1)  13
        CON(2) =s#de#
        CON(1)   0
```

```
        NIBASC \)\
        CON(1)  12
*
*
=sVALOR EQU     230             Valor
        CON(2)  16
        CON(2) 230              Message number 230
        CON(1)   4
        NIBASC \Valor\
        CON(1)  12
*
*
=sREBOS EQU     231             Rebosado
        CON(2)  24
        CON(2) 231              Message number 231
        CON(1)   8
        NIBASC \ Rebosad\
        NIBASC \o\
        CON(1)  12
*
*
=sPROHI EQU     232             Prohibid
        CON(2)  24
        CON(2) 232              Message number 232
        CON(1)   8
        NIBASC \ Prohibi\
        NIBASC \d\
        CON(1)  12
*
*
=sOPERA EQU     233             Operacion
        CON(2)  24
        CON(2) 233              Message number 233
        CON(1)   8
        NIBASC \Operacio\
        NIBASC \n\
        CON(1)  12
*
*
=sESTAD EQU     234             Estadistic
        CON(2)  26
        CON(2) 234              Message number 234
        CON(1)   9
        NIBASC \Estadist\
        NIBASC \ic\
        CON(1)  12
*
*
=sARCHI EQU     235             Archivo
        CON(2)  20
        CON(2) 235              Message number 235
```

```
        CON(1)   6
        NIBASC \Archivo\
        CON(1)  12
*
*
=sFALTA EQU     236             Se Falta
        CON(2)  24
        CON(2) 236              Message number 236
        CON(1)   8
        NIBASC \Se Falta\
        NIBASC \ \
        CON(1)  12
*
*
=sINVAL EQU     237             Invalid
        CON(2)  22
        CON(2) 237              Message number 237
        CON(1)   7
        NIBASC \ Invalid\
        CON(1)  12
*
*
=sINV-O EQU     238             Invalido
        CON(2)  11
        CON(2) 238              Message number 238
        CON(1)  13
        CON(2) =sINVAL
        CON(1)   0
        NIBASC \o\
        CON(1)  12
*
*
=sINV-A EQU     239             Invalida
        CON(2)  11
        CON(2) 239              Message number 239
        CON(1)  13
        CON(2) =sINVAL
        CON(1)   0
        NIBASC \a\
        CON(1)  12
*
*
=sDE    EQU     240             de
        CON(2)  14
        CON(2) 240              Message number 240
        CON(1)   3
        NIBASC \ de \
        CON(1)  12
*
*
=sDESTA EQU     241             de Estadisticos
```

```
        CON(2)  16
        CON(2) 241              Message number 241
        CON(1)  13
        CON(2) =sDE
        CON(1)  13
        CON(2) =sESTAD
        CON(1)   1
        NIBASC \os\
        CON(1)  12
*
*
=sSIN   EQU    242              Sin
        CON(2)  16
        CON(2) 242              Message number 242
        CON(1)   4
        NIBASC \ Sin \
        CON(1)  12
*
*
=sDEMAS EQU    243              Demasiad
        CON(2)  22
        CON(2) 243              Message number 243
        CON(1)   7
        NIBASC \Demasiad\
        CON(1)  12
*
*
=sASIEN EQU    244              Asiento
        CON(2)  20
        CON(2) 244              Message number 244
        CON(1)   6
        NIBASC \Asiento\
        CON(1)  12
*
*
=sDESCO EQU    245              Desconocid
        CON(2)  28
        CON(2) 245              Message number 245
        CON(1)  10
        NIBASC \ Descono\
        NIBASC \cid\
        CON(1)  12
*
*
=sSAQU  EQU    246              Saque
        CON(2)  16
        CON(2) 246              Message number 246
        CON(1)   4
        NIBASC \Saque\
        CON(1)  12
*
```

```
*
=s#de#  EQU     247             ### de ###
        CON(2)  15
        CON(2)  247             Message number 247
        CON(1)   0
        NIBASC \ \
        CON(2)  47
        CON(1)  13
        CON(2) =sDE
        CON(2)  47
        CON(1)  12
*
*
=sALGN  EQU     248             : Alinee y ENDLN
        CON(2)  39
        CON(2)  248             Message number 248
        CON(1)  11
        CON(1)  15
        NIBASC \: Alinee\
        NIBASC \ y ENDLN\
        CON(1)  12
*
*
=sFALLO EQU     249             Se Fallo
        CON(2)  24
        CON(2)  249             Message number 249
        CON(1)   8
        NIBASC \Se Fallo\
        NIBASC \ \
        CON(1)  12
*
*
        NIBHEX FF              Table terminator
*
* Poll handler goes here.  Handler for VER$ poll is
* provided
*
 POLHND ?B=0    B               VER$ poll?
        GOYES   hVER$0          Yes.
        GONC    hVER$2          No.  To hVER$2 w/carry clear.
 hVER$0 C=R3
        D1=C
        A=R2
        D1=D1-  (VER$en)-(VER$st)-2
        CD1EX
        ?A>C    A
        GOYES   hVER$1
        D1=C
        R3=C
*
**!! LCASC text to be returned for VER$ here
```

```
* Include a leading blank!!
*
 VER$st LCASC  \ ESP001\        For lack of any better name....
 VER$en DAT1=C (VER$en)-(VER$st)-2
 hVER$1 RTNSXM
*
**!! Continue poll handler here: Carry is clear, VER$ poll
*  has been handled.
*
 hVER$2
        ?B=0   P                Eliminate pTEST poll, which
        GOYES  EXIT               is in the following range.
        A=B    A                Poll number to A (for RANGE).
*
        NIBHEX 33               This is a LC(4)...
        CON(2) =pTRANS              pTRANS in C(B)
        CON(2) =pWARN               pWARN  in C(3-2)
*
        GOSBVL =RANGE           Poll number in range?
        GOC    EXIT             No.
 MSGhnd A=R0                    Fetch msg number in A(3-0).
        A=0    B
        ASL    A
        ?A#0   A                If m/f message, A(A)=0.
        GOYES  EXIT
        A=R0                    M/f message. Change LEX#
        A=A+1  XS                 to 01.
        R0=A
*            One message in the mainframe (message #88)
*            has a type(5) insertion (indirect msg number).
*            This indirect msg number must also be translated,
*            with a nested pTRANS poll.  But only if the
*            present poll is pMEM, pERROR or pWARN.
*            At this point, if the present poll is pTRANS,
*            exit with XM=0 ("handled").
*
        LC(2)  =eTFWRN          (hex 58) "TFM WRN Lnnn:"
        ?B>C   P                Don't poll for pTRANS poll.
        GOYES  HANDLD           pTRANS poll!  (pTRANS=EF)
        ?A#C   B                Message #88?  (58 hex)
        GOYES  EXIT             No. Exit poll.
        C=R2                    Yes. C(8-5)= insert msg number.
        GOSBVL =CSRC5           Shift msg number to C(A).
        CR0EX                   Put in R0.
        R2=C                    Store R0 in R2 during poll.
        GOSBVL =POLL            Poll to translate insertion
        CON(2) =pTRANS            message. (Slow poll because
*                                 nested.)
        RTNC                    Carry set= error from poll.
        C=R0                    Transltd msg to C(A).
        GOSBVL =CSLC5           Shift transltd msg to C(8-5).
```

