Part IV Technical Reference

Contents

Figures and Tables vii

PREFACE ix

CHAPTER 1 The Pascal Environment 1
System Memory Use 2
Useful Memory Addresses 6
Main Memory Pointers 7
Auxiliary Memory Pointers 7
Identification Flags 8
Screen Mode Flag 8
Flag to Check the Pascal System Version 8
Flag to Check the Interpreter Version 8
Flag to Check the Computer Type 9
The "Ignore External Terminal" Flag 10
Interpreter Addresses 10
How Pascal Loads Code Segments 11

CHAPTER 2 Disk Files 13
Reading a Disk Directory 14
Textfile Structure 16
Codefile Structure 17
Segments 18
Segment Dictionaries 21
Segment Numbers 27

Interface Text 28
Code Parts 30
Procedure Dictionaries 32
Procedures 32
Attribute Tables 33
P-Code Procedure Attribute Tables 34
Assembly-Language Procedure Attribute Tables 35
Relocation Tables 37
Linker Information 38
Linker Information Fields 40
Global Address Linker Information Types 40
Host-Communication Linker Information Types 41
Procedure and Function Linker Information Types 43
Miscellaneous Linker Information Types 43

CHAPTER 3 The P-Machine 45
The Evaluation Stack 47
Registers 47
The Program Stack and the Heap 48
SYSCOM 49
The Segment Table 50
Activation Records 51
Markstacks 53

CHAPTER 4 The P-Machine Instruction Set 57
Instruction Formats 58
Operand Formats 58
Formats of Variables on the Stack 59
Boolean 59
Integer 59
Long Integer 59
Scalar (User-Defined) 59
Char 60
Real 60
Pointer 60
Set 60
Records and Arrays 61
Strings 61
Formats of Constants in P-Code 61
Conventions and Notation 62
P-Machine Instructions 62
One-Word Loads and Stores 62
Constant 62
Local 63
Global 63
Intermediate 63
Indirect 64
Extended 64
Multiple-Word Loads and Stores (Sets and Reals) 64
Byte Array Handling 65
String Handling 65
Record and Array Handling 66

Dynamic Variable Allocation 67
Top-of-Stack Arithmetic 68
Integers 68
Noninteger Comparisons 69
Reals 69
Strings 70
Logical 70
Sets 71
Byte Arrays 72
Record and Word Array Comparisons 72
Jumps 72
Procedure and Function Calls 73
System Support Procedures 75
Miscellaneous 76
Numerical Listing of Opcodes 76

APPENDIX 4A Memory Maps 81
64K System Memory 82
128K System Memory 83
128K System Auxiliary Memory 84
Code Segments in a Codefile 85
Blocks in a Code Segment 86
Correlation Between Programs and Codefiles 87
Segment Dictionary 88
Interface Text in a Codefile 89
Code Part of a Code Segment 90
Procedure Code Structure 91
P-Code Procedure Attribute Table 91
6502 Procedure Attribute Table 92
Bytes and Words 93
Program Stack 94
Segment Table 95
Activation Record 96
Variable Allocation in an Activation Record 96

Figures and Tables

CHAPTER 1 — The Pascal Environment 1

Figure 1-1 The Pascal 64K System 3
Figure 1-2 The Pascal 128K System: Main Memory 4
Figure 1-3 The Pascal 128K System: Auxiliary Memory 5
Table 1-1 Version Flags Set at Location −16606 ($BF22 Hex) 9
Table 1-2 Hardware Identification Bit Settings 10

CHAPTER 2 — Disk Files 13

Figure 2-1 A Typical Codefile on Disk 17
Figure 2-2 A Typical Codefile 19
Figure 2-3 Correlation Between Programs and Segments in Codefiles 20
Figure 2-4 A Segment Dictionary 24
Table 2-1 Segment Number Assignment 28
Figure 2-5 Construction of Interface Text in a Codefile 29
Figure 2-6 The Code Part of a Code Segment 31
Figure 2-7 A Typical Procedure 33
Figure 2-8 P-Code Procedure Attribute Table 34
Figure 2-9 An Assembly-Language Procedure Attribute Table 36

CHAPTER 3 — The P-Machine 45

Figure 3-1 Relationship of Words and Bytes 46
Figure 3-2 The Program Stack and Heap With Four Active Procedures 49
Figure 3-3 The Segment Table 51
Figure 3-4 An Activation Record 52

Figure 3-5 The Order of Local Variable Allocation in an Activation Record 53

CHAPTER 4 The P-Machine Instruction Set 57

Table 4-1 P-Codes in Numerical Order 76

APPENDIX 4A Memory Maps 81

64K System Memory 82
128K System Memory 83
128K System Auxiliary Memory 84
Code Segments in a Codefile 85
Blocks in a Code Segment 86
Correlation Between Programs and Codefiles 87
Segment Dictionary 88
Interface Text in a Codefile 89
Code Part of a Code Segment 90
Procedure Code Structure 91
P-Code Procedure Attribute Table 91
6502 Procedure Attribute Table 92
Bytes and Words 93
Program Stack 94
Segment Table 95
Activation Record 96
Variable Allocation in an Activation Record 96

Preface

The *Technical Reference* is written for more advanced users of the Apple II Pascal 1.3 system. It describes the structure and operation of the P-machine and parts of the Pascal operating system. Before you use the information it contains, you should be familiar with the other parts of the *Apple II Pascal 1.3 Manual*.

Many of the concepts explained in this *Technical Reference* are intimately interrelated. You should first briefly read the entire book to gain an appreciation of how the concepts are interrelated before attempting to understand any specific concept in detail.

Here is an overview of what this part of the *Apple II Pascal 1.3 Manual* contains. It consists of four chapters:

- **Chapter 1: The Pascal Environment.** Describes how the Pascal 1.3 system uses the Apple II memory and gives some useful memory addresses. Describes how code segments are loaded into memory.
- **Chapter 2: Diskfiles.** Describes in detail the structure and format of Pascal disk directories, textfiles, and codefiles.
- **Chapter 3: The P-machine.** Introduces the concept of the Pascal P-Machine. Describes how the P-Machine resides and works within the structure of Apple II memory.
- **Chapter 4: The P-machine Instruction Set.** Gives opcodes and implementation information for all P-machine instructions.

Chapter 1 The Pascal Environment

The Apple Pascal system is a version of the UCSD Pascal system, a pseudomachine-based implementation of Pascal. This means that the Compiler converts Pascal program text into compact **pseudocode** or **P-code** to be executed by the **pseudomachine** or **P-machine**. The P-machine is implemented by the Pascal Interpreter—a program written in the **native code** of the Apple II's 6502 microprocessor. Every host computer operating under a version of UCSD Pascal has such an Interpreter that makes the host computer appear, from the viewpoint of a program being executed, to be a P-machine. The Interpreter is contained in the file SYSTEM.APPLE.

The Pascal operating system and various utility programs are also written in Pascal and run on the same Interpreter. The Pascal Compiler, Assembler, and Linker together produce completed **codefiles** of Pascal programs. Pascal codefiles are stored on external storage media, such as disks. The structure of codefiles is explained in Chapter 2.

To execute a Pascal program, Pascal loads the code of the program's main segment from the codefile into memory. It then begins executing the program code, one instruction at a time. As it finds that additional segments of the disk codefile are needed in memory for execution of the program, it loads the necessary segments. The process by which Pascal loads executable code into memory is explained at the end of this chapter.

System Memory Use

Apple Pascal comes in two versions: a 64K system and a 128K system. Either system will run on Apple II computers with 128K of memory; only the 64K system will run on 64K models. The process of loading one or the other system is described in Part I of this manual, *Getting Started.*

The two systems use memory differently. 64K system memory usage is shown in Figure 1-1. 128K system memory usage is shown in Figures 1-2 and 1-3.

Figure 1-1. The Pascal 64K System

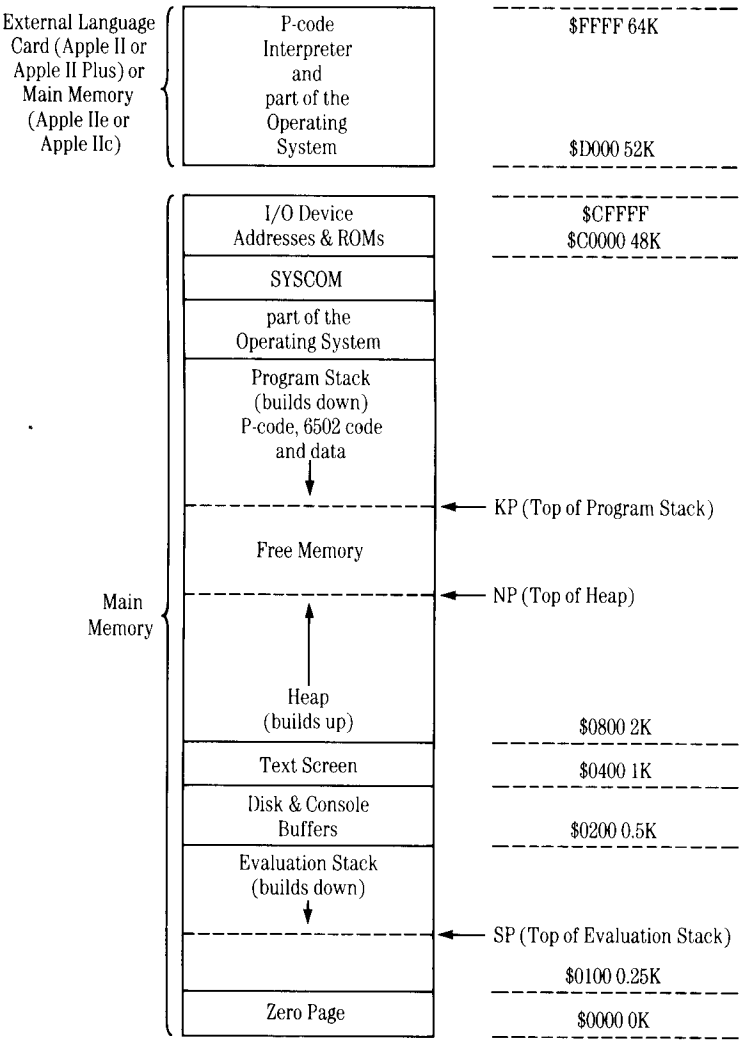


Figure 1-2. The Pascal 128K System: Main Memory

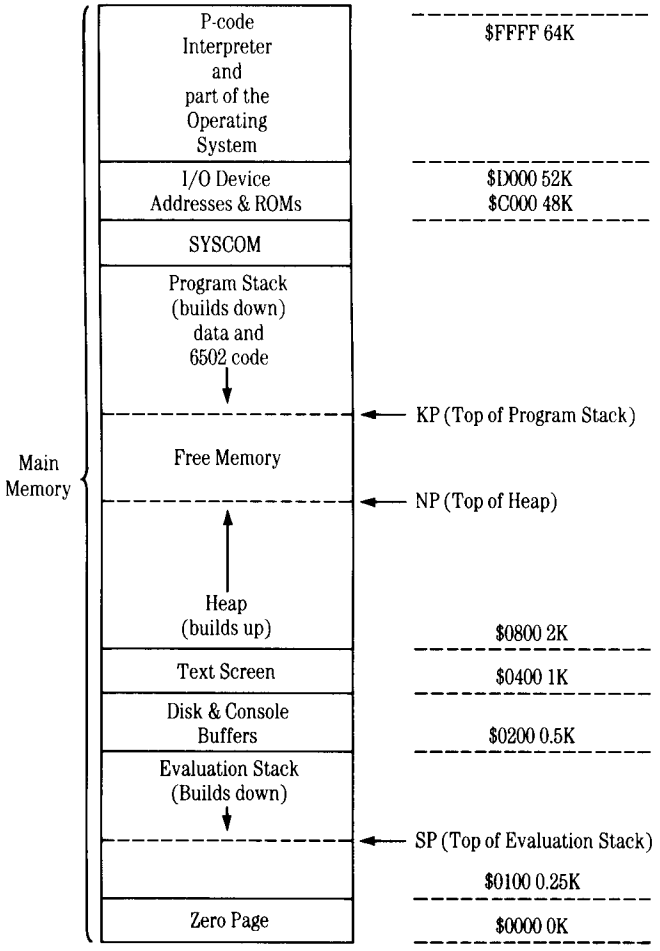
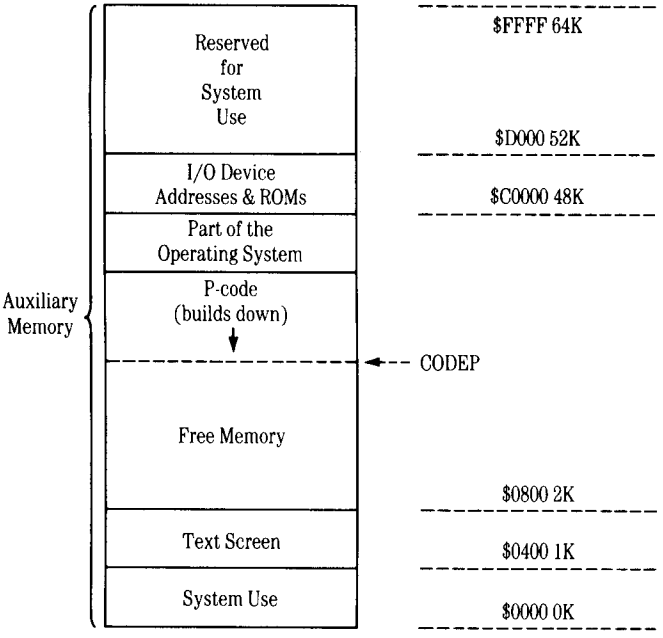


Figure 1-3. The Pascal 128K System: Auxiliary Memory



The principal difference in memory usage between the 64K and 128K Apple Pascal systems is this. In the 64K system, a single area of free memory is used by P-code, 6502 code, and data. In the 128K system there are two areas of free memory; one is used only by P-code, and the other is used only by 6502 code and data.

Here are some additional points about memory usage to supplement the information in Figures 1-1, 1-2, and 1-3:

- In both systems, pointer NP points to the top of the heap space. Pointer KP points to the top of the program stack. In the 64K system the stack contains P-code, 6502 code, and data; in the 128K system it contains only 6502 code and data. In the 128K system another pointer, CODEP, points to the top of the area used by P-code. The locations of these pointers are given below under "Useful Memory Addresses."
- The beginning of the heap ($0800) is shown for 80-column mode. In 40-column mode the heap starts 1K higher, at $0C00.

- If a program contains the declaration USES TURTLEGRAPHICS, the beginning of the heap is set to $4000 (16K) when the TURTLEGRAPHICS unit is loaded. The space below that is used for the high-resolution graphics memory.
- In the 128K system, using either Compiler swapping or operating system swapping adds to the space available for P-code, not the space available for data and 6502 code.

Useful Memory Addresses

There are several addresses in machine memory that contain information you may find useful. You can read them with the PEEK function; in some cases you can change their values with the POKE procedure. PEEK and POKE are described in Part III, Chapter 16. All memory addresses are given in hexadecimal form.

$005A **NP:** Two-byte pointer to top of Pascal heap in main memory.

$005C **KP:** Two-byte pointer to top of Pascal program stack in main memory.

$0060 **CODEP:** Two-byte pointer to lowest used word in auxiliary memory. Its possible value range is $C000 to $800.

$0062 **CODELOW:** Two bytes containing the lower limit value for CODEP. Memory below this point is reserved.

$BF0E **Screen Mode:** One-byte flag to show whether current screen display is 80 columns or 40 columns wide.

$BF21 **Pascal System:** One-byte flag to show which version of Apple Pascal is being used.

$BF22 **Interpreter Version:** One-byte flag to show which Pascal Interpreter is running, and other information.

$BF31 **Computer Type:** One-byte flag to show which Apple computer model is being used.

Where a *two-byte address* is shown, the memory location given is that of the less-significant byte. The more-significant byte is located one memory address higher.

Use of these memory locations is discussed in more detail in the sections that follow.

Main Memory Pointers

NP points to the top of the Pascal heap. The heap grows toward higher memory addresses from these locations:

- $0800 if the screen output is in 80-column mode.
- $0C00 if the screen output is in 40-column mode.
- $4000 if TURTLEGRAPHICS is being used.

The heap is discussed in more detail in Chapter 3.

KP points to the top of the Pascal program stack. The stack grows toward lower memory addresses. Its starting point is below $C000; the actual location depends on how much of the operating system is stored between it and $C000.

When the values of NP and KP meet, free memory is used up and an execution error occurs.

Auxiliary Memory Pointers

The Pascal 128K system uses CODEP and CODELOW to manage use of auxiliary memory. CODEP points to the lowest used word in the auxiliary memory space. CODELOW contains the lowest permissible value for CODEP. CODELOW defaults to $0800.

Because CODEP points to the lowest used word in the auxiliary memory space, it begins with the value of $C000 and works down until it hits the value CODELOW.

Your program can examine CODEP and CODELOW if it needs to. If your program runs under the 128K system, it cannot change CODEP, but it can change CODELOW if it uses part of the auxiliary memory. For example, to execute a program that uses the 560-dot high-resolution screen, you would change CODELOW to $4000 and then change it back to its original value after the program has run.

If you are using the 64K Pascal system on a machine with 128K of memory, you can use CODEP as a zero-page pointer to the auxiliary memory space. This feature is useful if you are managing this space yourself, rather than using the Pascal 128K system to manage it.

Here are several important reminders about your use of these variables:

- You must use even numbers when giving values to these variables because they point to words, not bytes.
- The system does not restore CODELOW or CODEP to their original values after executing your program. Whenever you have changed one of these variables, be sure to put the value back to what it was before your program ends.
- If your program runs under the 128K system, it can change only CODELOW; CODEP is changed only by the Pascal system.

Identification Flags

Both the 64K and the 128K system set four identification flags in main memory. Your program can use PEEK to read these flags. They are described below.

Screen Mode Flag

A byte at memory location −16626 ($BF0E hex) tells whether Pascal is operating in 40-column display mode or 80-column mode. If it is in 40-column mode, the value of the byte is 0; otherwise it is 4.

Flag to Check the Pascal System Version

When Pascal is started up, a flag is set at memory address −16607 ($BF21 hex) to identify which Pascal version is the one being used.

- If Pascal 1.3 is operating, the value of the byte at that location is *4*.
- If Pascal 1.2 is operating, the value of the byte at that location is *3*.
- If Pascal 1.1 is operating, the value of the byte at that location is *2*.

Flag to Check the Interpreter Version

To identify which Pascal Interpreter is executing, another flag is set at startup time, at memory address −16606 ($BF22 hex). This flag uses different bit settings to identify the variations being supported, as Table 1-1 shows.

Table 1-1. Version Flags Set at Location −16606 ($BF22 Hex)

Bit	Set To	Indicates
0	0	The Pascal development system is executing.
0	1	The Pascal run-time system is executing.
1	1	Floating-point operations are not supported.
2	1	Operations using sets are not supported.
5 6	1 0	The 48K Pascal Interpreter is executing.
5 6	0 0	The 64K Pascal Interpreter is executing.
5 6	0 1	The 128K Pascal Interpreter is executing.
7	0	All console output is directed to the text screen pages, an external terminal, or an 80-column card.
7	1	All console output is directed to the high-resolution pages.