```
       CR2EX                    Store back in R2.
       R0=C                     Original R0 back to R0.
 EXIT  C=-C-1 A                 Clear carry.
       RTNSXM
*
 HANDLD XM=0                    "Handled" for pTRANS poll.
       RTNCC
*
* End of LEXFILE
*
 FILEND
       END
```

17.5.2 One-shot HPIL Translator

This Spanish translator for HPIL messages would ALWAYS provide Spanish translation, as long as it was present in memory. (Due to a late-discovered bug in HPIL, for any HPIL message translator to work it must be positioned in the file chain search order before the HPIL ROM. The easiest way to do this is for the user to copy the translator into system RAM; this causes it to come before the HPIL ROM in the file chain search. Subsequent releases of the HPIL ROM will correct this problem.) To disable the translation, the translator file must be purged from RAM.

In order for this particular example to work properly, the mainframe translator shown in the previous example must also be in memory. (This is not true in general; this example was constructed in conjunction with the previous translator.)

```
       TITLE LEXFILE<840101.1823>
*
*  This file was generated on Wed Oct 19, 1983   9:47 am
*  File Header
       NIBASC \ESP255  \   File Name (for lack of better one...)
       CON(4) =fLEX          File Type
       NIBHEX 00             Flags
       NIBHEX 7490           Time
       NIBHEX 910138         Date
       REL(5) FILEND         File Length
*
       NIBHEX FF             Id
       CON(2) 255            Lowest Token
       CON(2)   0            Highest Token
       NIBHEX 00000          End of lex table chain
*
       NIBHEX F              Speed table omitted
       CON(4) (TxTbSt)+1-(*)  Offset to text table
       REL(4) MSGTBL        Offset to message table
```

```
        REL(5) PCLHND           Offset to poll handler
        STITLE M a i n  T a b l e
*  Main Table
*xromFF
        STITLE T e x t  T a b l e
*  Text Table
 TxTbSt                         Text table start
 TxTbEn NIBHEX 1FF              Text termination
        STITLE HPIL Message Table: Espanol
 MSGTBL
* HPIL error messages (Espanol) <840101.1823>
*
*
* NOTE ! NOTE ! NOTE ! NOTE ! NOTE ! NOTE ! NOTE ! NOTE ! NOTE
*   +---------------------------------------------------+
*   | The following Spanish messages are not meant to be    |
*   | the official translations of the HPIL ROM messages.   |
*   | Please excuse the attempt at translation -- this is   |
*   | only meant to be an example of a complete translator  |
*   | LEX file.                                             |
*   |                                                       |
*   | The translation of message 00 is shown as an example. |
*   | Since "HPIL" is a copywrited and widely accepted term,|
*   | it is not recommended that it be changed.  It is      |
*   | done here to demonstrate the implementation of a      |
*   | translated message prefix.  Any error or warning      |
*   | taken from this table will have the "HPCC" prefix     |
*   | displayed.  E.g., "HPCC ERR:Se Falta Medio".          |
*   +---------------------------------------------------+
*
*------------------------------------------------------
* The following equates define the message numbers for
* building blocks from the "01" table -- the Spanish
* translated mainframe messages.
*    E.g., sEXCHR=4E, so 1EXCHR=104E   (hex).
*
*
 1EXCHR EQU    256+(=sEXCHR)          Letras Demasiadas
 1MSPAR EQU    256+(=sMSPAR)          Se Falta Valor
 1ILPAR EQU    256+(=sILPAR)          Valor Prohibido
 1ILEXP EQU    256+(=sILEXP)          Expresion Invalida
 1SYNTX EQU    256+(=sSYNTX)          Sintaxis
 1FPROT EQU    256+(=sFPROT)          Archivo Protegido
 1FnFND EQU    256+(=sFnFND)          Archivo Desconocido
 1FEXST EQU    256+(=sFEXST)          Archio Existe
 1DVCNF EQU    256+(=sDVCNF)          Se Falta Accesorio
 1INV-0 EQU    256+(=sINV-0)          Estado Invalido
 1SYSER EQU    256+(=sSYSER)          Error de Sistema
 1DATTY EQU    256+(=sDATTY)          Data Invalida
 1IVARG EQU    256+(=sIVARG)          Valor Invalido
 1MEM   EQU    256+(=sMEM)            Memoria Insuficiente
```

```
*-----------------------------------------------------------
*
        CON(2) 128       Min message #
        CON(2) 193       Max message #
*
*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
* Message number 00 can be placed first because its  !!
* total length is 16 nibbles.  The first nibble past !!
* the range field MUST be a 0 !!!  If message 00 is  !!
* changed, another message with length=16 (or a      !!
* multiple of 16) MUST be placed first!!              !!
*00                           HPIL                    !!
=sHPIL  EQU    00              HPCC (HP Circuito de Canjear)
        CON(2)  16                                    !!
        CON(2) 00              Message number 00      !!
        CON(1)   4                                    !!
        NIBASC \HPCC \                                !!
        CON(1)  12                                    !!
*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*
*                             Message number 128 is a duplicate
*                             of message 00, so that MSG$(255000)
*                             will provide a translation.
*
=sHPIL* EQU    128
        CON(2)   8
        CON(2) 128             Message number 128
        CON(1)  13
        CON(2) =sHPIL
        CON(1)  12
*01                           ASSIGN IO Needed
*
=sNOASN EQU    129             Se Necesita ASSIGN IO
        CON(2)  27
        CON(2) 129             Message number 129
        CON(1)  13
        CON(2) =sNECES
        CON(1)   8
        NIBASC \ASSIGN I\
        NIBASC \O\
        CON(1)  12
*03                           Excess Chars
*
=sXCESS EQU    131             Letras Demasiadas
        CON(2)  11
        CON(2) 131             Message number 131
        CON(2)  15
        CON(4) =1EXCHR
        CON(1)  12
*04                           Missing Parm
*
```