Flag to Check the Computer Type

By identifying which machine it is running on, an application program for the Apple IIe or Apple IIc can take advantage of the computer's unique features but retain the capacity to run on the Apple II or Apple II Plus. Memory location −16591 ($BF31 hexadecimal) contains a flag you may use to determine from within a program whether the computer is an Apple II, Apple IIe, or Apple IIc. If the computer is a IIe, this same memory location also specifies whether the computer has an 80-column text card and whether it has the auxiliary 64K of RAM memory available on the Apple Extended 80-Column Text Card.

The flag bit settings listed in Table 1-2 are made whenever the Pascal system starts up. For the systems listed in the left column, the byte at memory location −16591 ($BF31 hex) has the bit settings shown on the right. Bits not listed are set to 0.

Table 1-2. Hardware Identification Bit Settings

System	Bit 7	Bit 6	Bit 1	Bit 0
Apple IIc	1	1	1	1
Apple IIe	1	0	0	0
with 80-column text card	1	0	0	1
with 128K memory	1	0	1	1
Apple II or II Plus	0	0	0	0

For Future Use: It is possible for the computer-type flag to be set so that bits 7, 6, 5, 1, and 0 are all set to 1. This value is currently unused, but is reserved for future use.

The "Ignore External Terminal" Flag

By setting a bit located on the Pascal startup disk, you can force Pascal to operate in 40-column mode, even though the hardware configuration may have 80-column capabilities. This bit is located in the directory area of the startup disk at block 2, byte 25, bit 3 (counting always from 0). If it is set to 1, Pascal ignores any 80-column firmware and operates only in 40-column mode.

You can use the utility program SET40COLS to set the "ignore external terminal" flag. SET40COLS is described in Part II of this manual, Chapter 9.

Interpreter Addresses

The Pascal Interpreter contains a 3-word table that gives entry points of interest to the advanced programmer. You can access these entry points by using the Assembler directive .INTERP. The following list gives their meanings.

.INTERP+0 — Address of the Interpreter's run-time execution error posting routine. The user can load the A register with the error number and execute the 6502 instruction `JSR @.INTERP` to invoke the system error message routine.

.INTER+2 — Address of the BIOS (input/output handling routine) dispatching table

.INTERP+4	Address of the location that contains the address of SYSCOM, the area used for communication between the Interpreter and the Pascal operating system

How Pascal Loads Code Segments

Apple Pascal loads code segments from disk into memory two ways, depending on whether the 64K system or the 128K system is running.

With the 64K system, Pascal simply transfers each segment from the codefile to the program stack, moving the KP pointer down as it does so.

With the 128K system, segment loading is more complicated. Pascal follows these steps:

1. It transfers as much of the segment as it can, in 512-byte blocks, from the codefile to the stack/heap space. For this transfer to work, there must be at least 512 bytes of memory available.
2. It transfers the segment from the stack/heap space to auxiliary memory.
3. It repeats steps 1 and 2 until the segment is completely loaded.
4. If the segment contains 6502 code, Pascal copies the 6502 code to the program stack.
5. If Pascal has copied 6502 code out of the segment, it moves the segment's P-code toward higher memory addresses, thereby reclaiming the space occupied by the 6502 code.

The following are potential errors that may occur in the segment loading process:

- With the 64K system, there may not be enough stack space to hold the segment.
- With the 128K system, there may be less than 512 bytes of space between the stack and heap pointers.
- With the 128K system, there may not be enough stack space to hold the 6502 code.
- With the 128K system, there may not be enough space in auxiliary memory to hold the entire segment.

If one of the first three errors occurs, the execution error message

```
Stack overflow
```

or

```
Execution error #4
```

will appear on your screen.

If the fourth error occurs, the execution error message

```
Codespace overflow
```

or

```
Execution error #16
```

will appear on your screen.

Chapter 2 Disk Files

This chapter describes how Apple Pascal stores text and code in disk files. It covers three major areas:

- How to access a Pascal disk directory
- The structure of textfiles created by the Apple Pascal Editor
- The structure of codefiles created by the Apple Pascal Compiler and Linker

Reading a Disk Directory

A disk directory is simply an array of records, each of which contains information about one file stored on the disk. By declaring a variable of congruent type, you can use UNITREAD to transfer the contents of a disk directory to memory. Your program can then access it. Here are the necessary declarations:

```
{First, some general declarations relating to disk directories:}

CONST maxdir = 77;       {Maximum number of entries in directory}
      vidleng = 7;       {Number of characters in volume ID}
      tidleng = 15;      {Number of characters in title ID}
      fblksize = 512;    {Standard disk block length}
      dirblk = 2;        {Directory starts at this disk-block address}

TYPE  daterec = PACKED RECORD      {Volume/file data mark}
                  month: 0..12;    {0 = meaningless date}
                  day:   0..31;    {Day of month}
                  year:  0..100    {100 = dated volume is temporary}
                END {daterec};
      vid = STRING [vidleng];      {Volume ID}
      dirrange = 0..maxdir;        {Possible number of files on disk}
      tid = STRING [tidleng];      {Title ID}
      filekind = [untypedfile, xdskfile, codefile, textfile, infofile,
                  datafile, graffile, fotofile, securedir];

{Now, the actual layout of the directory entry for each file stored on
the disk, plus the type declaration for the directory as a whole.  Each
entry is a packed record with a variant part, and the whole directory
is an array of such records.}

      direntry =
        PACKED RECORD
          dfirstblk: integer;   {1st physical disk address}
          dlastblk:  integer;   {Points to block after last used block}
          CASE dfkind: filekind OF
            securedir, untypedfile:  {Volume info—only in dir[0]}
              [filler1:    0..2048;  {Waste 13 bits for compatibility}
               dvid:       vid;      {Name of disk volume}
               deovblk:    integer;  {Last block in volume}
               dnumfiles:  dirrange; {Number of files in directory}
               dloadtime:  integer;  {Time of last access}
               dlastboot:  daterec]; {Most recent date setting}
            xdskfile, codefile, textfile, infofile, datafile,
            graffile, fotofile:      {Regular file info}
              [filler2:    0..1024;  {Waste 12 bits for compatibility}
               status:     boolean;  {For filer wildcards}
               dtid:       tid;      {Name of file}
               dlastbyte:  1..fblksize; {Num bytes in last file block}
               daccess:    daterec]  {Date of last modification}
        END
      directory = ARRAY [dirrange] OF direntry;
```

Having made the foregoing constant and type declarations, your program may now declare an array variable DIRINFO of type DIRECTORY and an integer variable DEVNUM to supply the volume number of a disk drive. UNITREAD will transfer the contents of the disk directory to DIRINFO:

```
VAR  DIRINFO : DIRECTORY;    {Array variable to hold directory info}
     DEVNUM : INTEGER;       {Volume number of disk drive}
        ...

        UNITREAD (DEVNUM, DIRINFO, SIZEOF(DIRINFO), DIRBLK);
```

Textfile Structure

The special format of a textfile is as follows:

- There are two blocks (1024 bytes) of **header** information at the beginning of the file. This information is used by the Pascal Editor. The Pascal system creates the header page when a user program opens a textfile. The header page is transferred only during disk-to-disk transfers; transfers to character devices, such as the console or printer, always omit the header page.
- The rest of the file consists of two-block **pages**. Each page contains lines of text, separated from each other by RETURN characters (ASCII 13). No line ever crosses a page boundary; thus a page contains only whole lines. After the last line on a page, the remainder of the page is filled with NUL characters (ASCII 00). READ and READLN skip the NUL characters, and WRITE and WRITELN provide them automatically. Thus this page formatting is normally invisible to a Pascal program.
- A sequence of leading spaces in a line may be compressed to a DLE-blank code. This code consists of a DLE control character (ASCII 16) followed by one byte containing the number of spaces to indent plus 32 (decimal). Using this code saves a considerable amount of space in files where indentation occurs frequently. The Editor is the main creator of DLE-blank codes; it usually outputs a DLE-blank code where a sequence of spaces occurs at the beginning of a line. However, the DLE-blank code is optional; some lines may have it and others may have space characters instead. Also, a line with no indentation may or may not be preceded by a DLE character and an indent code value of 32 (meaning 0 indentation).

GET, READ, and READLN convert DLE-blank coding to actual spaces on input from a textfile to a file variable of type TEXT or INTERACTIVE; thus the compression of spaces is also normally invisible to a Pascal program.

Various parts of the system that deal with files of characters (such as the Editor and the Compiler) are designed to take advantage of the special textfile format. For most purposes, it is recommended that you use the textfile type for any character files created by your programs. The name of every textfile must end in .TEXT.

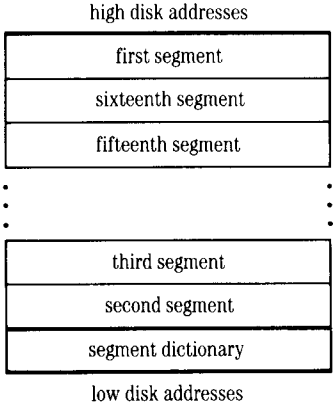
Codefile Structure

The remainder of this chapter is about Apple Pascal **codefiles**. A codefile may be any of the following:

- **Linked files** composed of **segments** ready for execution.
- **Library files** with units that may be used by programs in other codefiles.
- **Unlinked files** created by the Apple Pascal Compiler or Assembler.

A typical disk codefile resulting from the compilation of a program is diagrammed in Figure 2-1.

Figure 2-1. A Typical Codefile on Disk



All codefiles (linked and unlinked) consist of a **segment dictionary** in block 0 of the codefile, followed by a sequence of one to 16 code segments. The *host program* is compiled into one code segment, and each *SEGMENT procedure*, *SEGMENT function*, or *Program Unit* is translated into another code segment. The ordering of code segments in the codefile (from low disk address to high disk address) is determined by the order in which the Compiler encounters the executable code of each SEGMENT procedure, SEGMENT function, or Program Unit when compiling a program. This order may be changed by using the LIBRARY program described in Part II, Chapter 8.

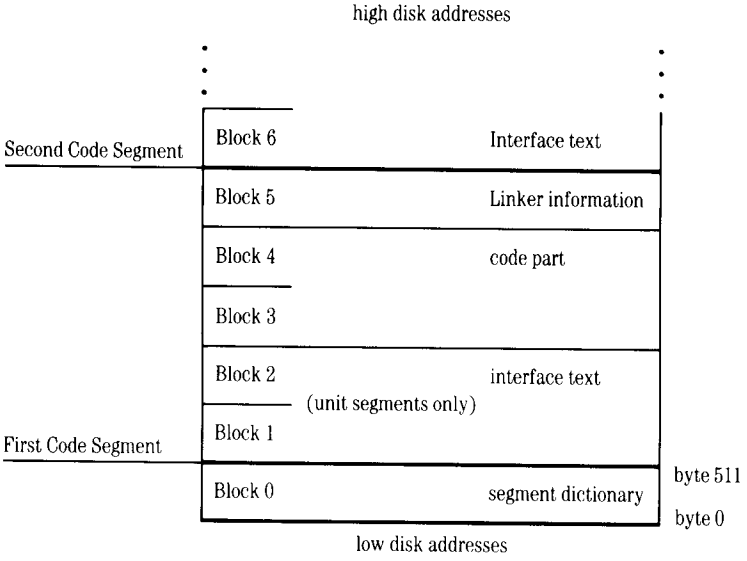
Each segment begins on a boundary between disk blocks (a **block** is 512 contiguous 8-bit bytes). Any segment may occupy up to 64 blocks.

Segments

A **segment** is either a **code segment** or a **data segment**. Program code is stored in code segments. Every program consists of at least one code segment, and some programs consist of many code segments. A code segment may contain either P-code, 6502 code, or a combination of both. Code segments may have three parts: **interface text**, actual P-code and/or 6502 code, and **Linker information** (Figure 2-2). These parts appear in this order on the disk, although not all types of code segments have all three parts. For example, interface text is present only in the code segments of Program Units. Code segments may be either linked or unlinked.

Data segments are areas of memory that are set aside at execution time as storage space for the local data of **Intrinsic Units**. In a disk codefile, data segments have only an entry in the segment dictionary: they do not occupy any blocks on the disk because they have no code part, interface text, or Linker information associated with them.

Figure 2-2. A Typical Codefile

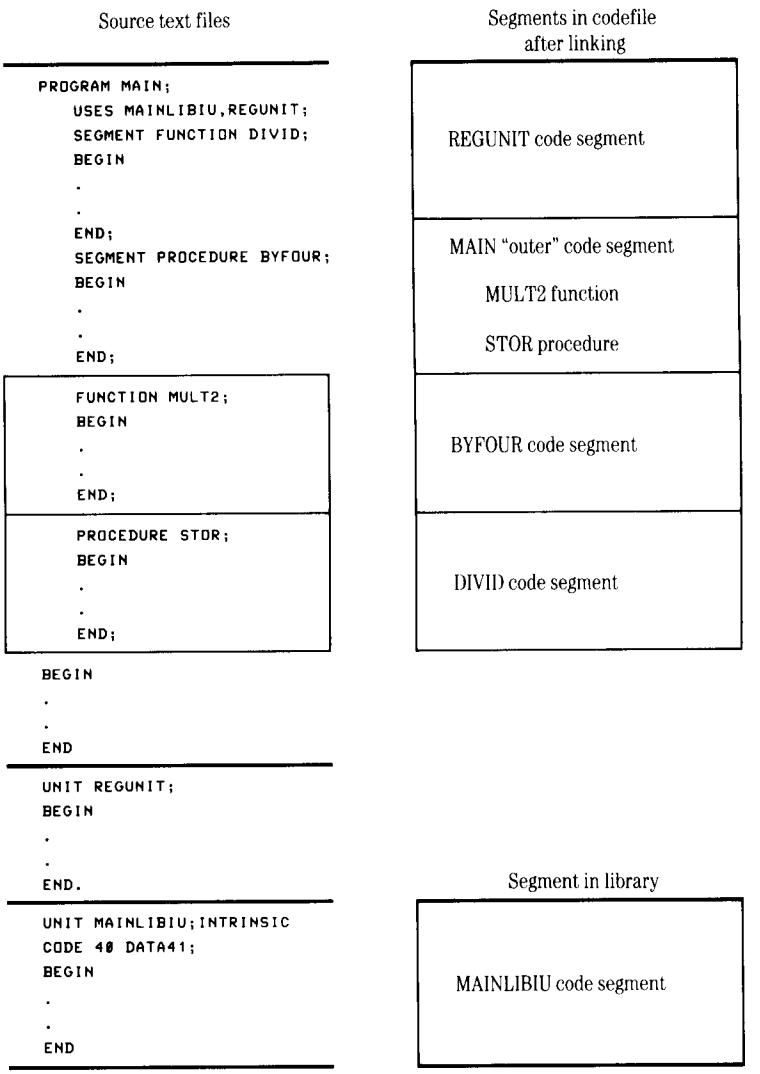


By the Way: Figure 2-2 is not meant to imply that all code segments are five blocks long; the code part of a segment can contain up to 64 blocks.

Whenever a complete program codefile is produced by the Compiler (and Assembler and Linker, if necessary), the following events occur:

- The user program or Program Unit results in one code segment in the codefile. This includes the user program's non-SEGMENT procedures and functions (MULT2 and STOR in Figure 2-3), and the user program body itself (MAIN in Figure 2-3).
- Each Pascal SEGMENT procedure or function results in another code segment in the codefile (BYFOUR and DIVID below).
- Each Regular Unit that the program USES is linked with the codefile and results in a code segment in the codefile (REGUNIT below). Each Intrinsic Unit that the program USES does not produce additional code segments in the program's codefile. Intrinsic Units are held as segments in program libraries, shared libraries, and the SYSTEM.LIBRARY file, and accessed by the program at *execution time* (MAINLIBIU below).

Figure 2-3. Correlation Between Programs and Segments in Codefiles



Segments are not nested in codefiles as they are in programs. Instead, every segment is a separate contiguous area of code and does not contain any other segments. For example, if a SEGMENT procedure contains another SEGMENT procedure, the compiled result comprises two distinct code segments, even though they are nested lexically in the source program.

Segmenting a program does not change the computation it performs. When a SEGMENT procedure, SEGMENT function, or Intrinsic Unit is called during the execution of a program, the Pascal Interpreter checks to see if that segment is already in memory due to a previous (and still active) invocation of the segment. If it is, control is transferred and execution proceeds; if not, the appropriate code segment is loaded into memory from the disk codefile before the transfer of control takes place. When no more active invocations of a segment exist, its code is removed from memory.

The following sections describe the portions of a code segment in greater detail. First the segment dictionary is described. Then the parts of a code segment are presented in the order in which they may occur in a codefile: the interface text, the code part, and finally the Linker information. The code part description is divided into sections describing the similarities and differences between the code parts of P-code and assembly-language procedures.

Segment Dictionaries

Every codefile (including library files) has a **segment dictionary** in block 0 that contains information needed by the Pascal system to load and execute the segments in that codefile. A segment dictionary has 16 **slots**, each of which either contains information about one segment, or is empty. Each non-empty slot includes the segment's name, kind, size (in bytes), and location in the codefile. The location of a code segment is given as the block number of the first block in the code part. Blocks in a codefile are numbered sequentially from 0, with block 0 as the segment dictionary. The location of a data segment is given as 0.

The information that describes each segment is contained in five different arrays within the segment dictionary. All information describing a segment is retrieved by selecting corresponding elements from each of these arrays.

Because a segment dictionary has 16 slots, numbered 0 through 15, one codefile can contain at most 16 segments. Intrinsic Units used by a program do not require entries in the segment dictionary of the program's codefile, because Intrinsic Unit code segments are in a library file, and appear in the

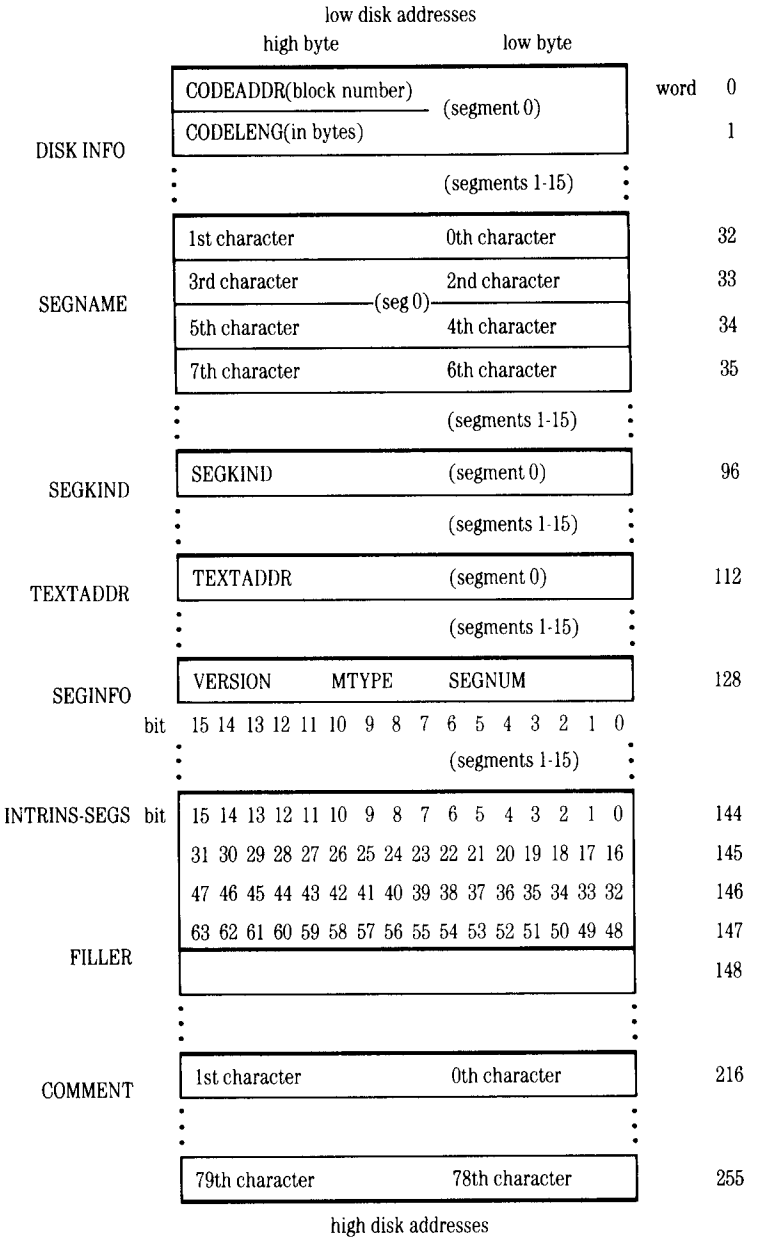
library file's segment dictionary. Therefore, a program can have a maximum of 16 segments, not counting segments from Intrinsic Units. Counting Intrinsic Units, the maximum number of segments is limited by the total number of segment numbers in the system—64 for the 128K system, 32 for the 64K system. However, the system reserves 12 segment numbers (0, 2 through 6, and 58 through 63) for its own use. The remaining segments may appear in a program codefile, a program library file, SYSTEM.LIBRARY, or library files specified in a *Library Name File*. Each of these codefiles can contain a maximum of 16 segments.