```
=sMSPAr EQU    132              Se Falta Valor
        CON(2)  11
        CON(2) 132              Message number 132
        CON(2)  15
        CON(4) =1MSPAR
        CON(1)  12
*05                             Illegal Parm
*
=sILPAr EQU    133              Valor Prohibido
        CON(2)  11
        CON(2) 133              Message number 133
        CON(2)  15
        CON(4) =1ILPAR
        CON(1)  12
*06                             Illegal Expr
*
=sILEXp EQU    134              Expresion Invalida
        CON(2)  11
        CON(2) 134              Message number 134
        CON(2)  15
        CON(4) =1ILEXP
        CON(1)  12
*07                             Syntax
*
=sSYNTx EQU    135              Sintaxis
        CON(2)  11
        CON(2) 135              Message number 135
        CON(2)  15
        CON(4) =1SYNTX
        CON(1)  12
*
* Errors 16-31 are tape errors
*
*16                             File Protect
*
=sfPROT EQU    144              Archivo Protegido
        CON(2)  11
        CON(2) 144              Message number 144
        CON(2)  15
        CON(4) =1FPROT
        CON(1)  12
*17                             End of Medium
*
=sEOTAP EQU    145              Fin de Medio
        CON(2)  23
        CON(2) 145              Message number 145
        CON(1)   6
        NIBASC \Fin de \
        CON(1)  13
        CON(2) =sMEDIO
        CON(1)  12
```

```
*18                                      Tape stall-Invalid Medium
*
=sSTALL EQU     146                       Medio Invalido
        CON(2)   14
        CON(2)  146                       Message number 146
        CON(1)   13
        CON(2)  =sMEDIO
        CON(2)   15
        CON(4)  =1INV-0
        CON(1)   12
*19                                      Not LIF-Invalid Medium
*
=sNOLIF EQU     147                       Medio Invalido
        CON(2)    8
        CON(2)  147                       Message number 147
        CON(1)   13
        CON(2)  =sSTALL
        CON(1)   12
*20                                      No Medium
*
=sNOTAP EQU     148                       Se Falta Medio
        CON(2)   27
        CON(2)  148                       Message number 148
        CON(1)    8
        NIBASC \Se Falta\
        NIBASC \ \
        CON(1)   13
        CON(2)  =sMEDIO
        CON(1)   12
*22                                      File Not Found
*
=sNFILE EQU     150                       Archivo Desconocido
        CON(2)   11
        CON(2)  150                       Message number 150
        CON(2)   15
        CON(4)  =1FnFND
        CON(1)   12
*23                                      New medium-Invalid Medium
*
=sNEWTA EQU     151                       Medio Invalido
        CON(2)    8
        CON(2)  151                       Message number 151
        CON(1)   13
        CON(2)  =sSTALL
        CON(1)   12
*24                                      No data -Invalid Medium
*
=sBLANK EQU     152                       Medio Invalido
        CON(2)    8
        CON(2)  152                       Message number 152
        CON(1)   13
```

```
        CON(2) =sSTALL
        CON(1)  12
*25                                 Record #-Invalid Medium
*
=sRECRD EQU     153                  Medio Invalido
        CON(2)   8
        CON(2) 153                   Message number 153
        CON(1)  13
        CON(2) =sSTALL
        CON(1)  12
*26                                 Checksum-Invalid Medium
*
=sCHSUM EQU     154                  Medio Invalido
        CON(2)   8
        CON(2) 154                   Message number 154
        CON(1)  13
        CON(2) =sSTALL
        CON(1)  12
*28                                 Size of File
*
=sTSIZE EQU     156                  Archivo Tamano
        CON(2)  35
        CON(2) 156                   Message number 156
        CON(1)  11
        CON(1)  13
        NIBASC \Archivo \
        NIBASC \Tamano\
        CON(1)  12
*30                                 File Exists
*
=sEFILE EQU     158                  Archio Existe
        CON(2)  11
        CON(2) 158                   Message number 158
        CON(2)  15
        CON(4) =1FEXST
        CON(1)  12
*31                                 Directory Full
*
=sDIRFL EQU     159                  Directorio Esta Lleno
        CON(2)  49
        CON(2) 159                   Message number 159
        CON(1)  10
        NIBASC \Director\
        NIBASC \io \
        CON(1)   9
        NIBASC \Esta Lle\
        NIBASC \no\
        CON(1)  12
*
* Errors 32-47 are HPIL Errors
*
```

```
*32                                   Device Not Found
*
=sTERM  EQU     160                    Se Falta Accesorio
        CON(2)  11
        CON(2) 160                     Message number 160
        CON(2)  15
        CON(4) =1DVCNF
        CON(1)  12
*34                                   Device Not Ready
*
=sNORDY EQU     162                    Accesorio No Esta Listo
        CON(2)  38
        CON(2) 162                     Message number 162
        CON(1)  13
        CON(2) =sACCES
        CON(1)  11
        CON(1)  13
        NIBASC \ No Esta\
        NIBASC \ Listo\
        CON(1)  12
*35                                   Loop Broken
*
=sLTIMO EQU     163                    Circuito Interrumpido
        CON(2)  49
        CON(2) 163                     Message number 163
        CON(1)  10
        NIBASC \Circuito\
        NIBASC \ In\
        CON(1)   9
        NIBASC \terrumpi\
        NIBASC \do\
        CON(1)  12
*36                                   Frame Error- Message Error
*
=sFLOST EQU     164                    Error de Marco
        CON(2)  27
        CON(2) 164                     Message number 164
        CON(1)   8
        NIBASC \Error de\
        NIBASC \ \
        CON(1)  13
        CON(2) =sMARCO
        CON(1)  12
*37                                   Frame Overrun- Message Error
*
=sOVRUN EQU     165                    Error de Marco
        CON(2)   8
        CON(2) 165                     Message number 165
        CON(1)  13
        CON(2) =sFLOST
        CON(1)  12
```

```
*38                                  Frame Changed- Message Error
*
=sLPERR EQU     166                   Error de Marco
        CON(2)   8
        CON(2) 166                    Message number 166
        CON(1)  13
        CON(2) =sFLOST
        CON(1)  12
*39                                  Unexpected Message
*
=sUNEXP EQU     167                   Marco Desconocido
        CON(2)  34
        CON(2) 167                    Message number 167
        CON(1)  13
        CON(2) =sMARCO
        CON(1)  11
        CON(1)  11
        NIBASC \ Descono\
        NIBASC \cido\
        CON(1)  12
*40                                  Frame Lost- Message Error
*
=sXXXXX EQU     168                   Error de Marco
        CON(2)   8
        CON(2) 168                    Message number 168
        CON(1)  13
        CON(2) =sFLOST
        CON(1)  12
*41                                  Invalid Mode
*
=sBADMD EQU     169                   Estado Invalido
        CON(2)  24
        CON(2) 169                    Message number 169
        CON(1)   5
        NIBASC \Estado\
        CON(2)  15
        CON(4) =1INV-O
        CON(1)  12
*42                                  Frame Timeout (SCI)- Loop Broken
*
=sFRTOI EQU     170                   Circuito Interrumpido
        CON(2)   8
        CON(2) 170                    Message number 170
        CON(1)  13
        CON(2) =sLTIMO
        CON(1)  12
*43                                  Frame Timeout (Loop)- Loop Broken
*
=sFRTOL EQU     171                   Circuito Interrumpido
        CON(2)   8
        CON(2) 171                    Message number 171
```