The following Pascal record declaration represents a segment dictionary.

```
RECORD

  DISKINFO:  ARRAY[0..15] OF
     RECORD
       CODEADDR:  INTEGER;  {location of code part}
       CODELENG:  INTEGER   {length of code part}
     END;

  SEGNAME:  ARRAY[0..15] OF PACKED ARRAY[0..7] OF CHAR;  {segment name}
  SEGKIND:  ARRAY [0..15] OF     {type of segment}
            (LINKED,             {fully executable segment}
             HOSTSEG,            {user program code segment}
             SEGPROC,            {unused}
             UNITSEG,            {compiled Regular Unit}
             SEPRTSEG,           {separate procedures and functions}
             UNLINKED-INTRINS,   {unlinked Intrinsic Unit}
             LINKED-INTRINS,     {linked Intrinsic Unit}
             DATASEG);           {data segment}

  TEXTADDR:  ARRAY[0..15] OF INTEGER;  {address of the first
                                        block of interface text, if any}

  SEGINFO:  PACKED ARRAY[0..15] OF PACKED RECORD
              SEGNUM:  0..255; {segment number}
              MTYPE:  0..15;   {machine type}
              UNUSED:  0..1;   {unused}
              VERSION:  0..7   {version number}
            END;

  INTRINS-SEGS:  SET OF 0..ss;  {intrinsic segment numbers needed for
                                 execution.  ss=63 for 128K system;
                                 ss=31 for 64K system}

  FILLER:  ARRAY [0..ff] OF INTEGER; {unused bytes filled with
                                      zeros.  ff=67 for 128K
                                      system, ff=69 for 64K}

  COMMENT:  PACKED ARRAY [0..79] OF CHAR  {comment}

END;
```

Figure 2-4 shows the structure of a segment dictionary.

Figure 2-4. A Segment Dictionary



By the Way: Figure 2-4 shows lower addresses at the top (in contrast to others in this manual) to match the structure of the Pascal segment dictionary declaration.

Each segment dictionary is composed of

- Five 16-element arrays—one element for each segment slot in the segment dictionary of a codefile;
- Information about the intrinsic segments used by the codefile;
- An optional comment.

Each element in the DISKINFO array consists of two *words* that describe the length and location of the segment within the codefile. For code segments, the CODEADDR field contains the block number of the start of the code part, and the CODELENG field contains the number of bytes in the code part of the segment. For data segments, the CODEADDR field is 0, and the CODELENG field contains the number of bytes to be allocated for data at execution time (the length of the data segment). Unused slots have their CODEADDR and CODELENG fields set to 0 (CODELENG=0 defines an empty slot).

Each element of the SEGNAME array is an eight-character array that contains the first eight characters of the user program, unit, SEGMENT procedure, SEGMENT function, or assembly-language procedure name that was translated into the corresponding segment. If the name is shorter than eight characters, it is padded on the right by spaces; if the name is longer than eight characters, it is truncated to the first eight characters. Unused segment slots have SEGNAME fields filled with eight ASCII space characters.

The SEGKIND array describes the type of segment. The possible values are as follows:

0: LINKED. A fully-executable segment. Either all references to Regular or Intrinsic Units have been resolved by the Linker, or none were present.

1: HOSTSEG. The main segment of a user program with unresolved external references.

2: SEGPROC. A SEGMENT procedure or function. This type is currently not used.

3: UNITSEG. A compiled Regular (as opposed to Intrinsic) Unit.

4: SEPRTSEG. Separately assembled procedures or functions, including EXTERNAL functions and procedures, and mixed segments of linked Pascal and assembly-language code. Assembly-language codefiles are always of this type.

5: UNLINKED-INTRINS. An Intrinsic Unit containing unresolved calls to assembly-language procedures or functions.

6: LINKED-INTRINS. An Intrinsic Unit properly linked with its called procedures and functions.

7: DATASEG. A data segment of an Intrinsic Unit. The segment dictionary entry specifies the amount of data space (in bytes) to allocate.

The TEXTADDR array of integers contains pointers to the block number of the start of the interface text of each Regular or Intrinsic Unit. The last block number of the interface text can be calculated by subtracting 1 from the value in the corresponding CODEADDR field. Interface text is described in detail below. Only unit segments have interface text; the TEXTADDR field is 0 for all other types of segments.

The SEGINFO array contains one word of additional information about each segment. Each word is composed of four fields:

- Bits 0 through 7 (the low-order byte) of each word specify the **segment number** (SEGNUM). This is the position the code segment will occupy in the **segment table** at execution time. In the 128K system the segment table is 64 entries long, hence valid numbers for the SEGNUM field are 0..63. In the 64K system the segment table is 32 entries long, hence valid numbers for the SEGNUM field are 0..31. Segment tables are described in Chapter 3.
- Bits 8 through 11 of the second byte in the SEGINFO word specify the **machine type** (MTYPE) of the code in the segment. The machine types are:

  0: Unidentified code, perhaps from another Compiler.

  1: P-code, most significant byte first.

  2: P-code, least significant byte first (a stream of packed ASCII characters fills the low byte of a word first, then the high byte). This kind of P-code is used by the Apple II family.

  3 through 9: Assembled native code, produced from assembly-language text. Machine type 7 identifies native code for the 6502 microprocessor.

- Bit 12 of the SEGINFO word is unused.
- Bits 13 through 15 of the SEGINFO word contain the version number of the system. The current version number is 6, indicating Apple II Pascal 1.3.

The SEGINFO array is the last of the five arrays that contain 16 elements, one element for each slot. The remainder of the segment dictionary contains information pertinent to the execution of the entire codefile.

In the 128K system, the INTRINS-SEGS field consists of four words (64 bits); in the 64K system it consists of 2 words (32 bits). These words specify which Intrinsic Units are needed to execute the codefile. Each Intrinsic Unit in a program library file, SYSTEM.LIBRARY, or a library file specified in a Library Name File is identified by a segment number (or two segment numbers if the Intrinsic Unit has both a code and data segment). Each one of the 64 bits in these words corresponds to one of the 64 possible intrinsic segment numbers. If the nth bit is set, the codefile needs the Intrinsic Unit whose segment number is n in order to execute. Bits corresponding to the segment numbers of unused Intrinsic Units are zeroed.

The FILLER array contains 68 unused words in the 128K system, 70 in the 64K system.

The COMMENT array contains text provided by a $C Compiler option or when the LIBRARY program is used. The $C Compiler option is described in Part II, Chapter 5.

Segment Numbers

At execution time, every segment has a segment number from 0 to 63 in the 128K system, or 0 to 31 in the 64K system. No two segments in the program can have the same number. Segment numbers are assigned as follows:

- The user program itself is segment 1.
- The segments used by the Pascal operating system are 0, and 2 through 6. 58 through 63 are reserved.

- The segment number of an Intrinsic Unit segment is determined by the unit's heading when the unit is compiled. These numbers can be found by using the LIBMAP utility program to examine the segment dictionary of the library to which the unit belongs.
- The segment numbers of Regular Unit segments and of SEGMENT procedures and functions within the program are automatically assigned by the system as the program is compiled and linked. They begin at 7 and ascend. Note that after a Regular Unit is linked with a program, it may not have the same segment number that was shown for it in the library's segment dictionary (when examined with the LIBMAP utility), because the Linker may reassign segment numbers of Regular Units.

Pascal's assignment of segment numbers is summarized Table 2-1.

Table 2-1. Segment Number Assignment

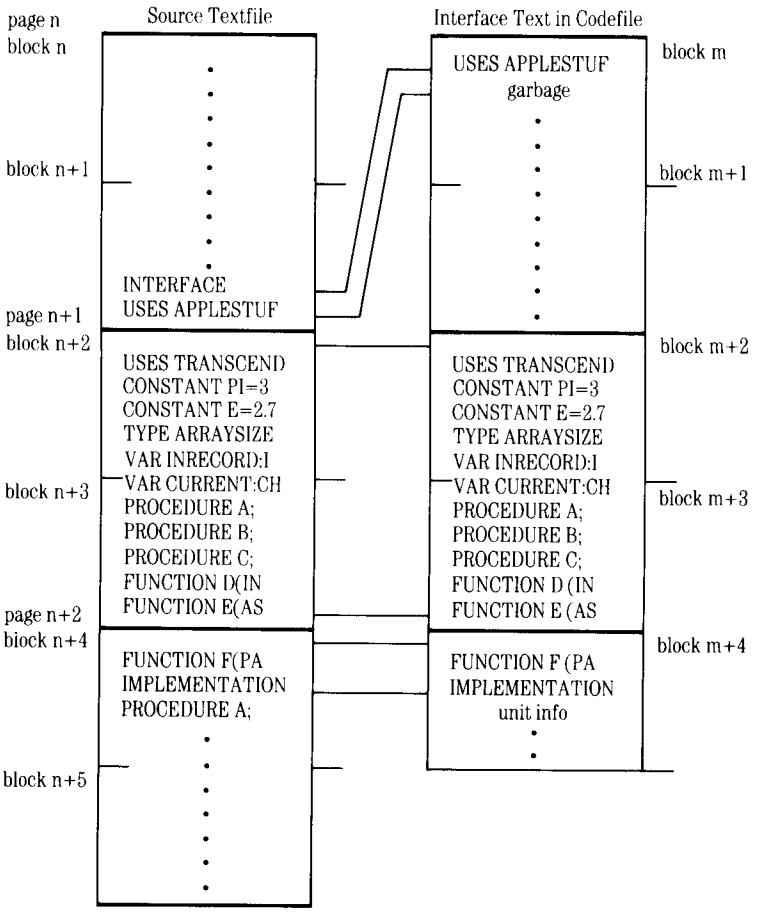
Segment Number	Assignment
0	Pascal operating system
1	User program
2...6	Pascal operating system
7...29	Units, SEGMENT procedures and functions
30	PASCALIO unit
31	LONGINTIO unit
32...57	Units, SEGMENT procedures and functions
58...63	Reserved

Normally, only when writing an Intrinsic Unit do you need to specify segment numbers. The factors that go into your choice of Intrinsic Unit segment numbers are set forth in Chapters 12 and 15 of Part III.

Interface Text

Code segments of Program Units may have interface text before their code part; host program segments, SEGMENT functions and procedures, and EXTERNAL procedures and functions never have interface text. The interface text contains the ASCII text of the INTERFACE section in the source text of a Program Unit. The construction of an interface text of a segment from its source text (by the Compiler) is shown in Figure 2-5.

Figure 2-5. Construction of Interface Text in a Codefile



The Pascal Compiler reads source text and produces interface text in two-block **pages** of 1024 bytes each. Interface text always starts on a page boundary and follows all of the conventions of a *textfile*, with the exception that the last page of the interface text may be either 1 or 2 blocks long. The interface text is identical to the source text, except for the first and last pages. The information in the first page of the source text is truncated, so that the first character in the output page is the character following the INTERFACE keyword in the original source text (*U* in Figure 2-5). This format may leave a considerable amount of unused space in the first page. The last page of the source text is truncated after the

IMPLEMENTATION keyword; it is possible that only one block of this page may be produced if the IMPLEMENTATION keyword occurs in the first block of the page. Valid data in each page of a textfile end with a CR (ASCII 13) followed by at least one NULL (ASCII 0).

The ten characters immediately following the IMPLEMENTATION keyword contain special unit information. All ten characters are ASCII spaces, except for an *E* in the ninth position required by the Pascal Compiler and LIBRARY programs to terminate the interface text. A *P* may occur, instead of a space, in the second of the ten character positions to signify to the Pascal Compiler that the unit requires the PASCALIO standard Program Unit. The fourth position will be occupied by an *L* if the unit requires the LONGINTIO standard Program Unit. These items—*IMPLEMENTATION*, *P*, *L*, and *E*—are all considered tokens by the Compiler; thus, their order is significant, but their spacing and case are not.

Interface text is not stripped of nonprinting characters or comments. Leaving the comments in the interface text produces more complete internal program documentation at the expense of increased codefile length.

> ***By the Way:*** The interface text of Program Unit segments is used only during compilation. Thus it can be removed from completed codefiles that will only be executed. The effect is a reduction in codefile size.

The TEXTADDR array of the segment dictionary contains pointers to the starting address of the interface text for each segment. The pointers specify block numbers, relative to the start of the codefile. The field is 0 for segments that are not Program Unit code segments, as well as Program Unit segments that do not have an interface part.

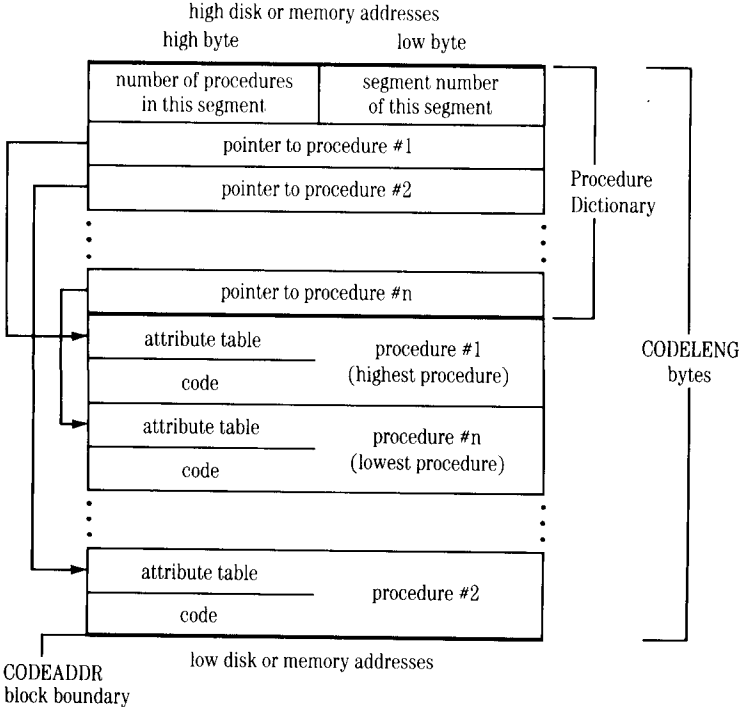
Code Parts

The code part of a code segment consists of

- Code for a group of up to 254 procedures;
- A **procedure dictionary**, containing information about the procedures.

Figure 2-6 is a diagram of the code part of a code segment. Each code part contains the code for the highest level procedure in the segment, as well as the code for each of the non-SEGMENT procedures and functions within the segment. The code of the highest level procedure, which is generated last, appears highest in the code part.

Figure 2-6. The Code Part of a Code Segment



Each procedure in a code part is assigned a **procedure number** starting at 1, for the highest level procedure or SEGMENT procedure, and ranging as high as 254. All references to a procedure are made via its segment number and procedure number. Translation from a procedure number to the location of the **procedure code** in the code segment is accomplished via the procedure dictionary.

Below the procedure dictionary is the code for the various procedures in the segment. The procedure dictionary grows downward toward lower disk addresses; the code part starts at the first byte of the block specified in the CODEADDR field of the segment dictionary and grows upward toward higher addresses.

Procedure Dictionaries

The position of the low-order byte of the highest word in a segment's procedure dictionary can be calculated as

CODEADDR * 512 + CODELENG − 2

This highest word in a procedure dictionary contains the segment number in its *low-order (even) byte*, and the number of procedures in the segment in its *high-order (odd) byte*. Below this word is a sequence of words that contain self-relative pointers to the top (high address) of the code of each procedure in the segment (shown in Figure 2-6).

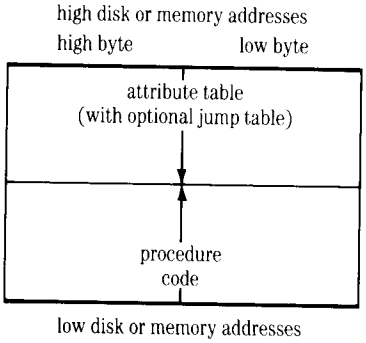
> *A Technical Note:* A **self-relative pointer** contains the absolute distance, in bytes, between the low-order byte of the pointer and the low-order byte of the word to which it points. To find the address referred to by a self-relative pointer, subtract the pointer from the address of its location.

A procedure's number is an index into the procedure dictionary: the nth word in the dictionary (counting downward from higher addresses) contains a pointer to the top (high address) of the code of procedure *n*. As 0 is not a valid procedure number, the 0th word of the dictionary is used to store the segment number of the code segment and the number of procedures in that code segment, as described above.

Procedures

Each procedure consists of two parts: the procedure code itself (in the lower portion of the procedure growing up toward higher addresses), and an **attribute table** of the procedure. Some procedures have a third part called the **jump table** located at the base of the attribute table. Figure 2-7 is a diagram of a typical procedure.

Figure 2-7. A Typical Procedure



Attribute Tables

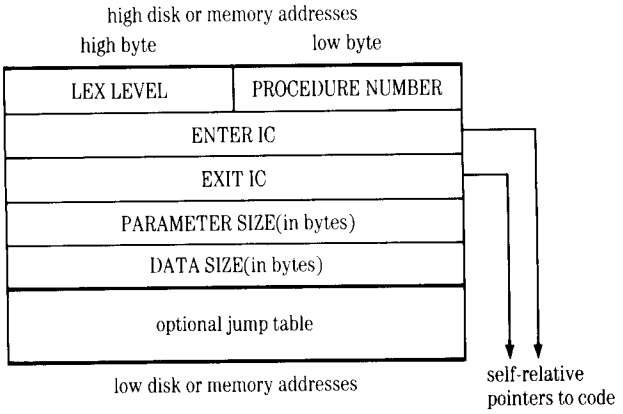
The **attribute table** of a procedure provides information needed to execute the procedure. Procedure attribute tables are pointed to by entries in the procedure dictionary of each segment.

The Compiler produces P-code by compiling source text, and the Assembler produces native code by assembling assembly-language. Procedures may contain solely P-code or native code, but not a mixture of both. It is possible to produce segments with procedures of both code types, by using the Linker. In this case the MTYPE field in the segment dictionary is set to the value for assembled native code (7), because the code for that segment is then machine-specific. The Interpreter is able to determine the type of code in a particular procedure via information contained in the procedure's attribute table. The format of the attribute table for an assembly-language procedure is very different from that for a P-code procedure. These two formats are described in the following sections.

P-Code Procedure Attribute Tables

The format of a **P-code procedure attribute table** is illustrated in Figure 2-8.

Figure 2-8. P-Code Procedure Attribute Table



The fields of a P-code procedure attribute table are

- PROCEDURE NUMBER: This field contains the **procedure number**. The procedure number field is the low-order (even) byte of the highest word in the attribute table.
- LEX LEVEL: This field specifies the depth of lexical nesting of the procedure. The **lexical level** of the Pascal operating system is −1, the lexical level of a user program is 0, that of the first nested procedure is 1, and so forth. The LEX LEVEL field is the high-order (odd) byte of the highest word in the attribute table.
- ENTER IC: This field contains a self-relative pointer (again, a positive number, pointing back) to the first P-code instruction to be executed in the procedure.
- EXIT IC: This field contains a self-relative pointer to the beginning of the sequence of P-code instructions that must be executed to terminate the procedure properly.

- PARAMETER SIZE: This field specifies the number of bytes of parameters passed to a procedure from its calling procedure. If the procedure is a *function*, this number includes the number of bytes to be reserved for the returned value.
- DATA SIZE: This field specifies the number of bytes to be reserved for local variables of the procedure.

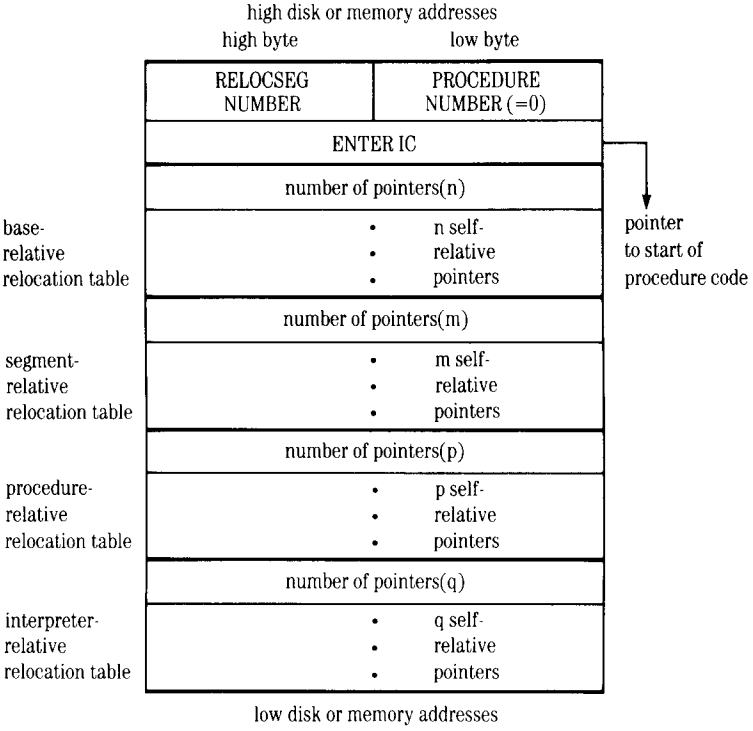
At the base of the attribute table there may be a section called the **jump table**. Jump tables are used by the P-machine to determine the locations specified by jump instructions. The entries are self-relative pointers to addresses within the procedure code. During execution, the **JTAB** pseudoregister points to the PROCEDURE NUMBER field of the attribute table of the currently executing procedure. See Chapter 3 for an explanation of pseudoregisters.

All jump instructions include a specified jump offset (n). In the case of short forward jumps, the jump table is ignored, and execution jumps by n bytes. In the case of backward or long forward jumps, the jump offset specifies a self-relative pointer in the jump table located n bytes below the location pointed to by the JTAB register. Execution jumps to the byte address pointed to by the self-relative pointer.

Assembly-Language Procedure Attribute Tables

The format of an attribute table of an assembly-language procedure is very different from that of a P-code procedure attribute table. It is illustrated in Figure 2-9.

Figure 2-9. An Assembly-Language Procedure Attribute Table



The highest word in the attribute table of an assembly-language procedure always has a 0 in its PROCEDURE NUMBER field. When the Interpreter encounters a 0 in the PROCEDURE NUMBER field as it loads the segment, it realizes it must "fix up" references in the procedure code according to information contained in the rest of the attribute table. The RELOCSEG NUMBER field contains either a 0 or a positive number (the significance of which is explained below in conjunction with base-relative relocation). In the case of Intrinsic Units without data segments, the number placed in this field is 1. The second highest word of the attribute table is, as in P-code procedure attribute tables, the ENTER IC field—a self-relative pointer to the first executable instruction of the procedure. Following this are four relocation tables used by the Interpreter. From high address to low address, they are base-relative, segment-relative, procedure-relative, and Interpreter-relative relocation tables.

Relocation Tables

A **relocation table** is a sequence of records that contain information necessary to relocate any relocatable addresses used by code within the procedure. Relocatable addresses are relocated whenever the segment containing the procedure is loaded into memory. Only native code procedures use relocatable addresses; procedures that contain P-code are completely position-independent, and no relocation list is needed.

The format of all four relocation tables is the same: the highest word of each table specifies the number of entries (possibly 0) that follow (at lower disk addresses) in the table. The remainder of each table comprises that number of one-word self-relative pointers to locations in the procedure code that must be "fixed." The locations are "fixed" when the segment is loaded by the addition of the appropriate relative relocation constant, which is known to the Interpreter.

Addresses pointed to by a **base-relative relocation table** are relocated relative to the address contained in the P-machine's **BASE** pseudoregister if the RELOCSEG NUMBER field of the procedure's attribute table is 0. The BASE register is a pointer to the **activation record** of the most recently invoked **base procedure** (lexical level 0 or −1). **Global** (lexical level 0) variables are accessed by indexing from the value of the BASE register. If the RELOCSEG NUMBER field is not 0, the relocations will be relative to the lowest address of the segment whose segment number is contained in the RELOCSEG NUMBER field. Base-relative relocation is used by assembly procedures that are linked with Intrinsic Units to access the Intrinsic Unit's data segment. .PUBLIC and .PRIVATE are the *Assembler directives* that generate base-relative relocation fields.

Addresses pointed to by a **segment-relative relocation table** are relocated relative to the lowest address of the segment. The value of the address of the lowest byte in the segment is added to each of the addresses pointed to in the relocation table. .REF and .DEF are the Assembler directives that generate segment-relative relocation fields.

Addresses pointed to by a **procedure-relative relocation table** are relocated relative to the lowest address of the procedure. The value of the address of the lowest byte in the procedure is added to each of the addresses pointed to in the relocation table.

The Interpreter-relative relocation fields point to relocatable addresses that access Pascal Interpreter procedures or variables. Addresses pointed to by an **Interpreter-relative relocation table** are relocated relative to a table in the Interpreter. See the explanation of the .INTERP directive in Part II, Chapter 6.

Linker Information

Following the code part of a segment there may be Linker information. Linker information is the portion of a code segment that enables the Linker to resolve references to variables, identifiers, and constants between separately compiled or assembled code. Segments produced by an Assembler always have Linker information. Segments produced by the Compiler have Linker information only if they are segments with EXTERNAL procedures or Program Units, or user programs that USE Regular Units.

The starting location of Linker information is not included in the segment dictionary (as was the case with the starting location of the interface text and code parts); it must be inferred. Linker information starts on the *block boundary* following the last block of code for a segment, and grows toward higher addresses. The block number of the first record of Linker information can be calculated as

CODEADDR + ((CODELENG + 511) DIV 512)

where CODEADDR and CODELENG are the values of fields in the segment dictionary.

Linker information is stored as a sequence of records—one record for each indentifier, constant, or variable that is referenced but not defined in the source, as well as records for items defined to be accessible from other procedures.

The following Pascal declaration describes one record that may appear within Linker information.

```
LITYPES = (EOFMARK, UNITREF, GLOBREF, PUBLREF, PRIVREF,
CONSTREF, GLOBDEF, PUBLDEF, CONSTDEF, EXTPROC, EXTFUNC,
SEPPROC, SEPFUNC, SEPPREF, SEPFREF); {Linker information types}

OPFORMAT = (WORD,BYTE,BIG);  {label size}

LCRANGE:  1..MAXLC;  {currently MAXINT (32767)}

PROCRANGE:  1..MAXPROC;  {currently 254}

LIENTRY = RECORD

   NAME: PACKED ARRAY[0..7] OF CHAR; {name of unit, proc, or variable symbol}

   CASE LITYPE:  LITYPES OF
      GLOBREF,    {reference to a global address}
      PUBLREF,    {reference to a host program variable}
      PRIVREF,    {reference to private variables in a host
                   activation record}
      CONSTREF,   {reference to a global constant}
      UNITREF,    {reference to a Regular Unit}
      SEPPREF,    {unused}
      SEPFREF:    {unused}
         (FORMAT:  OPFORMAT;
          NREFS:  INTEGER;
          NWORDS: LCRANGE;
          POINTERLIST: ARRAY [1..((NREFS-1) DIV 8)+1] OF
                       ARRAY [0..7] OF INTEGER); {segment-relative pointers}

      GLOBDEF:    {global address definition}
         (HOMEPROC: PROCRANGE;
          ICOFFSET: LCRANGE);

      PUBLDEF:    (BASEOFFSET: LCRANGE); {host program variable definition}

      CONSTDEF:   (CONSTVAL: INTEGER); {host program constant definition}

      EXTPROC,    {EXTERNAL procedure declaration}
      EXTFUNC,    {EXTERNAL function declaration}
      SEPPROC,    {separate assembly procedure}
      SEPFUNC:    {separate assembly function}
         (SRCPROC: PROCRANGE;
          NPARAMS: INTEGER);

      EOFMARK:    {end-of-file mark}
         (NEXTBASELC: LCRANGE;
          PRIVDATASEG: SEGNUMBER);

    END;
```

Linker Information Fields

The **Linker information types** GLOBREF, PUBLREF, PRIVREF, CONSTREF, and UNITREF, all have similar fields. The FORMAT field may be BIG, BYTE, or WORD, and specifies the format of the P-machine **operand** that refers to the entity given by the NAME array. See "Instruction Formats," in Chapter 4, for a description of these formats. The NREFS field specifies the number of references to this entity in the code segment; there will be an equivalent number of entries in the POINTERLIST array. The NWORDS field specifies the amount of space, in words, to be allocated for PRIVREF Linker information types; the NWORDS field is ignored for all other Linker information types.

The **POINTERLIST** array is a list of pointers into the code segment, each of which points to a location within the code segment where there is a reference to the entity specified by the NAME array. The locations are given as absolute byte addresses within the code segment. The POINTERLIST array is composed of records of eight words, but only the first ((NREFS − 1) MOD 8)+1 words of the last record are used. All unused words in each array are zeroed.

Global Address Linker Information Types

Separate assembly-language procedures and functions can share data structures and subroutines by means of the .DEF, .REF, .PROC, and .FUNC Assembler directives. These directives cause the Assembler to generate information that the Linker uses to resolve external references between *separate procedures* and *functions* in the same assembly or between procedures and functions assembled separately. Each entity referenced by a .REF Assembler directive results in a GLOBREF Linker information type entry that designates fields to be updated by the Linker. Each entity defined by a .DEF, .PROC, or .FUNC Assembler directive results in a GLOBDEF Linker information type entry that provides the Linker with the values to fix the .REF references.

The GLOBREF Linker information type is used to link addresses between assembled procedures. The FORMAT field is always WORD. The NREFS field specifies the number of pointers in the POINTERLIST array (each of which points to a different reference).

The GLOBDEF Linker information type defines the location of an entity in an assembled procedure. The HOMEPROC field contains the number of the procedure that defines the entity specified by the NAME array. The

ICOFFSET field specifies the location within the named procedure where the entity is defined. The location is given as a byte offset, relative to the start of the procedure. There is no POINTERLIST array associated with a GLOBDEF Linker information type.

As a program is linked, the Linker picks up each address defined explicitly by .DEF and implicitly by .PROC and .FUNC, and fixes up each reference to it in other procedures. The Linker must insert the final segment offset of the address in all words pointed to by the POINTERLIST array.

Host-Communication Linker Information Types

The Assembler directives .CONST, .PUBLIC, and .PRIVATE enable an assembly-language procedure or function to share addresses and data space with the host program that calls it. Data values and locations are referred to by name in both the host program and the called procedure or function. Each entity referenced by a .CONST, .PUBLIC, or .PRIVATE Assembler directive results in a CONSTREF, PUBLREF, or PRIVREF Linker information type entry, respectively, that designates fields to be fixed up by the Linker. Each entity defined by a CONSTANT or VARIABLE declaration results in a CONSTDEF or PUBLDEF Linker information type entry, respectively, that provides the Linker with the values to fix references. As a program is linked, the Linker picks up each entity defined by .CONST, .PUBLIC, and .PRIVATE, and fixes up each reference to it in other procedures. The Linker must insert the final segment offset of the address in all words pointed to by the POINTERLIST array.

The PUBLREF Linker information type is used to link global variables in the activation record of a host program to assembly-language procedures or Regular Units. The PUBLREF Linker information type results from a .PUBLIC directive in an assembly-language procedure or from use of variables declared in the INTERFACE of Regular Units. The NAME array specifies a variable that is referenced in the segment, and defined as a global variable in the host program. The FORMAT field is WORD for assembly-language procedures, and BIG for Regular Units. The NREFS field specifies the number of pointers in the POINTERLIST array (each of which points to a different reference). The Linker must add the offset of the referenced identifier to all words pointed to by the POINTERLIST array. Activation records are explained in Chapter 3.

The PUBLDEF Linker information type declares a global variable in the host program. A PUBLDEF Linker information type is generated for each global variable in the host program that appears in a VAR declaration. The BASEOFFSET field specifies the location of the variable specified by the

NAME array within the activation record of the host program that contains it. The location is given as a word offset, relative to the start of the **data area**. There is no POINTERLIST array associated with a PUBLDEF Linker information type.

The CONSTREF Linker information type is used to link constants in an assembled procedure to a global constant in the host program. The CONSTREF Linker information type results from a .CONST directive in an assembly-language procedure. The NAME array specifies a constant that is referenced in the segment, and defined as a global constant in the host program. The FORMAT field is WORD. The NREFS field specifies the number of pointers in the POINTERLIST array (each of which points to a different reference). The Linker must place the constant value into all locations pointed to by the POINTERLIST array.

The CONSTDEF Linker information type declares a global constant in the host program. A CONSTDEF Linker information type is generated for each global constant in the host program that appears in a CONSTANT declaration. The CONSTVAL field contains the value of the declared constant. There is no POINTERLIST array associated with a CONSTDEF Linker information type.

The PRIVREF Linker information type is used to indicate a reference to variables of an assembly-language procedure or Regular Unit, to be stored in the host program's global data area, and yet be inaccessible to the host program. The PRIVREF Linker information type results from either a .PRIVATE directive in assembly language, or by the use of global variables declared in the IMPLEMENTATION of Regular Units. The FORMAT field is always WORD. The NWORDS field specifies the amount of space, in words, to be allocated. The NREFS field specifies the number of pointers in the POINTERLIST array. The Linker must add the offset of the start of the allocated area within the global data area to all words pointed to by the POINTERLIST array.

The UNITREF Linker information type is used to link references between Regular Units. The NAME array specifies the name of a Regular Unit that is referenced within another Regular Unit. The FORMAT field is always BYTE. The NREFS field specifies the number of pointers in the POINTERLIST array (each of which points to a different reference). The Linker must insert the final segment number of the referenced unit in all locations pointed to by entries in the POINTERLIST array.

Procedure and Function Linker Information Types

Separate assembly-language procedures and functions are referenced via EXTERNAL declarations in the calling segment. The Linker information types EXTPROC, EXTFUNC, SEPPROC, and SEPFUNC, are used to link procedures and functions between segments. Each .PROC or .FUNC entity referenced by a PROCEDURE...EXTERNAL declaration results in an EXTPROC or EXTFUNC Linker information type entry, respectively, that designates fields to be fixed up by the Linker. All procedure or function code that begins with .PROC or .FUNC results in a SEPPROC or SEPFUNC Linker information type entry, respectively, that provides the Linker with the values to fix references. As each procedure or function is linked, the Linker picks up each procedure number and parameter size declared in the separate procedure or function, and transfers it to each external reference of that same procedure or function.

The SRCPROC field specifies the procedure number of the referenced or declared procedure. The NPARAMS field specifies the number of words of parameters indicated in the .PROC or .FUNC directive. There is no POINTERLIST array associated with EXTPROC, EXTFUNC, SEPPROC, or SEPFUNC Linker information types.

Miscellaneous Linker Information Types

The EOFMARK Linker information type indicates the end of Linker information records. Additionally, if the segment is of the host program, the NEXTBASELC field indicates the number of words in the host program's global data area. If the segment is an Intrinsic Unit code segment, the PRIVDATASEG field contains the segment number of the associated data segment.

Chapter 3 The P-Machine

The previous chapter discussed the static structure of program codefiles on disk and in memory. This chapter discusses the dynamic structure of program code as it is being executed in memory.

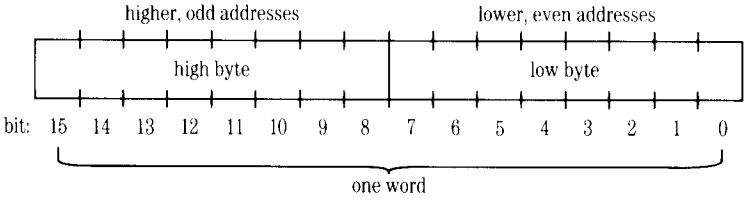
The Apple Pascal **pseudomachine** or **P-machine**, a version of the UCSD Pascal P-machine, is the software-generated *device* that executes P-code as its machine language. Every computer operating under a form of UCSD Pascal has been programmed to "look like" this common P-machine, or a related variant, from the viewpoint of a program being executed. The P-machine has an **evaluation stack**, several registers, and a **user memory**. The user memory contains the **program stack** and the **heap**. These memory structures are described in Chapter 1. They are discussed in detail below.

The P-machine supports

- Variable addressing, including strings, byte arrays, packed fields, and *dynamic variables*;
- Logical, integer, real, set, array, and string, top-of-stack arithmetic and comparisons;
- Multi-element structure comparisons;
- Branches;
- Procedure and function calls and returns, including overlayable procedures;
- Miscellaneous procedures used by system and user programs.

The P-machine uses 16-bit words, with two 8-bit bytes per word. Words consist of two bytes, of which the lower, even-address byte is least significant. See Figure 3-1. The least significant bit of a word is bit 0, the most significant is bit 15.