```
        CON(1)  13
        CON(2) =sLTIMO
        CON(1)  12
*44                             System Error (Bad cur addr)
*
=sSYSer EQU    172               Error de Sistema
        CON(2)  11
        CON(2) 172               Message number 172
        CON(2)  15
        CON(4) =1SYSER
        CON(1)  12
*45                             Self-test failed
*
=sTESTF EQU    173               Falta Verificarse
        CON(2)  41
        CON(2) 173               Message number 173
        CON(1)  10
        NIBASC \Falta Ve\
        NIBASC \rif\
        CON(1)   5
        NIBASC \icarse\
        CON(1)  12
*47                             Device Type
*
=sDTYPE EQU    175               Tipo de Accesorio
        CON(2)  25
        CON(2) 175               Message number 175
        CON(1)   7
        NIBASC \Tipo de \
        CON(1)  13
        CON(2) =sACCES
        CON(1)  12
*
*52                             Aborted
*
=sABORT EQU    180               Se Ha Abortado
        CON(2)  35
        CON(2) 180               Message number 180
        CON(1)  11
        CON(1)  13
        NIBASC \Se Ha Ab\
        NIBASC \ortado\
        CON(1)  12
*53                             Invalid Device Spec
*
=sDSPEC EQU    181               Especific'n de Accesorio
        CON(2)  40
        CON(2) 181               Message number 181
        CON(1)  11
        CON(1)  14
        NIBASC \Especifi\
```

```
        NIBASC \c'n de \
        CON(1)  13
        CON(2) =sACCES
        CON(1)  12
*54                                 Not numeric- Data Type
*
=sNNUMR EQU     182
        CON(2)  11
        CON(2) 182                   Message number 182
        CON(2)  15
        CON(4) =1DATTY
        CON(1)  12
*56                                 Invalid Arg
*
=sRANGE EQU     184                  Valor Invalido
        CON(2)  11
        CON(2) 184                   Message number 184
        CON(2)  15
        CON(4) =1IVARG
        CON(1)  12
*57                                 No Loop
*
=sNMBOX EQU     185                  Se Falta Circuito
        CON(2)  41
        CON(2) 185                   Message number 185
        CON(1)  10
        NIBASC \Se Falta\
        NIBASC \ Ci\
        CON(1)   5
        NIBASC \rcuito\
        CON(1)  12
*59                                 Insufficient memory
*
=sNORAM EQU     187                  Memoria Insuficiente
        CON(2)  11
        CON(2) 187                   Message number 187
        CON(2)  15
        CON(4) =1MEM
        CON(1)  12
*60                                 RESTORE IO Needed
*
=sOFFED EQU     188                  Se Necesita RESTORE IO
        CON(2)  29
        CON(2) 188                   Message number 188
        CON(1)  13
        CON(2) =sNECES
        CON(1)   9
        NIBASC \RESTORE \
        NIBASC \IO\
        CON(1)  12
*
```

```
*
* Error messages 64-end are building blocks
*
*
=sMARCO EQU     190             Marco
        CON(2)  16
        CON(2)  190             Message number 190
        CON(1)   4
        NIBASC \Marco\
        CON(1)  12
*
=sACCES EQU     191             Accesorio
        CON(2)  24
        CON(2)  191             Message number 191
        CON(1)   8
        NIBASC \Accesori\
        NIBASC \o\
        CON(1)  12
*
=sMEDIO EQU     192             Medio
        CON(2)  16
        CON(2)  192             Message number 192
        CON(1)   4
        NIBASC \Medio\
        CON(1)  12
*
=sNECES EQU     193             Se Necesita
        CON(2)  31
        CON(2)  193             Message number 193
        CON(1)  11
        CON(1)  11
        NIBASC \Se Neces\
        NIBASC \ita \
        CON(1)  12
*
        NIBHEX FF              Table terminator
*
* Poll handler goes here.  Handler for VER$ poll is
* provided
*
 POLHND ?B=0   B               VER$ poll?
        GOYES  hVER$0          Yes.
        GONC   hVER$2          No.  To hVER$2 w/carry clear.
 hVER$0 C=R3
        D1=C
        A=R2
        D1=D1- (VER$en)-(VER$st)-2
        CD1EX
        ?A>C   A
        GOYES  hVER$1
        D1=C
```

```
        R3=C
*
**!! LCASC text to be returned for VER$ here
* Include a leading blank!!
*
 VER$st LCASC  \ ESP255\      For lack of a better name....
 VER$en DAT1=C (VER$en)-(VER$st)-2
 hVER$1 RTNSXM
*
**!! Continue poll handler here: Carry is clear, VER$ poll
*  has been handled.
*
 hVER$2
        ?B=0   P              Eliminate pTEST poll, which
        GOYES  EXIT             is in the following range.
        A=B    A              Poll number to A (for RANGE).
*
        NIBHEX 33             This is a LC(4)...
        CON(2) =pTRANS            pTRANS in C(B)
        CON(2) =pWARN             pWARN  in C(3-2)
*
        GOSBVL =RANGE         Poll number in range?
        GOC    EXIT           No.
 MSGhnd C=R0                  Msg number to C(3-0).
        A=C    A              Copy msg num to A.
        P=     2
*                        Now load ID # of LEX file.
        LCHEX  FF             HPIL LEX#.
        P=     0
        ?A#C   A              Right LEX file?
        GOYES  EXIT           No. Don't translate.
        CSTEX                 Yes. Set bit7 (adds
        ST=1   7                128 to message number,
        CSTEX                   unless this bit already
        R0=C                    set -- just in case.)
        B=B+1  P              pTRANS poll?
        GOC    HANDLD         Yes. Exit "handled".
*
*         !  At this point, any message which has a type (5)
*         !  insertion must be checked.  These messages are
*         !  known at the time the msg table is constructed.
*         !  If we are handling such a message, a separate
*         !  (nested) pTRANS poll might have to be issued to
*         !  translate the inserted message; but only issue
*         !  the nested poll if you are currently handling a
*         !  a pMEM, pERROR or pWARN poll (pTRANS has already
*         !  exited). To issue the nested poll, fetch the
*         !  indirect msg number from R2, put it in C(A), then
*         !      CR0EX            Put it in R0(A).
*         !      R2=C             Store R0 in R2 during poll.
*         !      GOSBVL =POLL     Issue pTRANS poll.
```

```
*        !        CON(2) =pTRANS
*        !        RTNC              Carry set if error in poll.
*        !        C=R2
*        !        CR0EX             Original R0 back to R0.
*        !    And restore insert message number back in R2.
* NOTE: HPIL ROM does not have any type (5) insertions.
*
 EXIT   C=-C-1 A               Clear carry.
        RTNSXM
*
 HANDLD XM=0                   pTRANS poll "Handled".
        RTNCC
*
*
* End of LEXFILE
*
 FILEND
        END
```

17.5.3 Selectable Translator

The chapter entitled "Message Handling" describes the scheme behing a selectable translator. This example is built from the two in the previous subsections; it could be easily extended to include many more languages.

The structure of the controlling LEX file is described below, followed by example satellite LEX files.

CONTROLLING LEX FILE:

1) Provides keyword to select a language (Keyword and syntax has not been decided upon).

2) When language is selected, it searches the LEX system buffer for the entries for SATELLITE LEX FILE #1, SATELLITE LEX FILE #2, and so on. In each entry it replaces the address with one which will point to the appropriate language table in that satellite file.

3) Also when a language is selected, it opens a system buffer (number defined by bTRANS symbol) to store the current language. If the buffer exists, it simply modifies it.

4) Provides Poll handler for pCONFG which repeats step 2, using the language stored in the system buffer bTRANS as a reference.

5) Provides Poll handler for VER$ for the entire entourage of LEX files, supplying a string such as "TRANS:ESP" (indicating the language in effect; e.g., ESP = ESPANOL).

SATELLITE LEX FILE #1:

```
        TITLE LEXFILE<840101.1823>
*
*  This file was generated on Wed Oct 19, 1983   9:46 am
*  File Header
        NIBASC \TRANS01 \      File Name (for lack of better one...??
        CON(4) =fLEX           File Type
        NIBHEX 00              Flags
        NIBHEX 6490            Time
        NIBHEX 910138          Date
        REL(5) FILEND          File Length
*
        NIBHEX 10              Id
        CON(2) 255             Lowest Token
        CON(2)   0             Highest Token
        NIBHEX 00000           End of lex table chain
*
        NIBHEX F               Speed table omitted
        CON(4) (TxTbSt)+1-(*)  Offset to text table
        CON(4) 0000            No message table.
        REL(5) rtnsxm          No poll handler.
        STITLE M a i n   T a b l e
*  Main Table
=xrom01
        STITLE T e x t   T a b l e
*  Text Table
 TxTbSt                       Text table start
 TxTbEn NIBHEX 1FF            Text termination
*
*                             Poll handler for all translators
 POLHND                       in this satellite file.
        ?B=0   P              Eliminate pTEST poll, which
        GOYES  EXIT             is in the following range.
        A=B    A              Poll number to A (for RANGE).
*
        NIBHEX 33             This is a LC(4)...
        CON(2) =pTRANS            pTRANS in C(B)
        CON(2) =pWARN             pWARN  in C(3-2)
*
        GOSBVL =RANGE         Poll number in range?
        GOC    EXIT           No.
 MSGhnd A=R0                  Fetch msg number in A(3-0).
        A=0    B
```

```
        ASL    A
        ?A#0   A              If m/f message, A(A)=0.
        GOYES  EXIT
        A=R0                  M/f message. Change LEX#
        A=A+1  XS               to 01.
        R0=A
*           One message in the mainframe (message #88)
*           has a type(5) insertion (indirect msg number).
*           This indirect msg number must also be translated,
*           with a nested pTRANS poll.  But only if the
*           present poll is pMEM, pERROR or pWARN.
*           At this point, if the present poll is pTRANS,
*           exit with XM=0 ("handled").
*
        LC(2)  =eTFWRN        (hex 58) "TFM WRN Lnnn:"
        ?B>C   P              Don't poll for pTRANS poll.
        GOYES  HANDLD         pTRANS poll!  (pTRANS=EF)
        ?A#C   B              Message #88?  (58 hex)
        GOYES  EXIT           No. Exit poll.
        C=R2                  Yes. C(8-5)= insert msg number.
        GOSBVL =CSRC5         Shift msg number to C(A).
        CR0EX                 Put in R0.
        R2=C                  Store R0 in R2 during poll.
        GOSBVL =POLL          Poll to translate insertion
        CON(2) =pTRANS          message. (Slow poll because
*                               nested.)
        RTNC                  Carry set= error from poll.
        C=R0                  Transltd msg to C(A).
        GOSBVL =CSLC5         Shift transltd msg to C(8-5).
        CR2EX                 Store back in R2.
        R0=C                  Original R0 back to R0.
 EXIT   C=-C-1 A              Clear carry.
 rtnsxm RTNSXM
*
 HANDLD XM=0                  "Handled" for pTRANS poll.
        RTNCC
*
*
        STITLE Spanish table
*---------------------------------------------
* -- Truncated LEX file for Spanish translation --
*    (identical to a LEX file, but no file header)
*
        NIBHEX 10             Id
        CON(2) 255            Lowest Token
        CON(2)   0            Highest Token
        NIBHEX 00000          End of lex table chain
*
        NIBHEX F              Speed table omitted
        CON(4) (TxTbSt)+1-(*) Offset to text table
        REL(4) SPANms         Offset to message table
```

```
        REL(5) POLHND           Offset to poll handler
*  Main Table
*               When Spanish is selected, the entry in
*               LEX system buffer should point to the
*               label SPANtb.
=SPANtb
*  Text Table
 TxTbSt                         Text table start
 TxTbEn NIBHEX 1FF              Text termination
*
*
*  Message Table (Spanish)
*
 SPANms
        CON(2)   1        Min message #
        CON(2) 249        Max message #
*
=sZRDIV EQU      8              /Cero
        CON(2)  16
        CON(2)   8              Message number   8
   ...........................................................
   .  ... entire message table as shown in previous example  .
   ...........................................................
*
        NIBHEX  FF        Message table terminator.
*
*
*
        STITLE German table
*-------------------------------------------------
* -- Truncated LEX file for German translation --
*    (identical to a LEX file, but no file header)
*
        NIBHEX 10               Id
        CON(2) 255              Lowest Token
        CON(2)   0              Highest Token
        NIBHEX 00000            End of lex table chain
*
        NIBHEX F                Speed table omitted
        CON(4) (TxTbSt)+1-(*)  Offset to text table
        REL(4) GERMms          Offset to message table
        REL(5) POLHND           Offset to poll handler
*  Main Table
*               When German is selected, the entry in
*               the LEX system buffer should point to
*               the label GERMtb.
=GERMtb
*  Text Table
 TxTbSt                         Text table start
 TxTbEn NIBHEX 1FF              Text termination
*  Message Table (German)
```

```
*
 GERMms
        CON(2)   1       Min message #
        CON(2) 249       Max message #
*
   ...........................................................
   .  ... entire message table as translated into German     .
   ...........................................................
*
        NIBHEX  FF       Message table terminator.