Figure 3-1. Relationship of Words and Bytes



The Evaluation Stack

In the Apple II family, the evaluation stack uses a portion of the 6502 hardware stack, starting at memory location $1FF and growing downward to location $100. It is used for passing parameters, for returning function values, and as an operand source for many P-machine instructions. When an instruction is said to *push* an item, that item is placed on the top of the evaluation stack (the evaluation stack grows downward). The evaluation stack is extended by loads and is reduced by stores and most arithmetic operations.

Registers

The Apple II P-machine uses 8 **pseudoregisters**, and the hardware stack pointer. All registers are pointers to *word-aligned* structures, except the IPC register, which is a pointer to *byte-aligned* structures. The pseudoregisters are the following:

- SP: **Evaluation Stack Pointer**. This register contains a pointer to the current *top* of the evaluation stack (one byte below the last byte in use). It is actually the Apple II hardware stack pointer.
- IPC: **Interpreter Program Counter**. This register contains the address of the next instruction to be executed in the currently executing procedure. It is located at address $58.
- SEG: **SEGment pointer**. This register points to the highest word of the procedure dictionary of the segment to which the currently executing procedure belongs. It is located at address $56.
- JTAB: **Jump TABle pointer**. This register contains a pointer to the highest word of the attribute table in the procedure code of the currently executing procedure. (Attribute tables are explained in Chapter 2.) It is located at address $54.
- MP: **Markstack Pointer**. This register contains a pointer to the MSSTAT field, in the **markstack** of the currently executing procedure. Local variables in the activation record of the current procedure are accessed by indexing off of the location pointed to by the MP register. (Markstacks are explained later in this chapter.) It is located at address $52.

- BASE: **BASE procedure pointer**. This register contains a pointer to the MSSTAT field of the activation record of the most recently invoked base procedure (lexical level 0 or 1). Global (lex level 0) variables are accessed by indexing off of the location pointed to by the BASE register. (Activation records are explained later in this chapter.) It is located at address $50.
- KP: **program stacK Pointer**. This register contains a pointer to the lowest byte of the lowest word actually in use on the program stack. The program stack starts in high addresses of user memory and grows downward toward the heap. KP is located at address $5C.
- NP: **New Pointer**. This register contains a pointer to the current top of the heap (one byte above the last byte in use). The heap starts in low addresses of user memory and grows upward toward the program stack. It contains all dynamic variables. The heap is extended by the standard Pascal procedure NEW, and is cut back by the standard procedure RELEASE. NP is located at address $5A.
- STRP: **STRing Pointer**. This register exists in the 128K Pascal system only. It is a pointer to the first element of the linked list of strings and packed character arrays on the stack. Whenever the P-machine executes an LPA or LSA instruction (see Chapter 4), and the literal packed array or string constant contained in the instruction is not already on the program stack, the P-machine pushes it onto the program stack and links it to the list pointed to by this pseudoregister. STRP is located at address $5E.

The Program Stack and the Heap

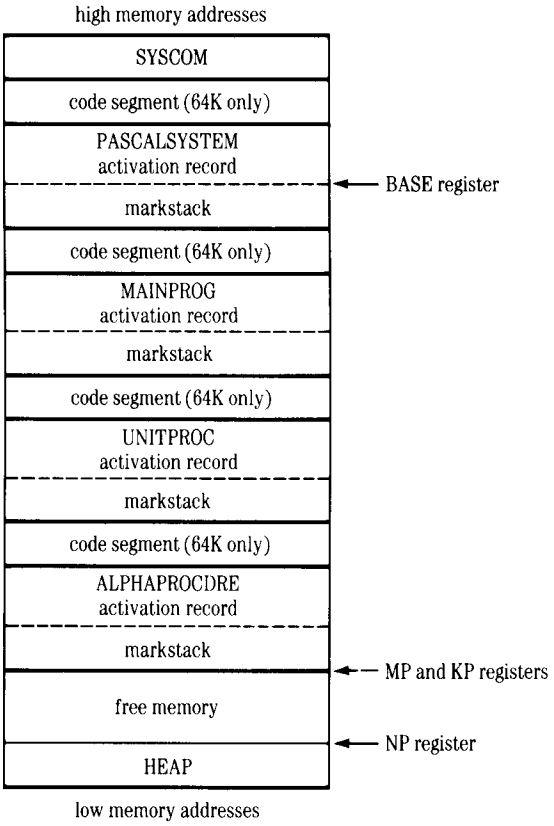
The operating system uses two dynamic structures called the program stack and the heap to store memory-resident code and data of an executing program. The program stack is used to store four kinds of items:

- In the 64K system only, a code segment for each active program segment and for each active Program Unit.
- In the 128K system only, assembly-language procedures and functions for each active program segment and for each active Program Unit.
- In both systems, an activation record containing local variables and markstack parameters for each procedure activation.
- In both systems, a data segment for each active Intrinsic Unit that requires one.

The heap is used to store dynamic variables.

Figure 3-2 is a diagram of the Apple Pascal program stack and heap with four active procedures.

Figure 3-2. The Program Stack and Heap With Four Active Procedures



SYSCOM

The operating system and the P-machine exchange information via the system communications area (also called **SYSCOM**) at the bottom (high addresses) of the stack. SYSCOM is accessible to both assembly-language procedures in the Interpreter and system procedures coded in Pascal (as if it

were part of the Pascal system global data). SYSCOM serves as an important communication link between these two levels of the system. These are the
fields in SYSCOM relevant to communication between the operating system and the P-machine:

- IORSLT: This field contains the error code returned by the last activated or terminated I/O operation (see Appendix 2H for a list of I/O Error messages).
- XEQERR: This field contains the error code of the last execution error (see Appendix 2H for a list of execution error messages).
- BOMBP: This field contains a pointer to the activation record of the procedure that caused the execution error.
- BOMBIPC: This field contains the IPC value when an execution error occurs.
- SYSUNIT: This field contains the Pascal volume number of the device from which the operating system was started up (usually the startup disk drive, volume #4).
- GDIRP: This field contains a pointer to the most recent disk directory read in, unless dynamic allocation or deallocation has taken place since then (see the MRK, RLS, and NEW instructions in Chapter 4). Disk directories are read into a temporary buffer directly above the heap.
- Segment Table: The segment table is a record that contains information needed by the P-machine to read code segments into memory or to allocate space for data segments.

The Segment Table

Every code segment has a name, but when a given segment references another during the execution of a program, it refers not to the segment's name, but to the segment's number. The Interpreter uses the segment number as an index into the segment table, which contains an entry for each segment in the program. See Figure 3-3. The segment table entries are indexed by segment number; each entry contains information needed to load the segment from the codefile on disk into memory. The segment table is a dynamic structure of SYSCOM, but is somewhat analagous to a segment dictionary, in that it is used to locate segments on disk.

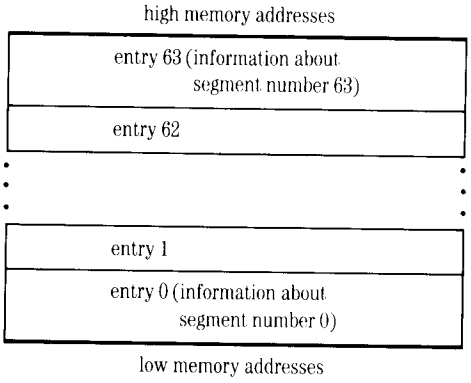
The segment table is located in the higher addresses of the SYSCOM area, at the bottom of the program stack. It contains entries for

- The segments of the Pascal operating system itself (numbers 0, 2..6);
- Each segment in the segment dictionary of the program codefile;
- Each Intrinsic Unit code and data segment needed by the host program.

No two segments in an executing program can have the same number because the numbers are used to index the segment table. The segment table has space for up to 64 entries in the 128K system, 32 in the 64K system. Because the system uses some segments, this means that 52 entries (26 in the 64K system) are left for the program to use.

> ***Remember:*** A program codefile contains 16 or fewer segments; any excess over 16 must be in either a program library, SYSTEM.LIBRARY, or library files specified in a Library Name File.

Figure 3-3. The Segment Table



Activation Records

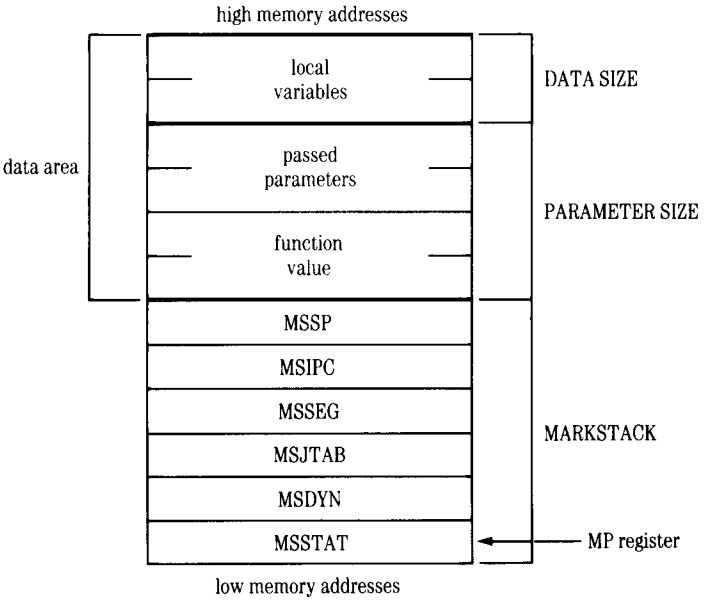
When a procedure is called, the code segment containing that procedure code is loaded by the Interpreter if it is not already present in memory. An activation record for the procedure is built on top of the program stack each time the procedure is called. See Figure 3-4. Only code segments require

activation records; data segments do not. The activation record for a procedure consists of

- The markstack, which contains addressing context information (static links), and information on the calling procedure's environment;
- Space for storing the value returned by the procedure, if the procedure is a function;
- Space for parameters passed to the procedure when it is called;
- Space for the local variables of the procedure.

Caution: When writing *recursive* procedures or functions, remember that each incarnation creates an activation record. These activation records can build up on the stack, causing a stack overflow. For further information on recursion, see Part III, Chapter 8.

Figure 3-4. An Activation Record



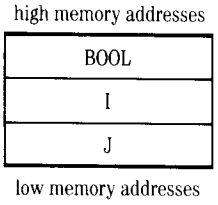
Space is allocated in the higher addresses of the activation record for variables local to the procedure. This variable space is allocated in the reverse order that variables are declared. Variables of the same type, whose declarations are separated by commas, are allocated space in forward order.

For example, the declarations

```
VAR I,J: INTEGER;
BOOL: BOOLEAN;
```

will cause space in the activation record to be allocated as shown in Figure 3-5.

Figure 3-5. The Order of Local Variable Allocation in an Activation Record



Space for parameter passing is allocated below the local variable space. If the procedure is a function, space is also reserved (below the parameter space) for storing the value returned by the function. A description of the format of variables in activation records is given in Chapter 4. The order of passed parameters is discussed in Part III, Chapter 9.

Local variables in the activation record of an active procedure are accessed by indexing from the location pointed to by the MP register. Global variables in the activation record of an active procedure are accessed by indexing from the location pointed to by the BASE register.

When a procedure is terminated, its activation record is removed from the stack.

Markstacks

The lower portion of the activation record is called a **markstack**. When a procedure call is made, the current values of the system pseudoregisters that characterize the operating environment of the calling procedure are stored in the markstack of the called procedure. Thus the system registers can be restored to precall conditions when control is returned to the calling procedure.

A procedure call causes the operating environment that existed in the system registers just at the time of the procedure call to be stored in the fields of the called procedure's markstack in the following manner:

System Registers		Markstack Fields
SP	→	MSSP (MarkStack evaluation Stack Pointer)
IPC	→	MSIPC (MarkStack Interpreter Program Counter)
SEG	→	MSSEG (MarkStack SEGment pointer)
JTAB	→	MSJTAB (MarkStack Jump TABle pointer)
MP	→	MSDYN (MarkStack DYNamic link)
STRP	→	MSSTRP (MarkStack STRing Pointer—128K system only)

The MSDYN field of a markstack contains a pointer to the MSSTAT field in the markstack of the procedure that called the new procedure. The combined MSDYN fields of all markstacks form a **dynamic chain** of links that describe the "route" by which the new procedure was called.

The MSSTAT field of a markstack contains a pointer to the MSSTAT field in the most recent markstack of the procedure that is the lexical parent of the called procedure. The Interpreter "knows" which procedure is the lexical parent, by looking up the **static chain** until it encounters a procedure whose lexical level is one less than the lexical level of the current procedure. The combined MSSTAT fields of a group of markstacks form a static chain of links that describe the lexical nesting of the called procedure.

The NP register is not stored because it does not change during a procedure call. The BASE register is not stored on the markstack because its value is related only to base procedure calls.

After building the new procedure's activation record on the program stack, new values for the IPC, SEG, JTAB, MP, and STRP registers are established. The registers are updated as follows:

- The IPC register points to the first instruction of the called procedure.
- The SEG register points to the procedure dictionary of the code segment that contains the called procedure.
- The JTAB register points to the attribute table of the called procedure.
- The MP register points to the markstack of the called procedure.
- The STRP register is initialized to NIL (zero).

After the registers are updated, the following takes place:

- If the called procedure has a lexical level of −1 or 0, the contents of the BASE register are saved on the evaluation stack, and the BASE register is set to the value of the MP register.
- Finally, KP is saved on top of the stack and a new value for KP is calculated.

These elements are not part of the markstack or activation record.

Each time a procedure is called, another activation record is added to the program stack. Once again the register values and the appropriate *static link* and *dynamic link* are stored in the new markstack, and the system registers are then updated. Note that the SEG register always points to the procedure dictionary of the segment that *contains* the procedure, and not the segment that called the procedure.

Once the code for a procedure has been loaded into memory, each further invocation of the same procedure causes only an activation record to be added to the program stack. The code is not loaded again.

When a return from a procedure occurs, the information in the markstack fields is transferred to the system registers, and the activation record of the inactive procedure is removed from the stack.

Additional information on procedure calls, and the relation of attribute tables to activation records, can be found in the section "Procedure and Function Calls" in Chapter 4.

Chapter 4 The P-Machine Instruction Set

Instruction Formats

Instructions for the P-machine consist of one or two bytes, followed by 0 to 4 parameters. Most parameters specify one word of information. There are five basic types of parameters:

UB: **Unsigned Byte**. Represents a nonnegative integer less than 256. The high-order byte of the parameter is implicitly zero.

SB: **Signed Byte**. Represents an integer from −128 to 127, in two's-complement form. The high-order byte is a sign extension of bit 7 of the low order byte.

DB: **Don't-care Byte**. Represents a nonnegative integer less than 128; thus it can be treated as SB or UB.

B: **Big**. This parameter is one byte long when used to represent values in the range 0 through 127, and is two bytes long when used to represent values in the range 128 through 32767. If the value represented is in the range 0 through 127, the high-order byte of the parameter is implicitly zero. If the value represented is in the range 128 through 32767, bit 7 of the first byte is cleared and the first byte is used as the high order byte of the parameter. The second byte is used as the low-order byte.

W: **Word**. A two-byte parameter, low byte first. Represents values in the range −32768 through 32767.

Any exceptions to these formats are noted below, in the descriptions of the individual instructions.

Operand Formats

Although an element of a structure in memory may be as small as one bit (as in a packed array of boolean), variables to be operated on by the P-machine are always unpacked into full words. All top-of-stack (**tos**) operations expect their operands to occupy at least one word on the evaluation stack.

Formats of Variables on the Stack

Variables are stored in activation records and on the evaluation stack in the manner described below.

Boolean

One word. Bit 0 indicates the value (0=FALSE, 1=TRUE), and this is the only information used by boolean comparisons. However, the boolean operators LAND, LOR, and LNOT operate on all 16 bits, in a bitwise manner.

Integer

One word, two's complement notation, capable of representing values in the range −32768..32767.

Long Integer

3..11 words. A variable declared as INTEGER[n] is allocated ((n+3) DIV 4) + 2 words of memory space. Regardless of the value of a long integer, its actual size remains the same as its allocated size. Each decimal digit of a long integer is stored as four bits of binary-coded decimal. The format of long integers on the stack is as follows:

word 0 (tos):	contains the allocated length, in words.
word 1 (tos−1):	low byte contains the sign (all zeros = positive, all ones = negative); high byte not used.
word 2 (tos−2):	four least significant decimal digits. The low byte contains the two more significant decimal digits (BCD). The high byte contains the two less significant digits.
word n (tos−n):	four most significant decimal digits. The low byte contains the two more significant decimal digits (BCD). The high byte contains the two less significant digits.

The format of long integers in activation records is as follows: word 0 is not stored; word 1 is the lowest word in memory; word n is the highest word in memory.

Scalar (User-Defined)

One word, in range 0..32767.

Char

One word, with the low byte containing a character. The internal character set is *extended* ASCII, with 0..127 representing the standard ASCII set, and 128..255 representing user-defined characters.

Real

Two words, whose format is diagrammed in Part III of this manual, Appendix 3C. In general, the format for 32-bit real numbers is as follows:

Bit	Item	Contained In
0..15	fraction	tos
15..22	fraction	
23..30	exponent	tos − 1
31	sign	

Pointer

One or three words, depending on the type of pointer. Pascal pointers (internal word pointers) consist of one word that contains a word address (the address of the low byte of the word). Internal byte pointers consist of one word that contains a byte address. Internal packed field pointers consist of three words:

word 0 (tos):	right bit number of field
word 1 (tos − 1):	field width (in bits)
word 2 (tos − 2):	word pointer to the word that contains the field

Set

0..31 words in an activation record, 1..32 words on the evaluation stack. Sets are implemented as bit vectors, always with a lower index of zero. A set variable declared as SET OF m..n is allocated (n+15) DIV 16 words of memory space. All words allocated in an activation record for a set contain valid information (the set's actual size agrees with its allocated size).

A set on the evaluation stack is represented by a word (tos) specifying the length of the set, followed by that number of words of information. The set may be padded with extra words (to compare it with another set of different size, say), and the length word changed to indicate the number of words in

the structure when padded. Before storing it back in an activation record, you must force a set back to the size allocated to it, by issuing an ADJ instruction.

Records and Arrays

Any number of words. Arrays are stored in forward order, with higher-indexed array elements appearing in higher-numbered memory locations. Only the address of the record or array is loaded onto the evaluation stack, never the structure itself. Packed arrays must have an integral number of elements in each word, as there is no packing across word boundaries (it is acceptable to have unused bits in each word). The first element in each word has bit 0 as its low-order bit.

Strings

1..128 words. Strings are a flexible version of packed arrays of CHAR. A STRING[n] declaration occupies (n DIV 2)+1 words of memory space. Byte 0 of a string is the current length of the string, and bytes 1..length(string) contain valid characters.

Formats of Constants in P-Code

Constant scalars, sets, and strings may be embedded in the instruction stream, in which case they have special formats.

- All scalars (excluding reals) greater than 127 are represented by two bytes, high byte first.
- All string literals occupy length(literal)+1 bytes of memory space, and are word-aligned. The first byte is the length, the rest are the actual characters. This format applies even if the literal should be interpreted as a packed array of characters.
- All reals, sets, and long integers are word-aligned and in REVERSE word order, that is, the higher-order bits of the real or set are in lower-numbered memory locations.

Conventions and Notation

Each operand on the evaluation stack (for example, tos or tos − 1), can contain from one byte to 256 bytes, depending on its type and value. Unless specifically noted to the contrary, operands used by an instruction are popped off the evaluation stack (removed from the stack and not returned) as they are used.