      .                                              .
       .                                              .
        ....... more language tables as desired .......
         .                                              .
          .                                              .

* End of LEXFILE
*
 FILEND
        END
```

SATELLITE LEX FILE #2:
This would be constructed the same as satellite LEX file #1, except that it would contain translators for the HPIL ROM, for example. The poll handler for pTRANS, pMEM, pERROR and pWARN would be the same as that found in the example for the one-shot HPIL translator.

ADDITIONAL SATELLITE LEX FILES:
An additional LEX files would be constructed for each translation of a master LEX file. E.g, one satellite file for the Text Editor, one for the MATH ROM, etc. Each satellite file would contain several message tables, one for each language.

HP-71 RESOURCE ALLOCATION	CHAPTER 18

There are several logical and physical resources in the operating system, such as ID numbers or fixed RAM locations, which will from time to time need to be allocated to OEMs or HP application projects. This chapter lists the current allocations for those system resources, such as LEX IDs, system buffer numbers, or poll numbers, that may be reserved out of a range of possible values.

HP-71 Operating system resources will be allocated in a conservative manner by arrangement with HP. If you wish to market software which requires that you reserve certain of these allocations for your exclusive use, please contact Systems Engineering Support in the HP Portable Computer Division Product Support Group at (503) 757-2000 for further information.

18.1 Device Types, Classes and Codes

A brief attempt to explain the very different functions of these similar-sounding terms:

A device type is a nibble which resides in a plug-in device's configuration ID. A value of 0-5 identifies a memory-type device. A value of F identifies a memory-mapped I/O device (such as HPIL mailbox). Because of the restrictions on Device Codes, device types of 6-E are not allowed.

A device class is a nibble which resides in a plug-in device's configuration ID. It is meaningful only for memory-mapped I/O devices, and identifies what sort of memory-mapped device it is. While the device type was used to inform the configuration code that the device should be configured in memory-mapped I/O space, the device class actually identifies what it is, so the support code (HPIL ROM, or whatever) can find it. This nibble becomes part of the configuration table entry.

The device code has nothing to do with the system configuration. It is used within the COPY command to identify memory OR non-memory devices to which the mainframe does NOT know how to copy. For example, EEPROM is a memory device to which the mainframe does not know how to copy; ":TAPE" is a non-memory device to which the mainframe does not know how to copy. Here is an example of how they are used:

MEMORY: If a "COPY TO :PORT(1)" is executed and the mainframe code sees something other than a RAM in PORT(1), it will issue a pCOPY poll seeking some handler which can copy to said device. The device is identified with a device code, which is, in this case, the device type + 1 (as determined from looking at the configuration tables). If, for example, PORT(1) contains an EEPROM (device type = 2), this poll will seek a handler which can copy to something with a device code of 3. Legal device codes are 0-6, although COPY will not poll if the destination has a device code of 0 or 1.

NON-MEMORY: If a "COPY ':TAPE'" is executed, a pFILXQ poll will allow HPIL to recognize ':TAPE', and handle the poll by saying that ':TAPE' has a device code of 8. A subsequent pCOPY poll will look for a handler for a device type of 8. The HPIL ROM will respond and handle the copy. All HPIL-recognized devices have a device code of 8; more specific identification is possible through the "internal coding" fields on the pFILXQ and pCOPYx polls.

Device code 7 is the card reader.

This number goes by several names, among them "Device ID" (in pFILXQ documentation) and "Device Type" (in pCOPYx documentation).

Here are the current allocations of device types, classes and codes:

18.1.1 Device Types

0 = RAM
1 = ROM
2 = EEPROM
3 = (unassigned)
4 = (unassigned)
5 = (unassigned)
F = Memory-mapped I/O

18.1.2 Device Class

0 = HPIL mailbox
1-F = (unassigned)

18.1.3 Device Codes

0 = System RAM
1 = Independent RAM
2-6 = Device type + 1

7 = Card reader
8 = HPIL
9-F = (unassigned)

18.2 File Types

The following file types are currently allocated for the HP-71 product:

MAINFRAME FILE TYPES

Type	Description	Security:	Normal	S*	P*	E*
			Hex Numeric Value			
BASIC	Tokenized BASIC program		E214	E215	E216	E217
BIN	HP-71 Microcode		E204	E205	E206	E207
DATA	Fixed Data		E0F0	E0F1	n/a	n/a
LEX	Language Extension		E208	E209	E20A	E20B
KEY	Key Assignment		E20C	E20D	n/a	n/a
SDATA	Stream Data		E0D0	n/a	n/a	n/a
TEXT	ASCII text, in LIF Type 1 format		0001	E0D5	n/a	n/a

APPLICATIONS FILE TYPES

Type	Description	Security:	Normal	S*	P*	E*
			Hex Numeric Value			
FORTH	Forth vocabulary file		E218	E219	E21A	E21B

* S indicates secure, P indicates Private, E indicates executable

18.3 Funny Physical Key Code Allocations

A lexfile may wish to "push" keys by grabbing the key definition poll. In order to force a key definition poll, the lexfile may put a funny physical keycode (PKC) into the keybuffer (possibly during the SREQ poll) which it will recognize as its own and not as a real key. To avoid conflict, lexfiles need to be assigned a unique PKC for this purpose. Refer to the chapter on "HP-71 Resource Allocation" for information on assignment of unique PKC's.

18.4 LEX IDs

There are 256 LEX IDs within the HP-71, numbered 00 to FF (Hex). They are allocated as described in this section. The first two (IDs 00 and 01) are used by the mainframe. One hundred and fifty LEX IDs are reserved for external or custom products.

An important feature of HP-71 LEX IDs is the ability to allocate portions of a LEX ID. Each LEX ID controls a set of keyword tokens and message numbers which are allocated individually or on a range basis. A particular application may use only a portion of the 255 keywords or 255 message numbers within one LEX ID. Another application can be allocated the next partition of entries within the same LEX ID, and so on. This allows full utilization of LEX IDs.

A summary of the current allocation of LEX IDs is provided below. A further breakdown of the token/message range allocations within the LEX IDs is provided following the summary.

LEX ID ALLOCATION SUMMARY

LEX ID RANGE (Hex)		CATEGORY	TOKENS	MESSAGES
00 - 01		MAINFRAME		
02 - 51		APPLICATIONS		
02 - 10		MATHEMATICS		
	02	Math	All	All
	03	Curve Fit	All	All
11 - 1F		ENGINEERING		
20 - 29		BUSINESS		
2A - 2E		INFORMATION MANAGEMENT		
2F - 33		LANGUAGES		
	2F	FORTH/Assembler	All	All
34 - 38		TOOLS		
	34	Debugger	All	All
39 - 4C		GENERAL PURPOSE		
	39	Editor	All	All
4D - 51		MISCELLANEOUS		
52 - 5B		USER'S LIBRARY		
	52	First LEX ID	01 - 03	0
	53	Second LEX ID	01	01
5C - 5E		TEMPORARY/SCRATCH		
5F - AE		EXTERNAL PRODUCTS (3rd Party, ISVs...)		
AF - E0		CUSTOM PRODUCTS		
E1 - F4		CUSTOM PRODUCTS - SPECIAL		
F5 - FF		PIL and I/O		
	F5	Wand	All	All
	FF	HPIL	All	All

Some detailed information:

All BASIC ROM applications sold by HP will respond to the VER$ poll to indicate the appropriate version of the software. This requires all BASIC ROM applications to include a LEX file containing no keywords, but the appropriate code to indicate the proper VER$. The last LEX ID for Custom Products - Special (244) will be used as the LEX ID to VER$ response of BASIC applications. This LEX ID may also be used for keywords by a particular custom application, without conflict.

The Temporary/Scratch LEX IDs allow users to generate their own temporary and personal LEX files without the intervention of HP needed. This guarantees that usage of this ID does not conflict with an HP supported or custom ROM.

The User's Library LEX files are collections of keywords and functions collected from HP-71 users. As additional keywords are received, the User's Library will release updated versions of these LEX files.

A further breakdown of certain LEX ID allocations is given below.

18.4.1 LEX ID 52 Hex - First User's Library ID

KEYWORD/FUNCTION TOKEN ALLOCATIONS

01 KEYWAIT$
Hold machine in low-power state until a key is placed in the key buffer.

02 SCROLL
Display a scrolled line.

03 MSG$
Returns translated error message by polling language translator LEX files.

18.4.2 LEX ID 53 Hex - Second User's Library ID

KEYWORD/FUNCTION TOKEN ALLOCATIONS

01 DEBUG
Accesses Hard-configured Debugger ROM

MESSAGE NUMBER ALLOCATIONS

01 Debugger Not Found

18.5 Poll Process Number Allocations

Following is a list of poll numbers defined for the mainframe.

Symbolic Name	Process # (HEX)	Brief Description
pVER$	00	VER$ poll
pDEVCp	01	Device Parse
pFILDC	02	File Spec Decompile
pFILXQ	03	File Execute - allows dedicated dvc
pFSPCp	04	File Spec Parse
pFSPCx	05	File Spec Execute
pCAT	06	CAT on non-mainframe device
pCAT$	07	CAT$ of non-mainframe file
pCOPYx	08	COPY execute: unknown Device|>8
pCREAT	09	Create file in external device
pDIDST	0A	Device ID store in RAM @ D1
pFPROT	0B	SECURE/UNSECURE/PRIVATE
pLIST	0C	LIST of non-mainframe file
pMERGE	0D	MERGE file dealing w/ funny device
pPRTCL	0E	Print class
pPRTIS	0F	Printer IS
pPURGE	10	PURGE on non-mainframe device
pRNAME	11	RENAME on non-mainframe device
pENTER	12	Enter data from HP-IL
pPIL2	13	Reserved for HPIL
pPIL3	14	Reserved for HPIL
pPIL4	15	Reserved for HPIL
pPIL5	16	Reserved for HPIL
pFINDF	17	Find file
pRDCBF	18	Read current record to file buffer
pRDNBF	19	Write buffer out & read next record
pWRCBF	1A	Write file buffer to current record
pKYDF	1B	Build key defn in KEYRD
pWTKY	1C	Waiting for key in KEYRD

pIMXQT	1D	IMAGE execution starts
pIMCHR	1E	Unrecognized IMAGE char in parse.
pIMXCH	1F	Unrecognized IMAGE symbol in execution.
pIMbck	20	IMAGE: bckwd search processing
pIMcpi	21	IMAGE: cmplx field initialization
pIMcpw	22	IMAGE: work on complex number
pCRT=8	23	Create non-HP-71 type file
pWCRD8	24	Write card, copycode=8
pEOFIL	25	End of file reached in READ #/PRINT #
pPRIN#	26	PRINT # on non-HP-71 type file
pREAD#	27	READ # on non-HP-71 type file
pSREC#	28	RESTORE # on non-HP-71 type file
pCURSR	29	Cursor Up/Down non-BASIC file type
pDATLN	2A	Return file data length on non-HP-71 file
pEDIT	2B	EDIT with non-BASIC file type
pFASCH	2C	Search for filetype by mnemonic
pFTYPE	2D	File type
pLIST2	2E	LIST non-BASIC/non-KEY file
pMRGE2	2F	MERGE non-BASIC/non-KEY file
pRUNft	30	RUN with unknown File Type
pRUNnB	31	RUN non-BASIC file
pPRGPR	32	PURGE of non-RAM file
pCRDAB	33	Abort card read poll
pRCRD	34	Read card poll
pWCRD	35	Write card poll
pCALRS	36	To restore information from CALL stack
pCALSV	37	To save information on CALL stack
pCMPLX	38	Complex math
pREN	39	Renumber a XWORD stmt with line #
pRTNTp	3A	Return Type unknown
pTIMR#	3B	Timer # > 3 in ON TIMER/OFF TIMER stmts
pTRFMx	3C	Supply Transform Handler Address
pFNIN	3D	Entering user-defined function
pFNOUT	3E	Exiting user-defined function
pTRANS	EF	Poll to Translate a Message
pTEST	F0	Test poll for timing POLLs.
pMEM	F1	Insufficient Memory
pERROR	F2	Error message about to go out.
pWARN	F3	Warning msg about to go out.
pPARSE	F4	Parse take-over poll - FAST Poll
pBSCen	F5	Entering BASIC interpreter
pBSCex	F6	Exiting BASIC interpreter
pZERPG	F7	Zero addresses/RAM associated w/ Program
pExcpt	F8	Exception check after statement
pSREQ	F9	Service request (if SREQ<>0)
pMNLP	FA	Main Loop
pCONFG	FB	Configuration
pPWROF	FC	Power off
pDSWKY	FD	Deep Sleep Wakeup -- key or not
pDSWNK	FE	Deep Sleep Wakeup -- no key down

pCLDST	FF	Cold start

18.6 Reserved RAM Allocations

Reserved RAM is a section of fixed address RAM provided by the operating system for use by application software on an allocation basis. No Reserved RAM has been allocated yet.