In the descriptions of the various P-machine instructions the parameters are given as UB , SB , DB , B , or W . The term *tos* means the operand on the top of the evaluation stack, *tos* − *1* is the next operand, and so on. The columns of information in the various instruction descriptions have the following meaning:

Column 1	Column 2	Column 3	Column 4
opcode mnemonic	decimal opcode	instruction parameters	full name and operation of the instruction

P-Machine Instructions

This section lists all the P-machine opcodes by their class of operation.

One-Word Loads and Stores

This section lists opcodes that load and store single words.

Constant

SLDC_0	0		**Short load one-word constant.** For an instruction SLDC_x , push the opcode, x , with the high byte zero. That is, push an integer with the value x.
SLDC_1	1		
: :	:		
SLDC_127	127		
LDCN	159		**Load constant NIL.** Push 0.
LDCI	199	W	**Load one-word constant.** Push W.

Local

SLDL_1	216		**Short load local word.** For an instruction SLDL_x, fetch the word with offset x in the data area of the executing procedure's activation record and push it.
SLDL_2	217		
: :	:		
SLDL_16	231		
LDL	202	B	**Load local word.** Fetch the word with offset B in the data area of the executing procedure's activation record and push it.
LLA	198	B	**Load local address.** Fetch the address of the word with offset B in the data area in the executing procedure's activation record and push it.
STL	204	B	**Store local word.** Store tos into word with offset B in the data area of the executing procedure's activation record.

Global

SLDO_1	232		**Short load global word.** For an instruction SLDO_x, fetch the word with offset x in the data area of the activation record of the base procedure and push it.
SLDO_2	233		
: :	:		
SLDO_16	247		
LDO	169	B	**Load global word.** Fetch the word with offset B in the data area of the activation record of the base procedure and push it.
LAO	165	B	**Load global address.** Fetch the address of the word with offset B in the data area of the activation record of the base procedure and push it.
SRO	171	B	**Store global word.** Store tos into the word with offset B in the data area of the activation record of the base procedure.

Intermediate

LOD	182	DB,B	**Load intermediate word.** Fetch the word with offset B in the activation record found by traversing DB links in the static chain, and push it.
LDA	178	DB,B	**Load intermediate address.** Fetch address of the word with offset B in the activation record found by traversing DB links in the static chain, and push it.

STR	184	DB,B	**Store intermediate word.** Store tos into the word with offset B in the activation record found by traversing DB links in the static chain.

Indirect

SIND_0	248		**Load indirect word.** Fetch the word pointed to by tos and push it (this is a special case of SIND_x, described below).
SIND_1	249		**Short index and load word.** For an instruction SIND_x, index the word pointer tos by x words, and push the word pointed to by the result.
SIND_2	250		
: :	:		
SIND_7	255		
IND	163	B	**Static index and load word.** Index the word pointer tos by B words, and push the word pointed to by the result.
STO	154		**Store indirect word.** Store tos into the word pointed to by tos − 1.

Extended

LDE	157	UB,B	**Load extended word.** Fetch the word with offset B in the data segment number UB (of an Intrinsic Unit) and push it.
LAE	167	UB,B	**Load extended address.** Fetch the address of the word with offset B in the data segment number UB (of an Intrinsic Unit), and push it.
STE	209	UB,B	**Store extended word.** Store tos into the word with offset B in the data segment number UB (of an Intrinsic Unit).

Multiple-Word Loads and Stores (Sets and Reals)

LDC	179	UB,<data>	**Load multiple-word constant.** Fetch the word-aligned <data> of UB words in reverse word order, and push the data.
LDM	188	UB	**Load multiple words.** Fetch UB words of word-aligned data in reverse order, whose beginning is pointed to by tos, and push the block.
STM	189	UB	**Store multiple words.** Transfer UB words of word-aligned data in reverse order, whose beginning is pointed to by tos, to the location block pointed to by tos − 1.

Byte Array Handling

LDB 190

Load byte. Index the byte pointer tos − 1 by the integer index tos , and push the byte (after zeroing high byte) pointed to by the resulting byte pointer.

STB 191

Store byte. Index the byte pointer tos − 2 by the integer index tos − 1 , and push the byte tos into the location pointed to by the resulting byte pointer.

String Handling

LSA 166 UB,<chars>

Load constant string address. (*64K system*): Push a byte pointer to the location containing UB, then skip IPC past <chars>. (*128K system*): Push a word pointer to the constant character string UB,<chars> onto the evaluation stack. As the constant string is contained in the code segment, not in the stack/heap space, a copy of the string is pushed onto the program stack. If this string has not previously been pushed onto the stack during the currently active procedure, copy UB<chars> onto the program stack (add one space to the end of the string if UB<chars> is an even number of characters); push a 16-bit integer onto the program stack that points to the first byte of the string in the procedure code; push a 16-bit linkage pointer onto the program stack that points to the string or packed array most recently pushed onto the program stack (the linkage pointer is 0 if no other string or packed array has yet been pushed onto the stack); push a pointer onto the evaluation stack that points to the string length byte UB on the program stack. If UB<chars> has been pushed onto the stack during the currently active procedure, push a pointer onto the evaluation stack that points to the string length byte UB on the program stack. The contents of the program stack are not changed. In either case, advance the IPC register past the original copy of the string in the code space.

SAS	170	UB	**String assign.** tos is either a source byte pointer or a character. (Characters always have a high byte of zero, while pointers never do.) tos−1 is a destination byte pointer. UB is the declared size of the destination string. If the declared size is less than the current size of the source string, give an execution error; otherwise transfer all bytes of source containing valid information to the destination string.
IXS	155		**Index string array.** tos−1 is a byte pointer to a string. tos is an index into the string. Check to see that the index is in the range 1..current string length. If so, continue execution; if not, give an execution error.

Record and Array Handling

MOV	168	B	**Move words.** Transfer a source block of B words, pointed to by byte pointer tos , to a similar destination block pointed to by byte pointer tos−1 .
INC	162	B	**Increment field pointer.** Index the word pointer tos by B words and push the resultant word pointer.
IXA	164	B	**Index array.** tos is an integer index, tos−1 is the array base word pointer, and B is the size (in words) of an array element. Compute a word pointer (tos−1) + (B * tos) to the indexed element and push the pointer.
IXP	192	UB1,UB2	**Index packed array.** tos is an integer index, tos−1 is the array base word pointer. UB1 is the number of elements per word, and UB2 is the field width (in bits). Compute a packed field pointer to the indexed field and push the resulting pointer.

LPA	208	UB,<chars>	**Load a packed array.** (*64K system*): Push a byte pointer to the first location following the one that contains UB, and then skip IPC past <chars>. (*128K system*): Push a word pointer to the packed array <chars> onto the evaluation stack. As the packed array is contained in the code segment, not in the stack/heap space, a copy of the array is pushed onto the program stack. If this array has not previously been pushed onto the stack during the currently active procedure, copy <chars> onto the program stack (add one space to the end of the array if <chars> has an odd number of characters); push a 16-bit integer onto the program stack that points to the first byte of the array in the procedure code; push a 16-bit linkage pointer onto the program stack that points to the string or packed array most recently pushed onto the program stack (the linkage pointer is 0 if no other string or packed array has yet been pushed onto the stack); push a pointer onto the evaluation stack that points to the first byte of the packed array on the program stack. If the same packed array has been pushed onto the stack during the currently active procedure, push a pointer onto the evaluation stack that points to the first byte of the array on the program stack. The contents of the program stack are not changed. In either case, advance the IPC register past the original copy of the array in the code space.
LDP	186		**Load a packed field.** Fetch the field indicated by the packed field pointer tos, and push it.
STP	187		**Store into a packed field.** Store the data tos into the field indicated by the packed field pointer tos−1.

Dynamic Variable Allocation

Note that the NP register points to the current top of the heap (one byte beyond the last byte in use). GDIRP is a SYSCOM field that points to the top of a temporary directory buffer above the heap.

NEW	158 1		**New variable allocation.** tos is the size (in words) to allocate for the variable, and tos−1 is a word pointer to a pointer variable. If the GDIRP field is non-NIL, set GDIRP to NIL. Store the NP register into the word pointed to by tos−1, and increment the NP register by tos words.
MRK	158 31		**Mark heap.** Set the GDIRP field to NIL, then store the NP register into the word indicated by the word pointer tos.
RLS	158 32		**Release heap.** Set the GDIRP field to NIL, then store the word indicated by the word pointer tos into the NP register.

Top-of-Stack Arithmetic

These operations perform arithmetic on values at the top of the stack.

Integers

Note: Overflows do not cause an execution error; they are ignored and the results are undefined.

ABI	128	**Absolute value of integer.** Push the absolute value of the integer tos . The result is undefined if tos is initially −32768.
ADI	130	**Add integers.** Add tos and tos−1 , and push the resulting sum.
NGI	145	**Negate integer.** Push the two's complement of tos . The result is undefined if tos is intially −32768.
SBI	149	**Subtract integers.** Subtract tos from tos−1 , and push the resulting difference.
MPI	143	**Multiply integers.** Multiply tos and tos−1 , and push the resulting product.
SQI	152	**Square integer.** Square tos , and push the result.
DVI	134	**Divide integers.** Divide tos−1 by tos and push the resulting integer quotient (any remainder is discarded). Division by zero causes an execution error.
MODI	142	**Modulo integers.** Divide tos−1 by tos and push the resulting remainder.
CHK	136	**Check against subrange bounds.** Insure that tos−1 <= tos−2 <= tos , leaving tos−2 on the stack. If conditions are not satisfied, give an execution error.
EQUI	195	tos−1 = tos .
NEQI	203	tos−1 <> tos .
LEQI	200	tos−1 <= tos .
LESI	201	tos−1 < tos .
GEQI	196	tos−1 >= tos .
GRTI	197	tos−1 > tos .
		Integer comparisons. Compare tos−1 to tos and push the result, TRUE or FALSE.

Noninteger Comparisons

The next six instructions are nonspecific noninteger comparisons. Comparisons using specific values of UB are given in later sections.

EQU	175	UB	tos − 1 = tos .
NEQ	183	UB	tos − 1 <> tos .
LEQ	180	UB	tos − 1 <= tos .
LES	181	UB	tos − 1 < tos .
GEQ	176	UB	tos − 1 >= tos .
GRT	177	UB	tos − 1 > tos .

Compare tos − 1 to tos , and push the result, TRUE or FALSE. The type of comparison is specified by UB :

Contents of tos − 1 & tos	Value of UB for Comparison
reals	2
strings	4
booleans	6
sets	8
byte arrays	10
words	12

Reals

FLT	138	**Float top-of-stack.** Convert the integer tos to a floating-point number, and push the result.
FLO	137	**Float next to top-of-stack.** tos is a real, tos − 1 is an integer. Convert tos − 1 to a real number, and push the result.
TNC	158 22	**Truncate real.** Truncate (as defined by Jensen and Wirth) the real tos and convert it to an integer, and push the result.
RND	158 23	**Round real.** Round (as defined by Jensen and Wirth) the real tos , then truncate and convert to an integer, and finally push the result.
ABR	129	**Absolute value of real.** Push the absolute value of the real tos .
ADR	131	**Add reals.** Add tos and tos − 1, and push the resulting sum.
NGR	146	**Negate real.** Negate the real tos , and push the result.
SBR	150	**Subtract reals.** Subtract tos from tos − 1 and push the resulting difference.

MPR	144	**Multiply reals.** Multiply tos and tos−1 and push the resulting product.
SQR	153	**Square real.** Square tos , and push the result.
DVR	135	**Divide reals.** Divide tos−1 by tos , and push the resulting quotient.
POT	158 35	**Power of ten.** If the integer tos is in the range 0 <= tos <= 38, push the real value 10 ^ tos . If the integer tos is not in this range, give an execution error.
EQUREAL	175 2	tos−1 = tos .
NEQREAL	183 2	tos−1 <> tos .
LEQREAL	180 2	tos−1 <= tos .
LESREAL	181 2	tos−1 < tos .
GEQREAL	176 2	tos−1 >= tos .
GTRREAL	177 2	tos−1 > tos . **Real comparisons.** Compare the real tos−1 to the real tos , and push the result, TRUE or FALSE.

Strings

EQUSTR	175 4	tos−1 = tos .
NEQSTR	183 4	tos−1 <> tos .
LEQSTR	180 4	tos−1 <= tos .
LESSTR	181 4	tos−1 < tos .
GEQSTR	176 4	tos−1 >= tos .
GRTSTR	177 4	tos−1 > tos . **String comparisons.** Find the string pointed to by word pointer tos−1 , compare it alphabetically to the string pointed to by word pointer tos , and push the result, TRUE or FALSE.

Logical

LAND	132	**Logical AND.** Push the result of tos−1 AND tos . This is a bitwise AND of two 16-bit words.
LOR	141	**Logical OR.** Push the result of tos−1 OR tos . This is a bitwise OR of two 16-bit words.

LNOT	147		**Logical NOT.** Push the one's complement of tos . This is a bitwise negation of one 16-bit word.
EQUBOOL	175 6		tos−1 = tos .
NEQBOOL	183 6		tos−1 <> tos .
LEQBOOL	180 6		tos−1 <= tos .
LESBOOL	181 6		tos−1 < tos .
GEQBOOL	176 6		tos−1 >= tos .
GRTBOOL	177 6		tos−1 > tos . **Boolean comparisons.** Compare bit 0 of tos−1 to bit 0 of tos and push the result, TRUE or FALSE.

Sets

ADJ	160	UB	**Adjust set.** Force the set tos to occupy UB words, either by expansion (putting zeros "between" tos and tos−1) or by compression (chopping high words off the set), discard the length word, and push the resulting set.
SGS	151		**Build a one-member set.** If the integer tos is in the range 0 <= tos <= 511 , push the set [tos] . If not, give an execution error.
SRS	148		**Build a subrange set.** If the integer tos is in the range 0 <= tos <= 511 , and the integer tos−1 is in the same range, push the set [tos−1..tos] (push the set [ ] if tos−1 > tos). If either integer exceeds the range, give an execution error.
INN	139		**Set membership.** If integer tos−1 is in set tos , push TRUE. If not, push FALSE.
UNI	156		**Set union.** Push the union of sets tos and tos−1 . (tos OR tos−1)
INT	140		**Set intersection.** Push the intersection of sets tos and tos−1 . (tos AND tos−1)
DIF	133		**Set difference.** Push the difference of sets tos−1 and tos . (tos−1 AND NOT tos).
EQUPOWR	175 8		tos−1 = tos .
NEQPOWR	183 8		tos−1 <> tos .
LEQPOWR	180 8		tos−1 <= (is a subset of) tos .
GEQPOWR	176 8		tos−1 >= (is a superset of) tos . **Set comparisons.** Compare set tos−1 to the set tos , and push the result, TRUE or FALSE.

Byte Arrays

EQUBYT	175 10	B	tos − 1 = tos .
NEQBYT	183 10	B	tos − 1 <> tos .
LEQBYT	180 10	B	tos − 1 <= tos .
LESBYT	181 10	B	tos − 1 < tos .
GEQBYT	176 10	B	tos − 1 >= tos .
GRTBYT	177 10	B	tos − 1 > tos .

Byte array comparisons. Compare byte array tos − 1 to byte array tos and push the result, TRUE or FALSE. Note: <=, <, >=, and > must be used with packed arrays of characters only. B specifies the number of bytes to compare.

Records and Word Array Comparisons

EQUWORD	175 12	B	tos − 1 = tos .
NEQWORD	183 12	B	tos − 1 <> tos .

Word or multiword structure comparisons. Compare word structure tos − 1 to word structure tos , and push the result, TRUE or FALSE. B gives the number of bytes to compare.

Jumps

The JTAB register points to the highest word of the attribute table in the currently executing procedure. The IPC register points to the next instruction to be executed in the currently activating procedure.

UJP	185	SB	**Unconditional jump.** SB is a jump offset. If this offset is nonnegative (a jump less than 128 bytes forward), it is simply added to the IPC register. (A value of zero for the jump offset will make any jump a two-byte NOP.) If SB is negative (a jump backward or more than 127 bytes forward), then SB is used as a byte offset into the jump table within the attribute table pointed to by the JTAB register, and the IPC register is set to the byte address (JTAB[SB]) − contents of (JTAB[SB]) .
FJP	161	SB	**False jump.** Jump (as described for UJP) if tos is FALSE.

XJP 172 W1,W2,<case table>,W3

Case jump. W1 is word-aligned and the minimum case selector of the case table. W2 is the maximum case selector. W3 is an unconditional jump offset past the case table. The case table is (W2 − W1 + 1) words long, and contains self-relative pointers.

If tos , the case selector expression, is not in the range W1..W2 , then point the IPC register at W3 . Otherwise, use (tos − W1) as an index into the case table, and set the IPC register to the byte address (casetable[tos − W1]) minus the contents of (casetable[tos − W1]), and continue execution.

Procedure and Function Calls

Here is the general method of procedure/function invocation:

1. Find the procedure code of the called procedure.
2. From the DATA SIZE and PARAMETER SIZE fields of the attribute table of the called procedure, determine the size (in bytes) of the needed activation record, and extend the program stack by that number of bytes.
3. Copy the number of bytes specified by the PARAMETER SIZE field from the top of the evaluation stack (tos) to the beginning of the space just allocated on the program stack. This passes parameters to the new procedure from its calling procedure.
4. Build a markstack, saving the SP, IPC, SEG, JTAB, STRP, MP, and a static link pointer (MSSTAT) to the most recent activation record of the procedure that is the lexical parent of the called procedure.
5. Calculate new values for the SP, IPC, JTAB, and MP registers; if necessary, calculate a new value for the SEG register. Issue an execution error if the program stack overflows.
6. If the called procedure has a lexical level of −1 or 0 (in other words, if it is a base procedure) save the value of the BASE register on the evaluation stack and then equate the BASE register with the MP register.
7. Save the value of the KP register on the program stack.
8. Calculate a new value for the KP register.

CLP 206 UB

Call local procedure. Call procedure number UB , which is an immediate child of the currently executing procedure and in the same segment. The MSSTAT field (static link) of the markstack is set to the value of the old MP register.

CGP	207	UB	**Call global procedure**. Call procedure number UB , which is at lexical level 1 and in the same segment as the currently executing procedure. The MSSTAT field (static link) of the markstack is set to the value of the BASE register.
CIP	174	UB	**Call intermediate procedure.** Call procedure number UB in the same segment as the currently executing procedure. The MSSTAT field (static link) of the markstack is set by looking up the dynamic chain (MSDYN fields) until an activation record is found whose caller had a lexical level one less than the procedure being called. Use that activation record's MSSTAT field (static link) as the static link of the new markstack.
CBP	194	UB	**Call base procedure.** Call procedure number UB , which is at lexical level −1 or 0. The MSSTAT field (static link) of the markstack is set to the MSSTAT field in the activation record of the procedure pointed to by the BASE register. The value of the BASE register is saved on the evaluation stack, after which it is set to point to the MSSTAT field of the activation record just created.
CXP	205	UB1,UB2	**Call external procedure.** Call procedure number UB2 , in segment UB1 . Used to call any procedure not in the same segment as the calling procedure, including base procedures. If the desired segment is not already in memory, it is read from disk. Build an activation record. Calculate the static link for the markstack (if the called procedure has a lex level of −1 or 0, set as in the CBP instruction; otherwise set as in the CIP instruction).
CSP	158	UB	**Call standard procedure.** Used to call standard procedures built into the P-machine.
RNP	173	DB	**Return from nonbase procedure.** DB is the number of words that should be returned as a function value (0 for procedures, 1 for nonreal functions, and 2 for real functions). Copy DB words from the higher addresses of the current procedure's activation record, and push them onto the evaluation stack. Then copy the information in the current procedure's markstack fields into the pseudoregisters to restore the calling procedure's correct environment.
RBP	193	DB	**Return from base procedure.** Move the value of the BASE register saved on the evaluation stack by a CBP, back into the BASE register, and then proceed as in the RNP instruction.