```
2F986
   Bit 0  Math ROM (Complex image status bit)
   Bit 1
   Bit 2
   Bit 3
2F987
   .
   .
   .
2F9E5
```

18.7 System Buffer ID Allocations

Buffer Name	Description	Range in Hex Start	Stop
bSTMT	Statement buffer	801	
bIEXKY	Immediate execute key	802	
bFIB	File information	803	
bASSGN	ASSIGN# information	804	
bFILE	Temp for file manipulation	805	
bSTAT	Statistics	806	
bCARD	Card reader	807	
bSTART	STARTUP command	808	
bECOMD	External command	809	
	Available	80A	80D
bKBDIS	KEYBOARD IS key defs	80E	
bPILSV	HPIL save area	80F	
bPILAI	ASSIGNIO names	810	
bSTMXQ	HPIL statement execution	811	
bMATH	Math ROM	812	
bSOLVE	SOLVE (Math ROM)	813	
bINTEG	INTEGRAL (Math ROM)	814	
bMATIO	Matrix IO (Math ROM)	815	

	Available	816	
bCFIT	(Curve Fitting ROM)	817	
bCHISQ	Chi Sq (Curve Fitting ROM)	818	
bGRAD	Gradient (Curve Fit ROM)	819	
bWAND	Wand Status/Cksum Info	81A	
bTRANS	Message Translator	BFA	
bCHARS	Alternate Character Set	BFB	
bLEX	LEX file addresses	BFC	
	Unused	BFD	
bROMTB	ROM Configuration Table	BFE	
bSCRTC	Scratch buffers	E00	FFF

18.8 GOSUB Stack Item Type Allocations (RETURN Types)

```
 0   Return to program
 1   Return to keyboard
 2   ON TIMER 1 ... GOSUB
 3   ON TIMER 2 ... GOSUB
 4   ON TIMER 3 ... GOSUB
 5
 6
 7
 8   Return to assembly language code
 9   Special (to be allocated)
 10  Special (to be allocated)
 11  Special (to be allocated)
 12  Special (to be allocated)
 13  Special (to be allocated)
 14  Special (to be allocated)
 15  Boundary Address
       If address = 0
         Environment boundary
       else
         Update address
```

18.9 System Flag Allocations

```
Flag #   Mnemonic   Function
------   --------   --------
**
** TEST AND MODIFY FLAGS
**
  -1     flQIET     Quiet Mode
  -2     flBEEP     Beep On
```

```
  -3     flCTON     Continuous On
  -4     flINX      Inexact result
  -5     flUNF      Underflow
  -6     flOVF      Overflow
  -7     flDVZ      Divide by Zero
  -8     flIVL      Invalid operation
  -9     flUSER     User Mode set
 -10     flRAD      RAD trig mode
 -11     flINFR     Round to Infinity
 -12     flNEGR     Negative Round
 -13     flFXEN     FIX/ENG flag
 -14     flSCEN     SCI/ENG flag
 -15     flLC       Lower Case enabled
 -16     flBASE     Base Option   (high bit!)
 -17     flDG0      Display digit bit 0
 -18     flDG1      Display digit bit 1
 -19     flDG2      Display digit bit 2
 -20     flDG3      Display digit bit 3
 -21     flPDWN     Don't pwr loop down autom.
 -22     flEXTD     Use extended addressing
 -23     flEOT      Entry terminated by EOT
 -24     flNZ4       "
 -25     flBPLD     Beep LOUD
 -26     flNOPR     Don't Prompt
 -27                Alternate message language
**
** TEST ONLY FLAGS
**
 -42     flMPI      Module pulled
 -43     flDORM     Machine is dormant
 -44     flRTN      Always Return from MEMERR
 -45     flCLOC     Clock mode (1 sec update)
 -46     flEXAC     EXACT flag
 -47     flCMDS     Command Stack Active
 -48     flCTRL     Control key hit
 -49     flPWDN     DSLEEP called from PWR down
 -50     flMKOF     Req set TRNOF in MAINLP
 -51     flTNOF     Turnoff at MAINLP
 -52     flVIEW     VIEW key pressed
 -53                Reserved for Future Use
 -54                Reserved for Future Use
 -55                Reserved for Future Use
 -56                Reserved for Future Use
 -57     flAC       AC Annunciator
 -58     flUSRX     User Mode suspend
 -59     flRPTD     Key repeated
 -60     flALRM     Alarm Annunciator
 -61     flBAT      Low Battery Annunciator
 -62     flPRGM     Program Annunciator
 -63     flSUSP     Suspend Annunciator
 -64     flCALC     Calc Mode Annunciator
```

GLOSSARY	APPENDIX A

Absolute address
An address which is equal to the exact physical address of the location it designates.

BET
Abbreviation for "Branch Every Time." Refers to a GOC or GONC machine instruction which is known to always be equivalent to a GOTO because the state of the carry is predicatably set or clear, respectively. This is a packing technique which saves 1 nibble (GOTO takes 4 nibbles while GOC and GONC take only 3) but should be used with caution and should always be clearly labeled as a BET.

CALL stack
The CALL stack is used to store the local environment of a program or subprogram which has called a subprogram or user-defined function.

File chain
The data structure by which the HP-71 file system stores multiple files in main RAM or in independent RAM.

General purpose buffer
Alternate name for a system buffer.

Independent RAM
A plug-in RAM memory module which has been declared as an independent file system by the FREE PORT command.

IRAM
Abbreviation for Independent RAM.

I/O buffer
Alternate name for a system buffer.

Main loop
The outermost loop of the HP-71 operating system; the control loop. See the "System Control" chapter for further information.

PC
Abbreviation for "Program Counter." The CPU program counter register is referred to as the PC register, and contains the address of the next instruction the CPU will execute. When the operating system is interpreting a BASIC file, the address of the next token to be interpreted is also loosely referred to as the "PC" of the interpreter, and it is stored in register D0.

RAM
Abbreviation for random access memory.

Relative address
An offset address; usually used to describe the contents of a field which contains an absolute address from which the absolute address of the field start has been subtracted, generating a positive or negative offset.

ROM
Abbreviation for read-only memory.

Saturn
The HP internal code name for the CPU and bus architecture used in the HP-71.

System buffer
An operating system resource in main RAM which can be created by a LEX file for data storage. Sometimes referred to as an I/O buffer or a general purpose buffer.

Titan
The HP internal code name for the HP-71 computer.

Version 79.10.13 of RUNIT's INDEX program

C

F

K

L

M

P

S

V

W

X

Y