EXIT	158 4	**Exit from procedure.** tos is the procedure number, tos−1 is the segment number. First, set the MSIPC field to point to the exit code of the currently executing procedure. If the current procedure is not the one to exit from, change the MSIPC field of each markstack to point to the exit code of the procedure that invoked it, until the desired procedure is found. Then continue execution. If at any time the saved MSIPC field of the main body of the operating system is about to be changed, give an execution error.

System Support Procedures

FLC	158 10	**Fillchar.** tos is the source character. tos−1 is the number of bytes in the source character that are to be filled. tos−2 is a byte pointer to the first byte to be filled in the destination. Copy the character tos into tos−1 characters of the destination.
SCN	158 11	**Scan.** tos is a two-byte quantity (usually the default integer 0) that is pushed, but later discarded without being used in this implementation. tos−1 is a byte pointer to the first character to be scanned. tos−2 is the character against which each scanned character of the array is to be checked. tos−3 is 0 if the check is for equality, or 1 if the check is for inequality. tos−4 specifies the maximum number of characters to be scanned (scan to the left if negative). If a character check yields TRUE, push the number of characters scanned (negative, if scanning to the left). If tos−4 characters are scanned before character check yields TRUE, push tos−4.
MVL	158 02	**Moveleft.** tos specifies the number of bytes to move. tos−1 is a byte pointer to the first destination byte. tos−2 is a byte pointer to the first source byte. Copy tos bytes from the source to the destination, proceeding from left to right through both source and destination.
MVR	158 03	**Moveright.** tos specifies the number of bytes to move. tos−1 is a byte pointer to the first destination byte. tos−2 is a byte pointer to the first source byte. Copy tos bytes from the source to the destination, proceeding from right to left through both source and destination.

Miscellaneous

TIM	158 09		**Time.** Pop two pointers to two integers, and place zero in both integers.
XIT	214		**Exit the operating system.** Do a cold start of the system, as the operating system's Quit command.
BPT	213	B	**Breakpoint.** Not used (acts as a NOP).
NOP	215		**No operation.** Sometimes used to reserve space in the code for later additions.

Numerical Listing of Opcodes

For your convenience in finding a given P-code instruction, here they are in the numerical order of their opcodes.

Table 4-1. P-Codes in Numerical Order

Decimal Opcode	Mnemonic	Full Name	Location in Main Listing
0	SLDC_0	Short-load one-word constant	One-Word Loads and Stores
1	SLDC_1	Short-load one-word constant	One-Word Loads and Stores
:::			
127	SLDC_127	Short-load one-word constant	One-Word Loads and Stores
128	ABI	Absolute value of integer	Top-of-Stack Arithmetic
129	ABR	Absolute value of real	Top-of-Stack Arithmetic
130	ADI	Add integers	Top-of-Stack Arithmetic
131	ADR	Add reals	Top-of-Stack Arithmetic
132	LAND	Logical AND	Top-of-Stack Arithmetic
133	DIF	Set difference	Top-of-Stack Arithmetic
134	DVI	Divide integers	Top-of-Stack Arithmetic
135	DVR	Divide reals	Top-of-Stack Arithmetic
136	CHK	Range check	Top-of-Stack Arithmetic
137	FLO	Float TOS − 1	Top-of-Stack Arithmetic
138	FLT	Float TOS	Top-of-Stack Arithmetic
139	INN	Set membership	Top-of-Stack Arithmetic
140	INT	Set intersection	Top-of-Stack Arithmetic
141	LOR	Logical OR	Top-of-Stack Arithmetic
142	MODI	Modulo integers	Top-of-Stack Arithmetic
143	MPI	Multiply integers	Top-of-Stack Arithmetic

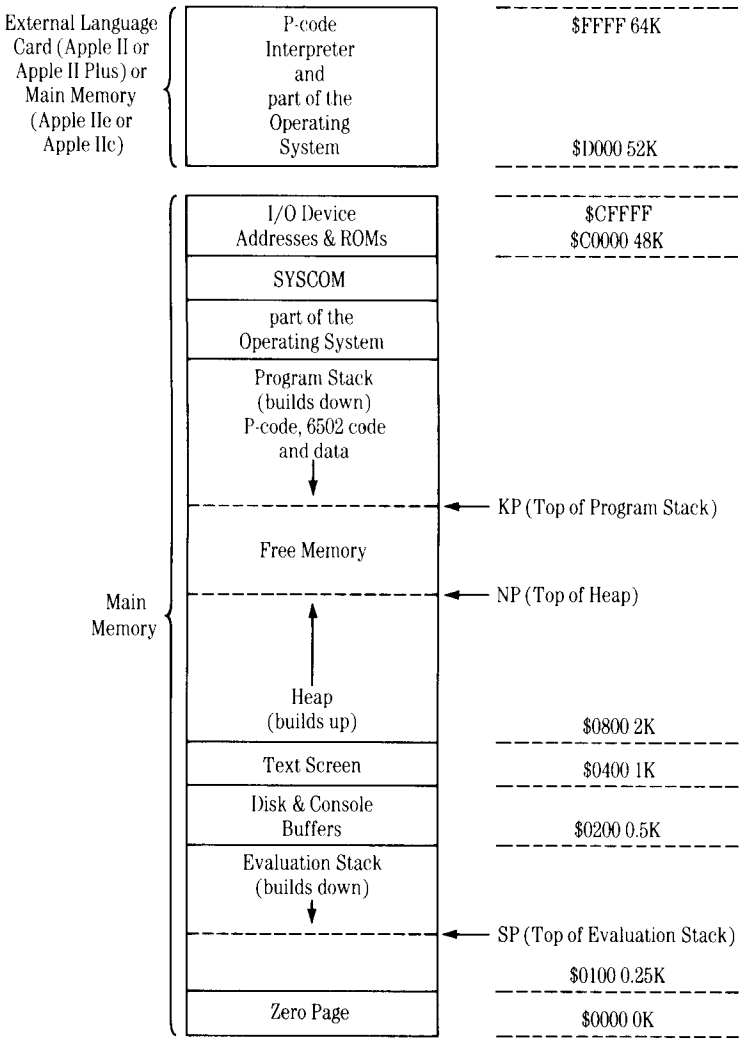
Decimal Opcode	Mnemonic	Full Name	Location in Main Listing
144	MPR	Multiply reals	Top-of-Stack Arithmetic
145	NGI	Negate integer	Top-of-Stack Arithmetic
146	NGR	Negate real	Top-of-Stack Arithmetic
147	LNOT	Logical NOT	Top-of-Stack Arithmetic
148	SRS	Build a subrange set	Top-of-Stack Arithmetic
149	SBI	Subtract integers	Top-of-Stack Arithmetic
150	SBR	Subtract reals	Top-of-Stack Arithmetic
151	SGS	Build a one-member set	Top-of-Stack Arithmetic
152	SQI	Square integer	Top-of-Stack Arithmetic
153	SQR	Square real	Top-of-Stack Arithmetic
154	STO	Store indirect word	One-Word Loads and Stores
155	IXS	Index string array	String Handling
156	UNI	Set union	Top-of-Stack Arithmetic
157	LDE	Load extended word	One-Word Loads and Stores
158	CSP	Call standard procedure	Procedure and Function Calls
158 1	NEW	New variable allocation	Dynamic Variable Allocation
158 2	MVL	Moveleft	System Support Procedures
158 3	MVR	Moveright	System Support Procedures
158 4	EXIT	Exit from procedure	Procedure and Function Calls
158 9	TIM	Time	Miscellaneous
158 10	FLC	Fillchar	System Support Procedures
158 11	SCN	Scan	System Support Procedures
158 22	TNC	Truncate real	Top-of-Stack Arithmetic
158 23	RND	Round real	Top-of-Stack Arithmetic
158 31	MRK	Mark heap	Dynamic Variable Allocation
158 32	RLS	Release heap	Dynamic Variable Allocation
158 35	POT	Power-of-ten	Top-of-Stack Arithmetic
159	LDCN	Load constant NIL	One-Word Loads and Stores
160	ADJ	Adjust set	Top-of-Stack Arithmetic
161	FJP	False jump	Jumps
162	INC	Increment field pointer	Record and Array Handling
163	IND	Static index and load word	One-Word Loads and Stores
164	IXA	Index array	Record and Array Handling
165	LAO	Load global address	One-Word Loads and Stores
166	LSA	Load constant string address	String Handling
167	LAE	Load extended address	One-Word Loads and Stores
168	MOV	Move words	Record and Array Handling
169	LDO	Load global word	One-Word Loads and Stores
170	SAS	String assign	String Handling
171	SRO	Store global word	One-Word Loads and Stores
172	XJP	Case jump	Jumps

Decimal Opcode	Mnemonic	Full Name	Location in Main Listing
173	RNP	Return from nonbase procedure	Procedure and Function Calls
174	CIP	Call intermediate procedure	Procedure and Function Calls
175	EQU	Equal	Top-of-Stack Arithmetic
175 2	EQUREAL	Real comparison	Top-of-Stack Arithmetic
175 4	EQUSTR	String comparison	Top-of-Stack Arithmetic
175 6	EQUBOOL	Boolean comparison	Top-of-Stack Arithmetic
175 8	EQUPOWR	Set comparison	Top-of-Stack Arithmetic
175 10	EQUBYT	Byte array comparison	Top-of-Stack Arithmetic
175 12	EQUWORD	Word or multiword structure comparison	Record and Word Array Comparisons
176	GEQ	Greater than or equal	Top-of-Stack Arithmetic
176 2	GEQREAL	Real comparison	Top-of-Stack Arithmetic
176 4	GEQSTR	String comparison	Top-of-Stack Arithmetic
176 6	GEQBOOL	Boolean comparison	Top-of-Stack Arithmetic
176 8	GEQPOWR	Set comparison	Top-of-Stack Arithmetic
176 10	GEQBYT	Byte array comparison	Top-of-Stack Arithmetic
177	GRT	Greater than	Top-of-Stack Arithmetic
177 2	GTRREAL	Real comparison	Top-of-Stack Arithmetic
177 4	GRTSTR	String comparison	Top-of-Stack Arithmetic
177 6	GRTBOOL	Boolean comparison	Top-of-Stack Arithmetic
177 10	GRTBYT	Byte array comparison	Top-of-Stack Arithmetic
178	LDA	Load intermediate address	One-Word Loads and Stores
179	LDC	Load multiple-word constant	Multiple-Word Loads and Stores
180	LEQ	Less than or equal	Top-of-Stack Arithmetic
180 2	LEQREAL	Real comparison	Top-of-Stack Arithmetic
180 4	LEQSTR	String comparison	Top-of-Stack Arithmetic
180 6	LEQBOOL	Boolean comparison	Top-of-Stack Arithmetic
180 8	LEQPOWR	Set comparison	Top-of-Stack Arithmetic
180 10	LEQBYT	Byte array comparison	Top-of-Stack Arithmetic
181	LES	Less than	Top-of-Stack Arithmetic
181 2	LESREAL	Real comparison	Top-of-Stack Arithmetic
181 4	LESSTR	String comparison	Top-of-Stack Arithmetic
181 6	LESBOOL	Boolean comparison	Top-of-Stack Arithmetic
181 10	LESBYT	Byte array comparison	Top-of-Stack Arithmetic
182	LOD	Load intermediate word	One-Word Loads and Stores
183	NEQ	Not equal	Top-of-Stack Arithmetic
183 2	NEQREAL	Real comparison	Top-of-Stack Arithmetic
183 4	NEQSTR	String comparison	Top-of-Stack Arithmetic
183 6	NEQBOOL	Boolean comparison	Top-of-Stack Arithmetic
183 8	NEQPOWR	Set comparison	Top-of-Stack Arithmetic
183 10	NEQBYT	Byte array comparison	Top-of-Stack Arithmetic
183 12	NEQWORD	Word or multiword structure comparison	Record and Word Array Comparisons
184	STR	Store intermediate word	One-Word Loads and Stores
185	UJP	Unconditional jump	Jumps
186	LDP	Load a packed field	Record and Array Handling

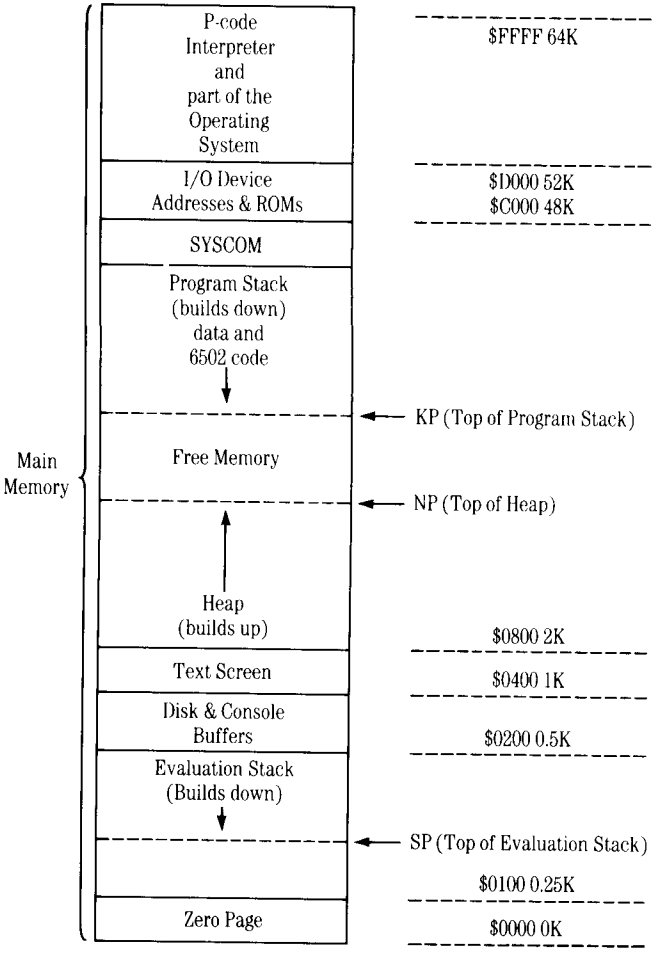
Decimal Opcode	Mnemonic	Full Name	Location in Main Listing
187	STP	Store into a packed field	Record and Array Handling
188	LDM	Load multiple words	Multiple-Word Loads and Stores
189	STM	Store multiple words	Multiple-Word Loads and Stores
190	LDB	Load byte	Byte Array Handling
191	STB	Store byte	Byte Array Handling
192	IXP	Index packed array	Record and Array Handling
193	RBP	Return from base procedure	Procedure and Function Calls
194	CBP	Call base procedure	Procedure and Function Calls
195	EQUI	Equals integer	Top-of-Stack Arithmetic
196	GEQI	Greater than or equal integer	Top-of-Stack Arithmetic
197	GRTI	Greater than integer	Top-of-Stack Arithmetic
198	LLA	Load local address	One-Word Loads and Stores
199	LDCI	Load one-word constant	One-Word Loads and Stores
200	LEQI	Less than or equal integer	Top-of-Stack Arithmetic
201	LESI	Less than integer	Top-of-Stack Arithmetic
202	LDL	Load local word	One-Word Loads and Stores
203	NEQI	Not equal integer	Top-of-Stack Arithmetic
204	STL	Store local word	One-Word Loads and Stores
205	CXP	Call external procedure	Procedure and Function Calls
206	CLP	Call local procedure	Procedure and Function Calls
207	CGP	Call global procedure	Procedure and Function Calls
208	LPA	Load a packed array	Record and Array Handling
209	STE	Store extended word	One-Word Loads and Stores
213	BPT	Breakpoint	Miscellaneous
214	XIT	Exit the operating system	Miscellaneous
215	NOP	No operation	Miscellaneous
216	SLDL_1	Short load local word	One-Word Loads and Stores
217	SLDL_2	Short load local word	One-Word Loads and Stores
:::			
231	SLDL_16	Short load local word	One-Word Loads and Stores
232	SLDO_1	Short load global word	One-Word Loads and Stores
233	SLDO_2	Short load global word	One-Word Loads and Stores
:::			
247	SLDO_16	Short load global word	One-Word Loads and Stores
248	SIND_0	Load indirect word	One-Word Loads and Stores
249	SIND_1	Short index and load word	One-Word Loads and Stores
250	SIND_2	Short index and load word	One-Word Loads and Stores
:::			
255	SIND_7	Short index and load word	One-Word Loads and Stores

Appendix 4A Memory Maps

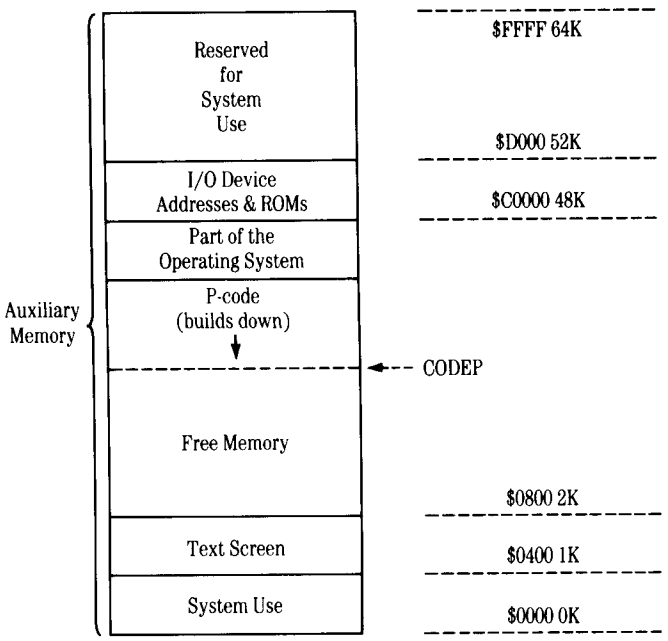
64K System Memory



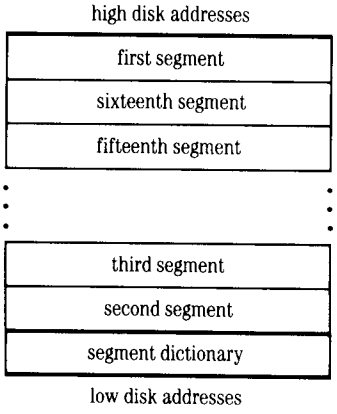
128K System Main Memory



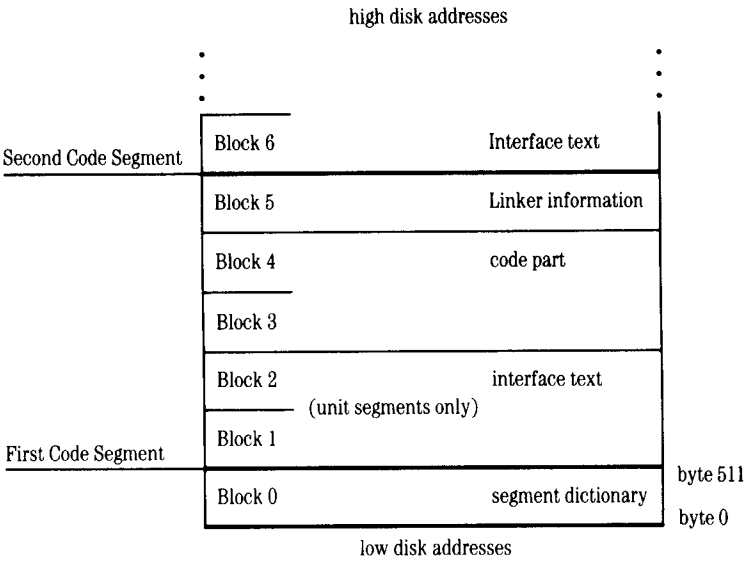
128K System Auxiliary Memory



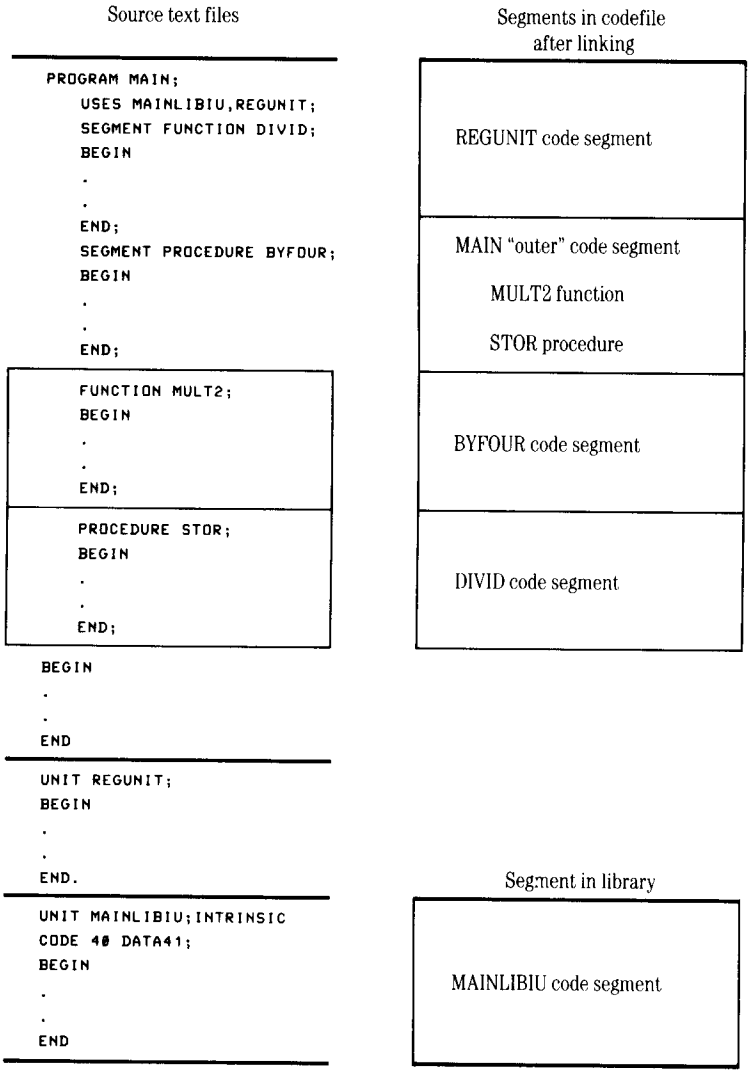
Code Segments in a Codefile



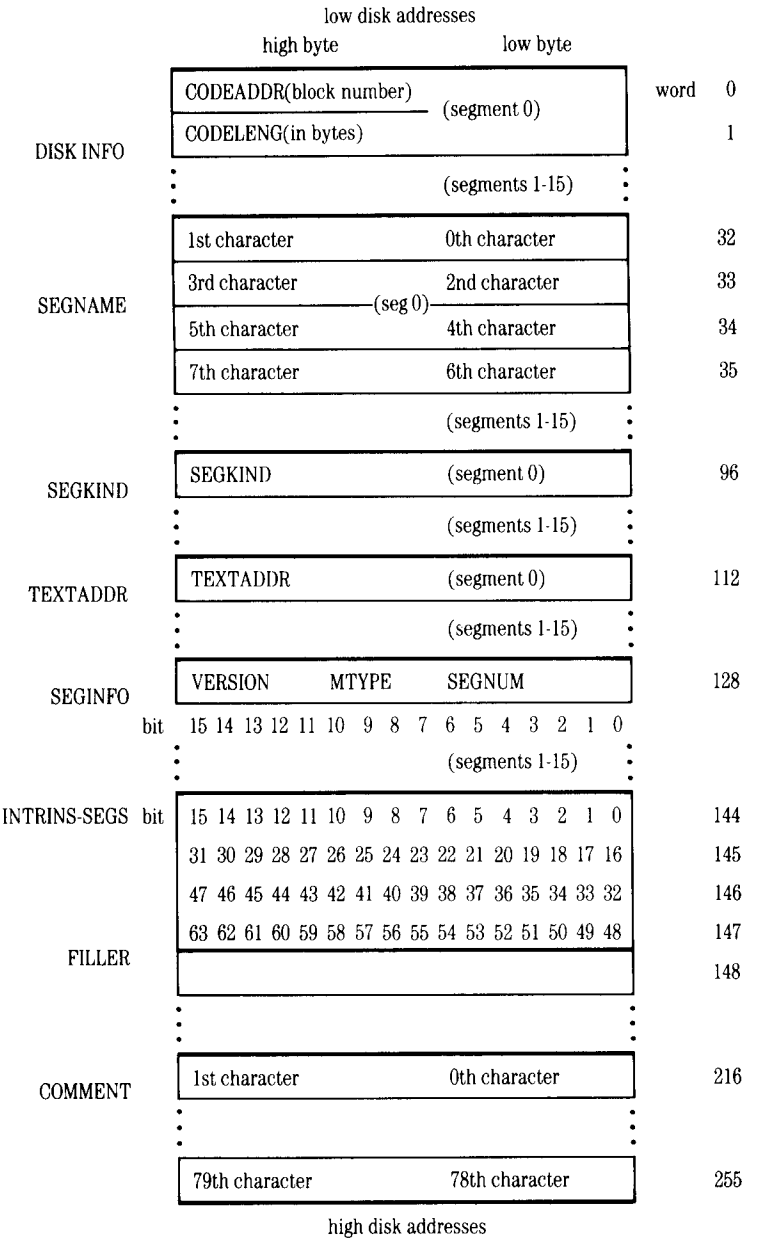
Blocks in a Code Segment

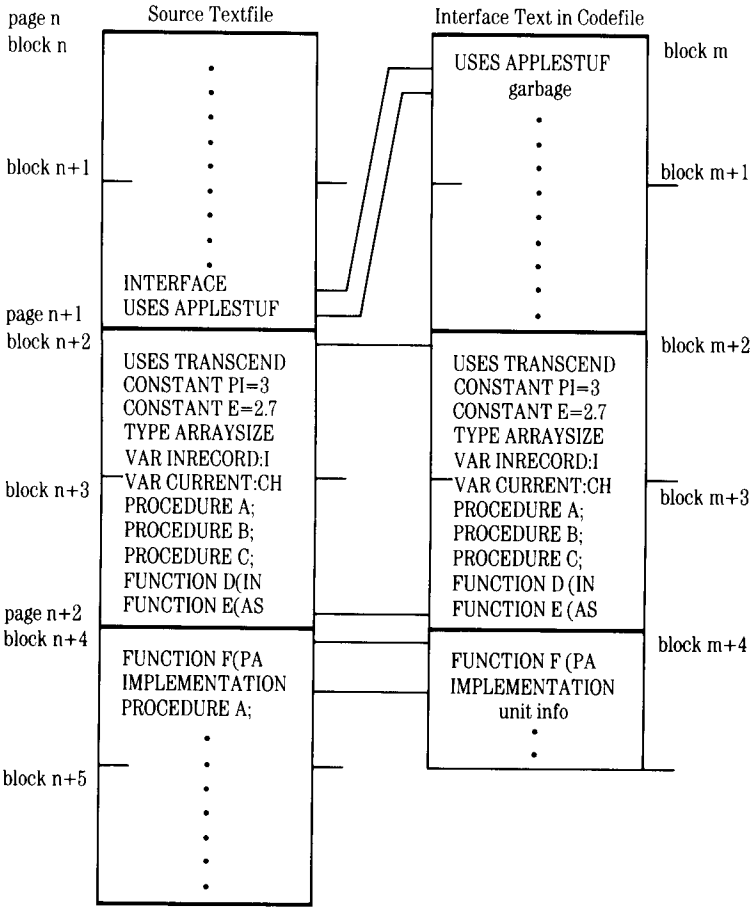


Correlation Between Programs and Codefiles

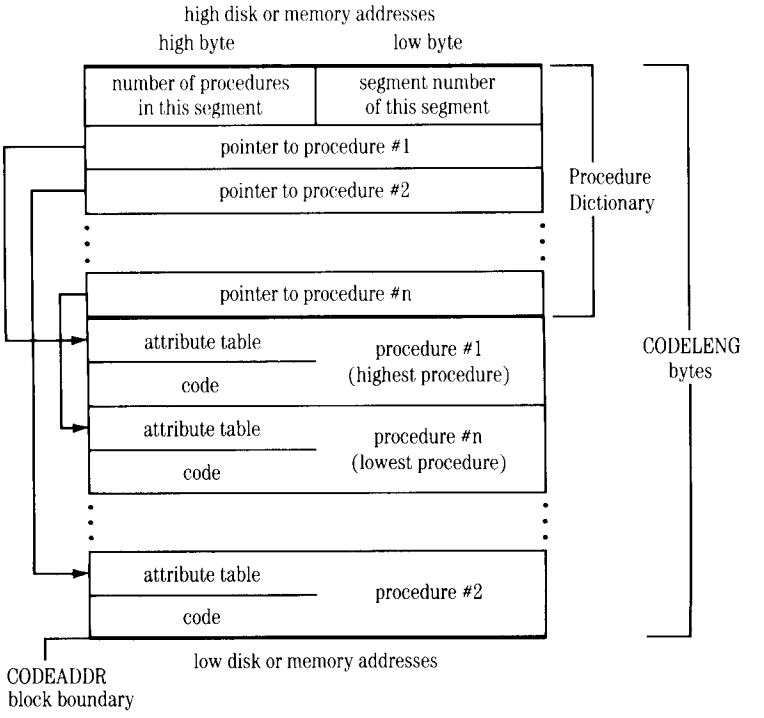


Segment Dictionary

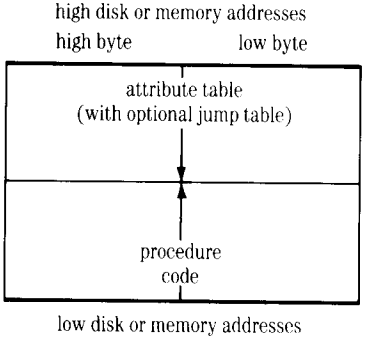



page n
block n
block n+1
page n+1
block n+2
block n+3
page n+2
block n+4
block n+5
Source Textfile
INTERFACE
USES APPLESTUF
USES TRANSCEND
CONSTANT PI=3
CONSTANT E=2.7
TYPE ARRAYSIZE
VAR INRECORD:I
VAR CURRENT:CH
PROCEDURE A;
PROCEDURE B;
PROCEDURE C;
FUNCTION D(IN
FUNCTION E(AS
FUNCTION F(PA
IMPLEMENTATION
PROCEDURE A;
Interface Text in Codefile
USES APPLESTUF
garbage
block m
block m+1
block m+2
block m+3
block m+4
USES TRANSCEND
CONSTANT PI=3
CONSTANT E=2.7
TYPE ARRAYSIZE
VAR INRECORD:I
VAR CURRENT:CH
PROCEDURE A;
PROCEDURE B;
PROCEDURE C;
FUNCTION D (IN
FUNCTION E (AS
FUNCTION F (PA
IMPLEMENTATION
unit info

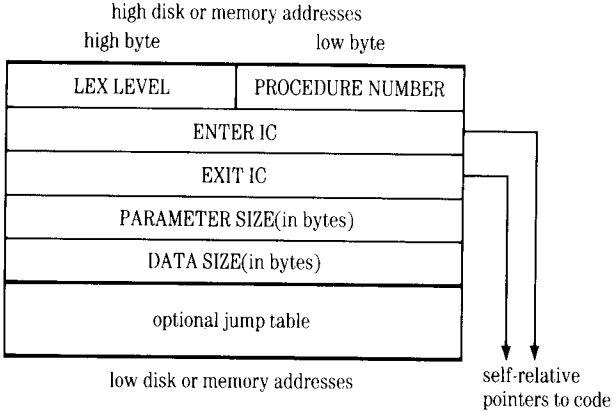
Code Part of a Code Segment



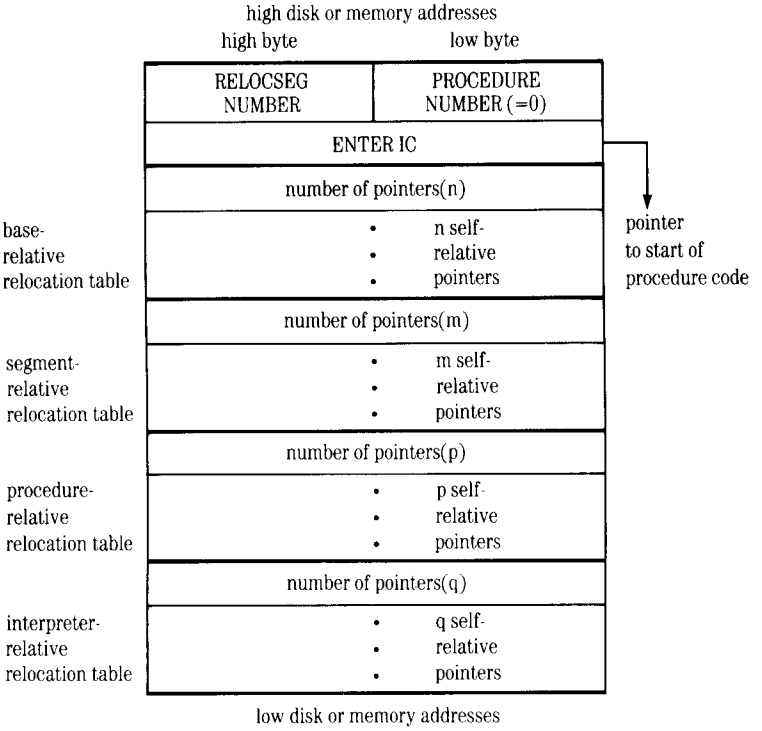
Procedure Code Structure



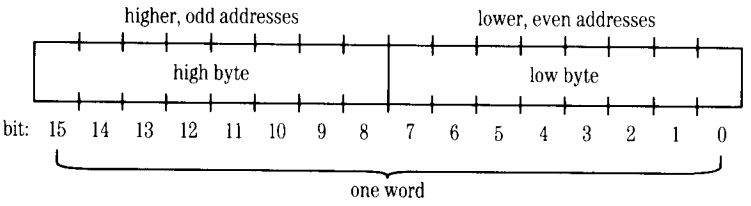
P-Code Procedure Attribute Table



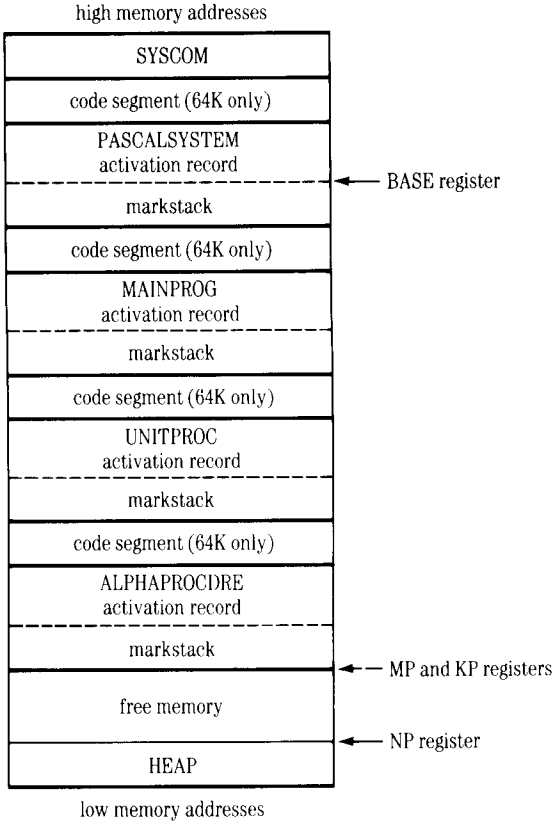
6502 Procedure Attribute Table

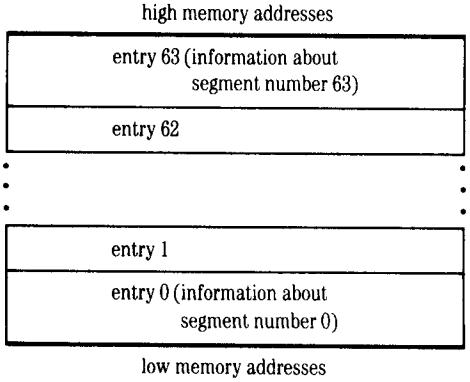


Bytes and Words

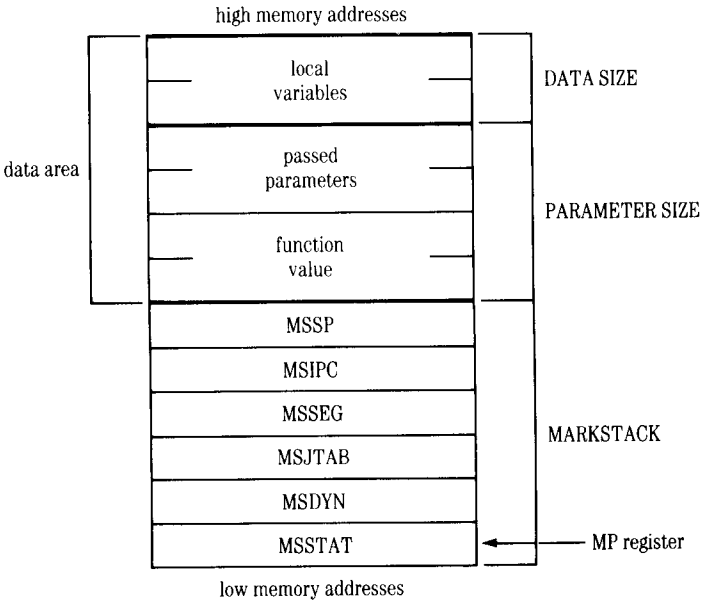


Program Stack




high memory addresses
entry 63 (information about segment number 63)
entry 62
entry 1
entry 0 (information about segment number 0)
low memory addresses

Activation Record



Variable Allocation in an Activation Record

For the declaration

```
VAR I, J : INTEGER;
BOOL : BOOLEAN;
```

the local variable portion of an activation record is constructed like this: