SOFTECH
MICROSYSTEMS

p-System™ Software
Reference Library

Internal Architecture

1-140.41.A

p-System™

Internal Architecture Reference Manual

SofTech Microsystems, Inc.
San Diego, California

1-140.41.A

Printed in the United States of America.

Disclaimer

This document and the software it describes are subject to change without notice. No warranty expressed or implied covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

PREFACE

This publication is a reference manual for the p-System. It covers the internal details of the p-System. The p-machine architecture and instruction set are covered. Code file format, low-level I/O mechanisms, and operating system details are also addressed.

For further information about the system and its use, refer to the following publications:

Personal Computing with the UCSD p-System
Operating System Reference Manual
Program Development Reference Manual
Assembler Reference Manual
Optional Products Reference Manual
Adaptable System Installation Manual
UCSD Pascal Handbook
FORTRAN-77 Reference Manual
BASIC Reference Manual

T A B L E

O F

C O N T E N T S

INTRODUCTION 1-3

PURPOSE OF THIS MANUAL 1-3

A BRIEF HISTORY OF THE SYSTEM . . . 1-5

CODE FILE FORMAT 2-3

INTRODUCTION 2-3

CODE SEGMENTS 2-3

Code Segments and Byte Sex 2-6

Routine Dictionary 2-7

Routine Code 2-7

The Constant Pool/Real Constants 2-8

The Relocation List 2-15

Segment Reference List 2-19
Linker Information 2-22

CODE FILE ORGANIZATION 2-29
The Segment Dictionary 2-29
Assembler-Generated Code Files 2-36

THE P-MACHINE 3-3

OVERVIEW 3-3
Emulative Execution 3-4
The Stack and the Heap 3-4
Code Segments 3-5
Device I/O 3-7

CODE SEGMENT ENVIRONMENTS 3-8
Segment Information Blocks (SIBs) 3-8
Environment Records (E_Recs) 3-13

TASK ENVIRONMENTS 3-17

P-MACHINE INSTRUCTIONS 3-22
The Intrinsic P_MACHINE 3-22

P-MACHINE REGISTERS 3-25
CURPROC 3-25
CURTASK 3-26
EREC 3-26
EVEC 3-26
IORESULT 3-27
IPC 3-27

MP 3-27
READYQ 3-28
SP 3-28

FAULTS 3-29

EXECUTION ERRORS 3-31

P-CODE INSTRUCTIONS 3-40
Instruction Parameters 3-42
Dynamic Operands 3-43
Activation Records 3-47

P-CODE DESCRIPTIONS 3-49

STANDARD PROCEDURES 3-182

UNIT I/O PROCEDURES 3-183

STRING PROCEDURES 3-191

COMPILER PROCEDURES 3-196

CODE POOL PROCEDURES 3-201

CONCURRENCY PROCEDURES 3-207

MISCELLANEOUS PROCEDURES 3-209

LONG INTEGERS 3-211
Number Format 3-211
Long Integer Constants 3-213
LONGOPS Routines 3-215
DECOPS Routines 3-216

Processor-Specific Information 3-224
8086/8088/LSI-11/6809/9900 3-224
68000 3-224
Z80/8080/6502 3-225
HP-87 3-225

LOW-LEVEL I/O 4-3

THE I/O SUBSYSTEM 4-3

DEVICE I/O ROUTINES 4-8
Calling the RSP/IO 4-9
Devices and Device Numbers 4-10
User-Defined Devices 4-10
CONTROL Parameters 4-11
IORESULT and Completion Codes 4-13
Logical Disk Structure 4-14
Physical Sector Addressing Mode 4-15
Physical Sector Numbers 4-16
Physical Sector Size 4-17

THE RSP 4-18
Calling Mechanisms 4-18
UNITREAD and UNITWRITE 4-19
Parameter Description 4-19
Parameter Stack Format 4-21
UNITBUSY 4-22
UNITWAIT 4-23
UNITCLEAR 4-24
UNITSTATUS 4-24

RSP Responsibilities 4-25
Special Character Output Handling . . . 4-25
Blank Compression Code (DLE's) . . . 4-26
Carriage Return — Line Feed 4-26
NOCRLF Bit 4-27
Special Character Input Handling 4-27
EOF Character 4-28
ALPHALOCK Character 4-29
Other Characters 4-29
NOSPEC Bit 4-30
Translation for Subsidiary Volumes . . . 4-30

BIOS 4-32
Design Goals 4-32
Completion Codes 4-33
Calling Mechanisms 4-34
Console 4-34
Printer 4-35
Disks 4-35
Remote 4-36
User-Defined Devices 4-36
Character Codes 4-37
BIOS Responsibilities 4-39
Console 4-39
Console Output Requirements 4-39
Console Output Options 4-41
Console Input Requirements 4-43
Console Input Options 4-43

START/STOP 4-44
FLUSH 4-45
BREAK 4-46
Type-Ahead 4-47
Input Character Mask 4-47
Initialization and Control 4-48
Console Status 4-50
Printer 4-50
Printer Output Requirements 4-51
Printer Input Requirements 4-52
Printer Initialization and Control . . . 4-52
Printer Status 4-53
Disk 4-53
Mapping Blocks on Physical Sectors . . 4-53
Bootstrap Location 4-54
Physical Sector Mode 4-55
Disk Output Requirements 4-56
Disk Input Requirements 4-57
Disk Initialization and Control 4-57
Disk Status 4-57
Remote 4-58
Remote Output Requirements 4-58
Remote Input Requirements 4-58
Remote Initialization and Control . . . 4-59
Remote Status 4-59
User-Defined Devices 4-59
Special BIOS Calls 4-60
System Output 4-60

System Input 4-60
System Initialization and Control 4-60
System Status 4-61

BIOS CALLING CONVENTIONS 4-62

PROCESSOR-SPECIFIC BIOS CALLS . . . 4-64
8086/8088 4-64
8080/Z80 4-67
6502 4-69
6809 4-71
68000 4-73

THE OPERATING SYSTEM 5-3

OVERVIEW OF THE OS 5-3

P-MACHINE SUPPORT 5-5
The Heap: An Overview 5-5
MARK and RELEASE 5-5
NEW and VARNEW 5-6
DISPOSE and VARDISPOSE 5-7
PERMNEW and PERMDISPOSE 5-7
Heap Implementation 5-8
Unit Organization 5-8
Heap Globals 5-9
Tactics 5-11
Run-Time Environment 5-12

THE CODE POOL 5-13
Fault Handling 5-18
Concurrency 5-20

I/O SUPPORT 5-22
FIBs 5-22
Directories 5-24

VARIETIES OF I/O 5-25
Record I/O 5-25
Screen I/O 5-25
Block I/O 5-25
Text I/O 5-26

PROGRAM EXECUTION 6-3

BUILDING A RUN-TIME ENVIRONMENT . . 6-3

QUICKSTARTING PROGRAMS 6-5
Program Invocation Overview 6-6
Segment Dictionary Structure 6-10
PED Structure 6-13

Table of Contents

APPENDICES

A: P-MACHINE OPCODES (Alpha) A-3
B: P-MACHINE OPCODES (Numeric) A-7
C: P-MACHINE INTRINSICS A-11
D: PASCAL DEFINITIONAL RSP A-12
E: PASCAL DEFINITIONAL BIOS A-19
F: ASCII TABLE A-27
G: GLOSSARY A-28

INDEX I-1

CHAPTER 1

INTRODUCTION

PURPOSE OF THIS MANUAL

This manual describes the internal design of the p-System™. The coverage includes the p-machine, operating system, basic I/O, and the way in which these elements are organized to support the running of a program written in UCSD Pascal®, BASIC, or FORTRAN-77.

It should serve as a guide and reference for more advanced users of the p-System, but isn't intended to be a stand-alone definition for the use of implementors. Such a definition doesn't yet exist; if one is written, it will probably be based on the format of this book.

Perhaps the best way to use this manual is to read it sequentially, skipping those sections (such as the list of p-codes) that go into very specific detail. This should give the reader a fairly complete picture of what goes on within the p-System. If the user then needs to know specific internal details, the relevant section can be referred to later.

While few users will want or need to implement a p-System from scratch, the internal descriptions provided in this guide should be useful to a number of audiences.

The largest audience is probably those who will make no specific use of the information. To these users, the benefit will be a better understanding of the p-System's operation and a general improvement in their ability to engineer programs for effective execution in the p-System environment.

Second, there are the implementors of system software facilities that complement existing p-System capabilities; for instance, new language translators, new system utilities, or p-machine emulators (PMEs) for additional processors. For this group of programmers, the Internal Architecture Reference Manual presents more information than was available in the past.

Finally, there are the implementors with a compelling need to use facilities such as the ability to explicitly generate p-codes in a Pascal program, where an ordinary Pascal construct wouldn't suffice (we take it for granted that only a compelling need would lead you to take such steps).

All of these audiences (but particularly the last) should understand that the principal commitment of SofTech Microsystems (and its licensees) is to the user facilities, and not to any of the specific implementation strategies that are described in this guide. Programmers who take advantage of "internal tricks" do so at their own risk.

A BRIEF HISTORY OF THE SYSTEM

The software system that is now called the p-System began when Kenneth Bowles was responsible for teaching the introductory programming course at the University of California, San Diego. In late 1974, under Bowles' direction, a group of undergraduate and graduate students began to implement Pascal for microcomputers.

Before this time, the introductory programming course had been taught using a large time-shared computer. This presented a bottleneck—many people used the machine, so its turnaround was sometimes quite slow, and a student's productivity was to some extent limited by the availability of the card punches. Furthermore, the machine's time-sharing environment, its accounting system, its complexity, and the amount of sensitive information that it stored prevented the student from any extensive "hands on" use of the machine or its facilities. In brief, the computer was intimidating.

These were the main reasons for the decision to change the nature of the beginning programming course. It would be self-paced, to accommodate the large number of students, and each individual student's study habits (UC—Irvine's physics program had been doing this successfully for a couple of years). It would use Pascal, rather than the dialect of Algol that was specific to the University's large time-sharing computer; and, it would use microcomputers.

The decision to use small computers was motivated partly by their low cost, and partly by the desire to give students an opportunity to program in an interactive environment. The system was first implemented for a number of PDP-11/10's with floppy disks and VT-50 terminals. Students were expected to buy their own floppy disk, and use it for storing the system and their own programs.

It was the interactive environment that led to some of UCSD Pascal's deviations from the standard language, mostly as regards INTERACTIVE files and the handling of EOF and EOLN. The type STRING came about from the desire to teach basic programming concepts without recourse to numerical problems (which distracted many students from the actual problems of programming).

The user interface of the p-System, by which we mean the philosophy of displaying a menu or prompt at every level of the p-System, and organizing them in a tree structure, was intended to be easy to learn for the complete novice, yet usable (that is, not cumbersome) for the experienced user. This proved very successful, and has been retained.

The emulative approach to executing Pascal was present from the beginning. P-code, adapted from the original design by Urs Amman of the "Eidgenossische Technische Hochschule," in Zurich, was designed to be compact and easily generated by a compiler; because of the constraints of the microprocessor environment, the goal was to keep the compiler and the code files as small as possible. The tradeoff in execution time was felt to be an affordable cost (time has borne out this decision).

All of the original implementations were on PDP-11/LSI-11 machines. Because of the emulative approach, it was a relatively straightforward matter to rewrite the p-machine emulator for the 8080 and Z80, and subsequently, for many other processors.

This adaptation of the PME (sometimes called the interpreter) was originally motivated by the search for less expensive hardware, but it was soon recognized that software portability was valuable in itself. The economics of the computer business, especially the microprocessor field, dictated this. It isn't a new observation that hardware costs continue to plummet, while software, being "hand-made," continues to be very expensive; it is relatively new to encounter a software system that, through modularity and portability, addresses the problem as thoroughly as does the p-System.

Code File Format

CHAPTER 2

CODE FILE FORMAT

INTRODUCTION

This chapter describes the internal format of p-System code files. Code files may contain either p-code or native code. P-code is the output of the compilers and is described in Chapter 3. Native code is specific to a particular processor and is output by the assemblers and the Native Code Generator. Code files also contain various sorts of "housekeeping" information.

CODE SEGMENTS

A code segment is a section of executable code which is brought into memory as a whole unit. Each segment consists of a collection of routines (procedures, functions, and so forth), together with descriptive information. The code and information in a segment are contiguous since the code segment is the "unit of movement" for code.

There are up to 255 routines within a segment, numbered 1 through 255.

At compile time, segments are assigned a name and a number. The name is eight characters long. It is used by the operating system to handle intersegment references at associate time. (Associate time is the time it takes the operating system to stitch together the units referenced by a program.) It is also used when maintaining code files with LIBRARY. The number is used to reference the segment at run-time.

The beginning (low address) of a code segment is a record that contains the following information about the segment:

pointer to the procedure dictionary
pointer to the relocation list
the eight character name of the segment (four words)
byte sex indicator
pointer to the constant pool
real size indicator
space reserved for future use (two words)

Figure 2-1 illustrates a code segment as it would be loaded into memory. The various substructures of a code segment are described below. Note that all fields within a code segment are word-aligned. Also, all intersegment pointers are word offsets from the base of the segment.

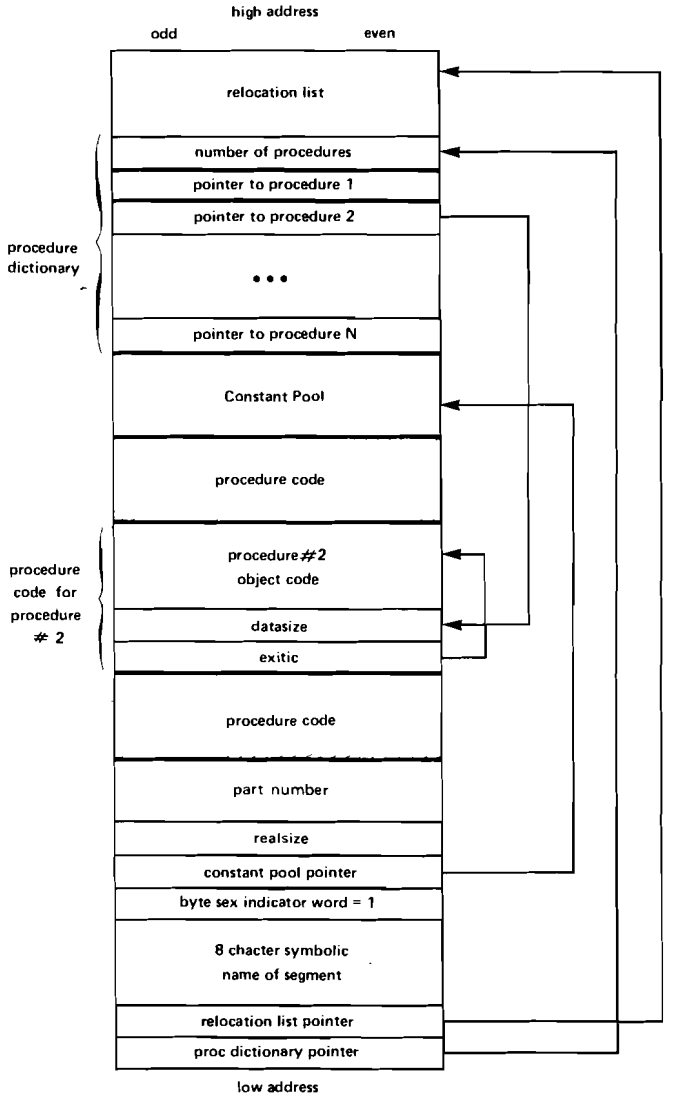


Figure 2-1. Executable Code Segment Format

Code Segments and Byte Sex

Code segments are independent of the byte sex of the host processor. A number of system components cooperate to achieve this independence.

There are two groups of word-oriented (byte-sex-dependent) information. The first is superstructure information, such as the routine dictionary. This information is flipped by the operating system when a segment is loaded. The second is embedded information, such as constants (accessed by the Load Constant (LDC) p-machine instruction or by Case Jump (XJP) tables). This sort of information is flipped by the PME.

The compiler produces code segments that contain word information in the natural order of the machine on which the compiler is run. Immediately following the segment's 8 character name is a flag that always contains the constant 1, in the byte sex of the original machine; if read in the opposite byte sex, it appears to be 256.

When a segment is loaded by the operating system, and its byte sex flag indicates that the sex of the segment is opposite that of the running machine, the segment superstructure is byte-swapped. Embedded information is then flipped by the PME.

The net result is that segments of either sex can run on any machine.

Routine Dictionary

The first word in a code segment points to word 0 of the segment's routine dictionary (also called the "procedure dictionary"). The routine dictionary is a list of pointers to the code for each routine in the segment. Each routine dictionary pointer is a segment relative word pointer.

Routines within a segment are numbered 1 through 255. A routine's number is a negative index into the routine dictionary; the n'th word in the dictionary contains a pointer to the code for routine n.

The first word (word 0) of the dictionary contains the number of routines in the segment.

In the case of EXTERNAL and FORWARD routines, the source code may contain a routine's declaration but not its code. The corresponding routine dictionary entry is zero (at least, before linking).

Routine Code

The code of a routine consists of two words: Data_Size and Exit_IC, followed by the executable object code. The object code may be entirely p-code, entirely native code, or a mixture of the two.

Data_Size is the number of words of local data space that must be allocated when the procedure is called. Data_Size doesn't include parameters; the routine's parameters are assumed to already be on the stack. The first executable instruction starts at the word immediately following the Data_Size word. If the first executable instruction is native code, Data_Size is negative. No local data space is allocated for assembly language procedures.

If this first instruction is a p-code instruction, then Exit_IC is a segment relative byte pointer to the code that must be executed when the procedure is exited. Otherwise, Exit_IC is undefined at run-time.

If the code of the routine contains both p-code and native code, it is still the first instruction of the routine that determines these conditions. Procedure code produced by a Native Code Generator always starts with a p-code. Thus, Data_Size and Exit_IC are defined as in a procedure which consists entirely of p-code.

The Constant Pool/Real Constants

Multi-word constants are stored together in a single constant pool for the entire segment. The constant pool begins immediately after the last body of procedure code in the segment.

The location of the constant pool is contained in the constant pool pointer, a segment relative word pointer that immediately follows the byte sex indicator word at the beginning of the segment; it points to the low address of the constant pool. If the constant pool pointer is equal to zero, the segment doesn't contain a constant pool.

Constants are referenced by word offsets relative to the beginning (low address) of the constant pool.

The constant pool is divided into two subpools: the real pool and the main pool.

The first word of the constant pool points to the beginning of the real pool. This is a word pointer relative to the start of the constant pool; if there are no real constants in the code segment, this word will be 0. The first word of the real pool contains the number of real constants in the real pool.

Figure 2-2 illustrates a constant pool with an embedded real subpool.

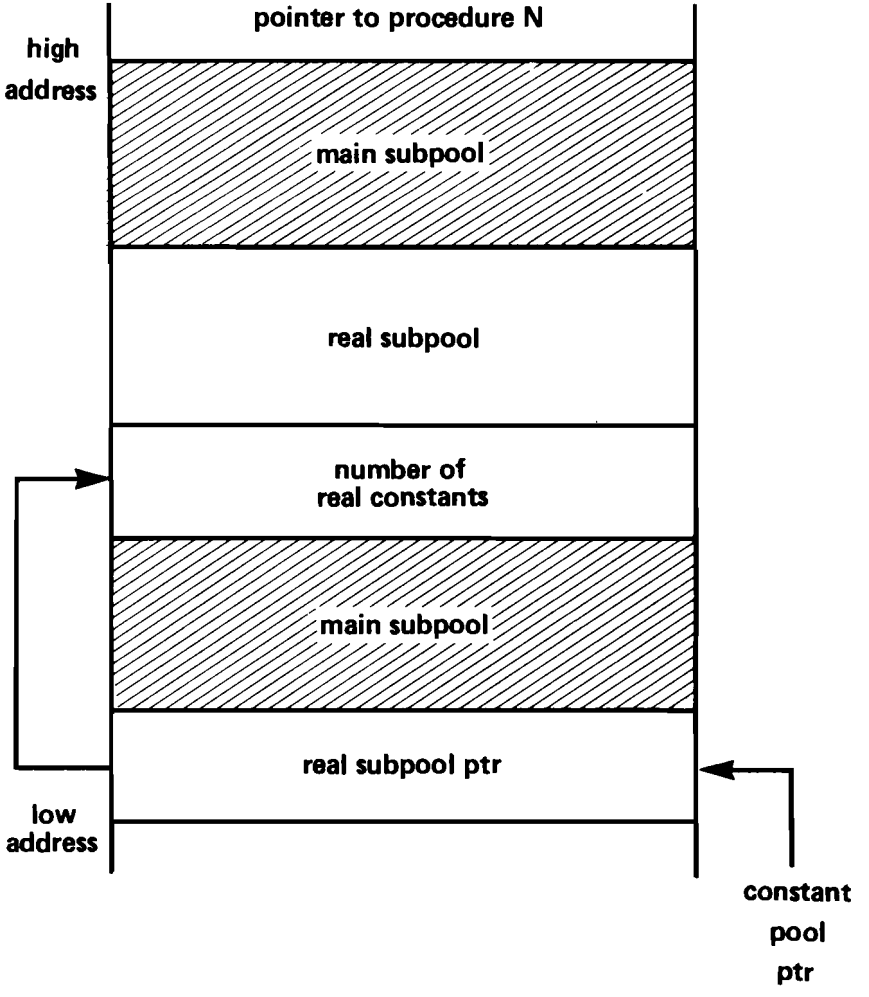


Figure 2-2. Constant Pool

Real constants are compiled into a processor-independent ("canonical") format and are converted, at segment load-time, into a processor-specific internal format. Real constants are generated as either 2-word (32-bit) or 4-word (64-bit) floating point data formats. Code files containing real constants can be transported across all p-System implementations which use the same real size. In order to transport them to a machine using a different real size, they must be recompiled. Within a single program, all compilation units must share the same size for real values.

The default real size of a code file created by the compiler is determined by the p-machine emulator in use at compile-time. The $R compiler directive may override this default, however.

The real size at compilation time is embedded in every code segment (even though it may not reference any reals). The Real_Size word at the base of the segment contains this value.

A 32-bit real constant is represented by a three-word record. The first word contains a signed integer representing the exponent value. The following two words contain the mantissa digits. A mantissa word representing significant mantissa digits contains an integer whose absolute value is between 0 and 9999; its value corresponds to four mantissa digits. The first mantissa word is signed and, thus, contains the mantissa sign. The second mantissa word may contain a negative value; in this case, it doesn't contain any significant digits and is disregarded when constructing the internal representation of the real constant. It serves as a terminator word for the constant conversion routines. The decimal point is defined to lie to the right of the four digits in the last valid (used) mantissa word. The digits in the last mantissa word are left-justified. For example, if the real value is 1.1, the first mantissa word contains 1100 decimal (or 044C hexadecimal).

Example:

1..4 significant mantissa digits:
The first mantissa word contains a signed value between 0 and 9999. The second word contains a negative value. The implied decimal point position is at the end of the first word.

5..8 significant mantissa digits:
The second mantissa word contains a positive value between 1 and 9999, and represents up to four low-order digits. The first word contains a signed value between 1 and 9999; it represents the four high-order digits. The implied decimal point position is at the end of the second word.

A 64-bit real constant is represented by a record whose length may vary between four and six words, depending upon the number of significant digits in the constant. The first two words of a 64-bit constant are identical in format to those of a 32-bit real constant; thus, the format always contains an exponent word and a first mantissa word. An enumeration of the remaining words for all cases follows:

1..4 significant mantissa digits:
Mantissa word 2 contains a negative terminator. Word 3 is zeroed and is present solely to provide sufficient space for the native format.

5..8 significant mantissa digits:
Mantissa word 2 contains 1 to four digits (left-justified). Word 3 contains a negative terminator.

9..12 significant mantissa digits:
Mantissa word 2 contain four digits. Word 3 contains one to four digits (left-justified). Word 4 contains a negative terminator.

13..16 significant mantissa digits:
Mantissa words 2 — 3 contain four digits. Word 4 contains one to four digits. Word 5 contains a negative terminator.

17..20 significant mantissa digits:
Mantissa words 2 — 4 contain four digits. Word 5 contains one to four digits.

Real constants are converted to native machine format when a code segment is loaded into memory; this may result in a significant run-time overhead for programs that are memory bound. Time-critical programs of this nature may sacrifice portability for execution speed by using Real Convert utility to convert their real subpools into native machine format within the code file itself. This is done by replacing the canonical form of each real constant in the code file with a native real constant. The modified subpool is merged with the main pool by setting the real pool pointer to zero, thus eliminating the usual conversion process during a segment load. Because the constant pool is transformed in place, constant offsets embedded in the code file don't require updating. (This, of course, reduces the portability of the program.)

The Relocation List

The last (high address) body of information in a code segment is the relocation list. The second pointer at the beginning of the code segment points to the last (highest address) word in the relocation list. This pointer is a segment relative word pointer; if there is no relocation list, it is equal to zero.

The relocation list contains all the information necessary to fix any absolute addresses used by code within the segment, whenever the segment is loaded or moved in memory. Such absolute addresses are needed only by native code. Segments containing exclusively p-code are completely position-independent; no relocation list is needed.

A relocation list consists of zero or more relocation sublists. Each sublist contains code offsets for objects that must be relocated, and specifies the type of relocation that must be done. Sublists can occur in any order, and more than one sublist can have the same type of relocation.

The following code fragment shows the format of the heading of a sublist:

```
Loc_Types=(Reloc_End, {signals end of entire relocation list}
       Seg_Rel,   {relative to address of base of this segment}
       Base_Rel,  {relative to data segment given in DATASEGNUM}
       Interp_Rel,{relative to PME's interp-relative table}
       Proc_Rel); {relative to address of lst instruction in proc}

List_Header=PACKED RECORD
          List_Size: integer;   {number of pointers in sublist}
          Data_Seg_Num: 0..255;  {local segment number for Base_Rel}
          Reloc_Type: Loc_Types; {relocation type of sublist entries}
        END;
```

Each sublist contains a List_Header and zero or more segment relative byte pointers to the objects which must be relocated. The Reloc_Type field in the List_Header defines what kind of relocation will be applied to all objects designated by the sublist.

The relocation type Proc_Rel is generated by the assembler, but is changed by the linker into Seg_Rel. Proc_Rel sublists should never be encountered when loading and relocating assembly code.

The Data_Seg_Num field in the List_Header is only used in sublists with a Reloc_Type of Base_Rel, and in all other cases should be zeroed. It specifies the local segment number of the data segment to which all the sublist's pointers are relative. Since the assembler can't know this segment number in advance, it should zero-fill the field and leave the responsibility for correctly setting this field to the linker.

The List_Size field in the List_Header contains the number of pointers in the sublist.

Figure 2-3 illustrates a relocation list with multiple sublists.

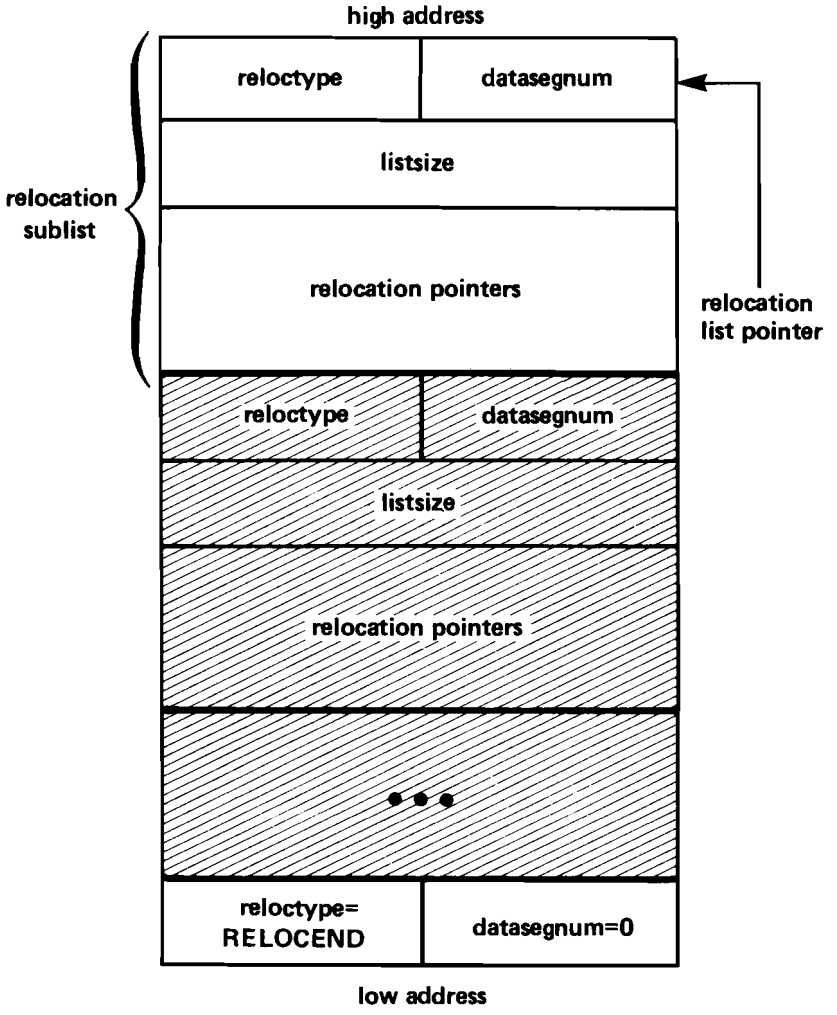


Figure 2-3. Relocation List

The relocation list is intended to be used from high address down to low address. Each sublist in turn is processed from high to low until a sublist with a relocation type of Reloc_End is encountered. The Data_Seg_Num and List_Size should be 0 for this terminating entry.

The relocation list is located at the end of the code segment, since it is sometimes possible to discard the relocation information after the segment has been loaded into memory.

Segment Reference List

In the p-machine (described in the next chapter), each code segment is associated at run-time with an "environment vector" that defines the mapping of each segment number to the segment or unit that it designates. Each compilation unit has its own independent (that is, local) series of segment numbers, and its own environment vector. In this way, a particular unit may be referenced by more than one unit, and each unit that references it may use a different segment number. (More about environment vectors appears in the section, "Code Segment Environments" in the next chapter.)

When a compilation unit references one or more other compilation units, the principal segment of the compilation contains a segment reference list. This list defines the connection between the segment numbers that appear in the object code (they are created by the compiler), and the names of the units to which they refer. Only principal segments contain segment reference lists.

The segment reference list, when present, is located above the relocation list (it grows toward higher memory addresses). The list is used by the operating system at associate time. It doesn't occupy any space in memory during the program's execution (since the segment length field doesn't include it).

The segment reference list associates the name of each compilation unit (which doesn't change) with the number by which that that compilation unit is referenced.

The following fragment of Pascal code describes a record in the segment reference list:

```
Seg_Rec= PACKED RECORD
Seg_Name: PACKED ARRAY [0..7] OF CHAR; {referenced segment name}
        Seg_Num: 0..255; {associated segment number}
        Filler: 0..255; {reserved for future use}
      END;
```

The Seg_Refs entry in the segment dictionary (described below) contains the number of words in the segment reference list. The Code_Leng field in the segment dictionary can be used as a segment relative word pointer to the start of the segment reference list. The segment reference list consists of one or more Seg_Rec's, starting directly above the relocation lists (or procedure dictionary) and continuing towards higher memory addresses. A Seg_Rec consists of Seg_Name, which contains the name of the segment; Seg_Num, which contains the number by which the segment is referenced within this current code segment; and some filler.

The segment reference list is terminated by a Seg_Rec with a blank-filled Seg_Name and Seg_Num of zero.

Seg_Rec's with a Seg_Name of '***' are generated so that the operating system can execute the initialization and termination code sections of a unit. Before executing a host program, the operating system constructs a list of all units used that contain a reference to '***', and uses this list to execute the initialization/termination sections of all such units before/after the host program is invoked.

When the initialization/termination section of a unit (which is procedure 1) is compiled, the following instruction is emitted between the initialization and termination parts:

```
CXG <***'s Seg_Num>, 1
```

where CXG is the p-code representation of a global procedure call. A local segment number is reserved for the '***' segment reference, and the operating system creates a linear list that links together the units of a program that require initialization. At the end of this list is the outer body of the main program. The operating system invokes the program by calling the first initialization code on this list, which calls the next, and so forth up to the body of the main program. When the main program terminates, the calling chain is "popped," and termination sections are executed in the reverse order.

Linker Information

Linker information is a portion of a code segment that allows the linker to resolve references between p-code and native code. Segments output by an assembler always have linker information. Segments output by a compiler have linker information only if they contain an EXTERNAL routine. Only principal segments may contain EXTERNAL routines.

Linker information is a sequence of 8-word records, starting on the block boundary following the end (high address) of the segment reference list. The end of the sequence contains the value EOF_Mark. Linker information records are always eight words long; unused records and unused fields are zero-filled.

If a code segment has linker information, the Has_Linker_Info boolean in Seg_Misc in the segment dictionary is TRUE. The starting block of linker information, relative to the start of the code file, can be calculated from the formula:

```
Code_Addr + ((Code_Leng + Seg_Refs + 255) DIV 256)
```

where Code_Addr, Code_Leng, and Seg_Refs are all values in the segment dictionary (see below).

Two fields are common to all linker information records. The Name field contains an 8-character segment name. The LI_Type field determines the nature of the linker information in the remainder of the record.

The following fragment of psuedo-Pascal code describes a linker information record:

```
Ptr_Rec_Num = {an integral number of 8-word pointer records}
              {this is variable from record to record};

LI_Types    = (EOF_Mark, Glob_Ref, Publ_Ref, Priv_Ref, Const_Ref,
               Glob_Def, Publ_Def, Const_Def, Ext_Proc, Ext_Func,
               Sep_Proc, Sep_Func);

LI_Entry    = RECORD
               Name: PACKED ARRAY [0..7] OF CHAR;
               CASE LI_Type: LI_Types OF

                Glob_Ref, Publ_Ref, Const_Ref
                        : (Format: (Word, Byte, Big);
                           N_Refs:  integer);

                Priv_Ref: (Format: (Word, Byte, Big);
                           N_Refs:  integer;
                           N_Words: integer);

                Ext_Proc, Ext_Func
                        : (Src_Proc: integer;
                           N_Params: integer);

                Sep_Proc, Sep_Func
                        : (Src_Proc: integer;
                           N_Params: integer;
                           Kool_Bit: boolean);

                Glob_Def: (Home_Proc: integer;
                           IC_Offset: integer);

                Publ_Def: (Base_Offset: integer;
                           Pub_Data_Seg: integer);

                Const_Def: (Const_Val: integer);

                EOF_Mark:
              END {CASE};

            Ptr_List: ARRAY[0..Ptr_Rec_Num] OF
                      ARRAY [0..7] OF integer

          END {LI_Entry};
```

Glob_Ref, Publ_Ref, Const_Ref, and Priv_Ref are linker information types generated by an assembler. Each consists of two fields that precede a list (Ptr_List) of segment relative byte pointers into the associated segment. Format contains the size of the fields pointed to by the accompanying list. N_Refs contains the number of pointers in the list. Ptr_List contains multiples of eight words; all unused words should be zero.

For these types of linker information records, Ptr_Rec_Num = ceiling(N_Refs/8), where ceiling(n) is the smallest integer >= n.

Glob_Ref is used to link identifiers in two or more assembled routines. Name is an identifier that is referenced within the segment and defined in some other assembled routine. Format should always be word. The linker must add the final segment offset of the referenced object to all words pointed at by Ptr_List. This offset must be in the correct addressing mode, that is, in bytes or words, depending on the processor being used.

Publ_Ref is used to link an identifier in an assembled routine to a global variable in a compilation unit. Name is an identifier that is referenced in the segment and defined as a global variable in some other compilation unit. Format should always be word. The linker must add the offset of the referenced object to all words pointed at by Ptr_List.

Const_Ref is used to link an identifier in an assembled routine to a global constant in a compilation unit. Name is an identifier that is referenced in the segment, and defined as a global constant in some compilation unit. Format may be either byte or word. The linker must place the constant value into all locations pointed at by Ptr_List.

Priv_Ref is used to allocate space in the global data segment. Format should always be word. N_Words specifies the number of words to allocate. The linker must add the offset of the start of the allocated area within the global data segment to all words pointed at by Ptr_List.

Ext_Proc and Ext_Func are generated by a compiler to reference EXTERNAL routines. There is no Ptr_List. Src_Proc is the number assigned to the routine. N_Params is the number of words allocated for parameter passing.

Sep_Proc and Sep_Func are generated by an assembler for routine declarations. There is no Ptr_List. Src_Proc is the number assigned to the routine. N_Params is the number of words allocated for parameter passing. Kool_Bit is TRUE if the routine is relocatable, and FALSE otherwise. Thus, with Kool_Bit = FALSE, and .RELPROC and .RELFUNC generate Sep_Proc or SepFunc records with Kool_Bit = TRUE.

Glob_Def declares a global identifier in an assembled routine. A Glob_Def record is generated for each label defined by a .DEF, .PROC, .FUNC, .RELPROC, or .RELFUNC directive. There is no Ptr_List. Name is an identifier defined within the segment, and may be referenced by any other assembled routines within the same segment. Home_Proc contains the number of the routine in which Name is defined. IC_Offset is a byte offset to Name, relative to the start of the routine in which Name is defined.

Publ_Def declares a global variable in a compilation unit. A Publ_Def record is generated for each global variable in a compilation unit that is visible to any EXTERNAL routines. There is no Ptr_List. Base_Offset is the word offset of the variable, relative to the start of the data segment that contains it. Pub_Data_Seg is the local number of the data segment that contains the variable.

Const_Def declares a global constant in a compilation unit. A Const_Def record is generated for each global constant in a compilation unit that is visible to any EXTERNAL routines. There is no Ptr_List. Const_Val contains the value of the constant.

EOF_Mark indicates the end of used linker information records. Name should be blank-filled.

The following table shows the types of segments (as defined in the segment dictionary), and the types of segment reference records that can be contained in the associated linker information. Note that Proc_Seg's can't have linker information at all.

	Prog_Seg	Unit_Seg	Seprt_Seg
Glob_Ref			yes
Publ_Ref			yes
Priv_Ref			yes
Const_Ref			yes
Ext_Proc	yes	yes	
Ext_Func	yes	yes	
Sep_Proc			yes
Sep_Func			yes
Glob_Def			yes
Publ_Def	yes	yes	
Const_Def	yes	yes	
EOF_Mark	yes	yes	yes

CODE FILE ORGANIZATION

The Segment Dictionary

The first block of a code file contains the first record of that file's segment dictionary. A segment dictionary consists of a linked list of dictionary records; if the dictionary is longer than one record, subsequent records are embedded in the code file. These are each one block long, and are located between code segments.

A single dictionary record can describe up to 16 distinct segments. The information describing a segment is contained in six different arrays; the information describing a segment is found by using a single index value to select a component from each of these arrays. Entries in the segment dictionary describe only segments whose code bodies are included in the code file.

The following fragment of Pascal code describes a segment dictionary record:

```
CONST Max_Dic_Seg = 15; {maximum segment dictionary record entry}

TYPE  Seg_Dic_Range = 0..Max_Dic_Seg; {range for segment dictionary entries}

   Segment_Name = PACKED ARRAY [0..7] OF CHAR; {segment name}

   {segment types}
       Seg_Types = (No_Seg,     {empty dictionary entry}
                    Prog_Seg,   {program outer segment}
                    Unit_Seg,   {unit outer segment}
                    Proc_Seg,   {segment procedure inside program or unit}
                    Seprt_Seg); {native code segment}

   {machine types}
       M_Types = (M_Psuedo, M_6809, M_PDP_11, M_8080, M_Z_80,
                            M_GA_440, M_6502, M_6800, M_9900,
                            M_8086, M_Z8000, M_68000, M_HP87);

   {p-machine versions}
       Versions = (Unknown, II, II_1, III, IV, V, VI, VII);

   {segment dictionary record}
       Seg_Dict = RECORD
         Disk_Info:
           ARRAY [Seg_Dic_Range] OF {disk info entries}
             RECORD
               Code_Addr: integer;  {segment starting block}
               Code_Leng: integer;  {number of words in segment}
             END {of RECORD};
         Seg_Name:
           ARRAY [Seg_Dic_Range] OF Segment_Name; {segment name entries}
         Seg_Misc:
           ARRAY [Seg_Dic_Range] OF {misc entries}
             PACKED RECORD
               Seg_Type: Seg_Types;     {segment type}
               Filler: 0..31;           {reserved for future use}
               Has_Link_Info: boolean; {need to be linked?}
               Relocatable: boolean;    {segment relocatable?}
             END {of PACKED RECORD};
         Seg_Text:
           ARRAY [Seg_Dic_Range] OF integer; {start blk of interface text}
         Seg_Info:
           ARRAY [Seg_Dic_Range] OF {segment information entries}
             PACKED RECORD
               Seg_Num: 0..255;          {local segment number}
               M_Type: M_Types;          {machine type}
               Filler: 0..1;             {reserved for future use}
               Major_Version: Versions; {p-machine version}
             END {of PACKED RECORD};
         Seg_Famly:
           ARRAY [Seg_Dic_Range] OF {segment family entries}
         RECORD
           CASE Seg_Types OF
             Unit_Seg, Prog_Seg:
               (Data_Size: integer;   {data size}
                Seg_Refs: integer;    {segments in compilation unit}
                Max_Seg_Num: integer; {number of segments in file}
                Text_Size: integer);  {# of blks interface text}
             Seprt_Seg, Proc_Seg:
               (Prog_Name: Segment_Name); {outer program/unit name}
         END {of Seg_Famly};
```

```
    Next_Dict: integer; {block number of next dictionary record}
    Filler: ARRAY [1..2] OF integer;
    Checksum: integer;        {see QuickStart in Chapter 6}
    Ped_Block: integer;       {see QuickStart in Chapter 6}
    Ped_Blk_Count: integer;   {see QuickStart in Chapter 6}
    Part_Number: RECORD
                 A, B: integer;
                 END;
    Copy_Note: string[77]; {copyright notice}
    Dict_Byte_Sex: integer; {machine sex (Sex = 1)}
END {of SEG_DICT};
```

Disk_Info contains information about the segment's location within the file. Segment code always starts on a block boundary. Code_Addr is the number of the block where the segment code starts (relative to the start of the code file). Code_Leng is the number of 16-bit words in the segment. This size includes the relocation list but doesn't include the segment reference list. All unused entries in this array should be zeroed.

Seg_Name contains the first eight characters of the program, unit, segment, or assembly procedure name. Unused entries should be blank-filled.

Seg_Misc contains miscellaneous information about the segment. Seg_Type indicates the type of segment. Prog_Seg and Unit_Seg are outer segments of programs and units, respectively. Proc_Seg is a segment routine within either a unit or a program outer segment. Seprt_Seg is an unlinked native code segment. Has_Link_Info indicates whether linker information has been generated for this segment. Linker information resides in the blocks that directly follow the segment reference list. Linker information starts on a block boundary. The Boolean Relocatable specifies whether a code segment is statically or dynamically relocatable.

Dynamically relocatable code segments reside in the code pool (described in the next chapter); their position in memory may change many times during execution. Statically relocatable code segments are loaded only once, in a fixed position on the system heap (also described in the next chapter); they remain position-locked and memory-locked throughout their lifetime.

All segments that contain only p-code are position-independent and, thus, dynamically relocatable. Segments that contain native code may be dynamically relocatable provided they make no assumptions about either the lifetime of any modifications made to the segment body itself or to the exact location of the segment body in memory across the execution of a single p-code.

Dynamically relocatable native code is generated by assembling routines using the RELPROC or RELFUNC assembler directives; a linked code segment containing assembler routines is dynamically relocatable only if all of its assembler routines were originally specified as dynamically relocatable. Note that the use of these assembler directives is an assertion by the programmer that the routines declared behave properly; the system doesn't enforce this, so caution must be used. If a routine is to be dynamically relocatable, you can't expect it to store information in the segment body, be self-modifying, or store pointers to the code segment in data variables, and then assume that things will proceed correctly the next time the routine is called.

The Boolean Relocatable is unaffected by the presence or absence of relocation lists, and isn't relevant to concurrency considerations.

Seg_Text contains the starting block of the segment's INTERFACE text section, relative to the start of the code file. The INTERFACE text section can appear anywhere within the code file that contains the code segment it describes. The Seg_Text array entry, in conjunction with the Text_Size field in the Seg_Family record, indicates the address and length of the INTERFACE section in blocks. The INTERFACE text section always starts on a block boundary and follows all of the conventions of a text file with the exception that the last page of the section may be either one or two blocks long. Only segments with a Seg_Type of Unit_Seg have INTERFACE sections. All other segments and unused entries should be zero-filled.

Seg_Info contains further information about the segment. Seg_Num is the segment number. M_Type tells what kind of object code is in the segment. If there is any native code in the segment, then M_Type will have one of the processor-specific M_Type's. If the segment consists exclusively of p-code, then its M_Type is M_Psuedo. Major_Version gives the version of the p-machine on which the code file is intended to run.

Seg_Famly contains information about the code segment's compilation unit. The information contained in this array depends on whether Seg_Type indicates a principal or a subsidiary segment.

If the segment is a subsidiary segment, then Seg_Famly contains the first eight characters of the parent compilation unit's name, stored in Prog_Name. If this name isn't known at code file generation time (as is the case with Seprt_Seg's), the field should be blank-filled.

If the segment is a principal segment, then the information in Seg_Famly consists of four fields:

- Data_Size is the number of words in this segment's base data segment. The variables of principal segments are referenced from any location, including their own outer routine bodies, via global loads and stores (rather than local operations). Therefore, the Data_Size field associated with the body of an outer routine in a code segment should be zero, so that no superfluous memory will be allocated in an unused local data area.

- Seg_Refs is the size in words of the segment reference list for this segment.

- Max_Seg_Num is the total number of segment numbers assigned to this compilation unit. Max_Seg_Num includes all segments with assigned numbers, regardless of whether the segment body is contained in this file or not.

- Text_Size is the number of blocks of INTERFACE text within the compilation unit. Text_Size is used in conjunction with the Seg_Text array to specify the INTERFACE text for a compilation unit of type Unit_Seg; it is zero-filled for all other compilation unit types.

If the segment is unused (Seg_Type = No_Seg), then Seg_Famly should be zero-filled.

Next_Dict contains the block number of the next segment dictionary record, relative to the start of the code file. In the last record of the segment dictionary, Next_Dict should be zero.

Part_Num contains the SofTech internal part number for the file.

Filler is reserved for future use and should always be zero-filled.

Copy_Note is reserved for a copyright message, which can be created with either the LIBRARY utility or a compiler directive.

Sex corresponds to the byte sex of the segment dictionary. It is a full word that contains the value 1, with the same byte sex as the rest of the dictionary record. Thus, when this word is examined by a program running on a machine with the same byte sex as the code file, it will appear as a 1; on a machine of opposite sex, it will appear as a 256. System programs use this word to detect the sex of the dictionary, and, if necessary, byte-swap the word-oriented fields of the dictionary.

Assembler-Generated Code Files

Code files generated by an assembler have a slightly different structure from those generated by a compiler. A relocation list is generated for each procedure in an assembler-generated segment (instead of one relocation list for the whole segment). These are the only sort of lists that may contain Proc_Rel relocation. These lists are placed immediately after the body of the procedure they describe. The start or high end address of each list is pointed at by the segment relative word pointer contained in the Exit_IC field of each assembler-generated procedure.

An assembler-generated segment is also unique in that during the linking process, the code bodies of all its procedures and functions may be copied into one of the segments of the compilation unit to which it is being bound. Further, the name of the segment or segments that the assembly code may be linked to is never known at assembly time. It is, however, always assumed that any number of assembly procedures or functions that communicate via REFs and DEFs are always bound into the same segment, regardless of whether they were assembled together.

The Data_Size word generated by the assembler for each routine should have a value of -1 (0FFFF HEX); this indicates a data size of zero that is one's complemented, to signal that the first instruction of the code body is native code.

Finally, since the assembler-generated code segments can't know what program or unit they are to be linked to, the Prog_Name entry in the Seg_Famly array of the segment dictionary should be blank-filled, and the Data_Seg_Num field in the List_Header record of all Base_Rel relocation sublists should be zero-filled.

It is the linker's responsibility, when linking assembler-generated segments, to convert all Proc_Rel relocation sublists into Seg_Rel relocation lists, to correctly set the Data_Seg_Num field in the List_Header of all Base_Rel relocation sublists, and to collect all relocation sublists and place them after the procedure dictionary of the code segment. The linker should also update the Relocatable bit in the Seg_Misc array, depending on the information supplied in linker information.

p-machine

p-machine

CHAPTER 3

THE P-MACHINE

OVERVIEW

The p-machine is an idealized machine. Compiled user programs, system programs such as the filer, and the operating system itself run on the p-machine. Code for the p-machine is known as p-code, and all code files in the system consist of either p-code or native code (that is, code for a particular physical processor).

P-code is designed to be compact, so that programs in p-code are much shorter than equivalent programs in native code. P-code is also designed to be easily generated by a compiler.

Because p-code is compact and simple, relative to native codes, it's fairly easy to implement the p-machine on a variety of actual processors. It is also easier (and cheaper) to maintain a system that runs on one p-machine, rather than a family of systems, each dedicated to a particular physical processor. This is the key to the portability of the p-System.

Emulative Execution

The "p" in "p-code" and "p-machine" stands for "pseudo." The p-machine emulator program is written in the native code of a particular processor. It is responsible for executing p-code instructions and controlling machine-dependent I/O. The p-machine emulator is also called the PME (or the interpreter).

At run-time, the user's program (or a portion of it) is in main memory. The PME fetches each p-code instruction, in sequence, and performs the appropriate action.

The process of bootstrapping involves loading the PME (if necessary) and starting its execution. (The next step is to call the operating system, which runs on the p-machine).

The Stack and the Heap

The system maintains data in two dynamic structures called the stack and the heap. The stack is used for static variables, bookkeeping information about procedure and function calls, and evaluation of expressions. The heap is used for dynamic variables, including the structures that describe a program's environment. It is also used to store private stacks for subsidiary processes and to store code segments that are position-locked.

The stack is an integral part of the p-machine architecture. Most p-code instructions affect the stack in one way or another.

The heap is an integral part of the system, but is primarily supported by the operating system, rather than the p-machine.

Both the stack and the heap reside in main memory, and grow toward each other in a (largely) first-in-first-out manner. Between them is an area of memory that is partly unused, but may contain the code pool (see below).

The heap is more fully described in Chapter 5, "The Operating System."

Code Segments

In the p-System, program code is stored in one or more segments. A code segment may contain either p-code or native code (or both). Besides the code itself, each code segment contains bookkeeping information for the system's use, and (usually) a pool of constants.

Every "compilation unit" (a separately compiled Pascal PROGRAM or UNIT) results in a "principal segment" of code. In addition, there may be "subsidiary segments," if the program or unit contains SEGMENT routines. Information embedded in the compilation's code file contains the references among the (possibly) various compilation units that are part of the full program.

When a program is X(ecuted, the operating system reads this reference information and resolves the references by finding the location of all compilation units needed by the program (including subsidiary segments and indirect references, such as a UNIT using another UNIT). Tables are built that may be used at run-time to make references (such as procedure calls) from one segment to another.

The segments of a running program compete for space in main memory with each other. If the code pool is internal (between the stack and heap), then the segments also compete with the stack and the heap. The principal constraint (as far as code segments are concerned) is that both the calling and called segment must be present in main memory for an intersegment call to succeed.

Segments in main memory are stored contiguously in an area called the code pool. On nonextended memory systems, the code pool resides between the stack and the heap (an internal code pool). On extended memory systems, the code pool resides outside of the stack/heap area (an external code pool). The code segments may be moved about within the code pool or discarded in order to create more room.

Code segments are described in this chapter. Code pool handling is described in Chapter 5, "The Operating System."

Device I/O

Device I/O and control is accomplished by calls from the language level to routines within the PME. The device I/O routines then call on the routines of the PME's BIOS (for Basic I/O Subsystem), and the BIOS routines control the peripheral hardware directly. I/O environment dependencies are, thus, isolated in the BIOS, and it is possible to adapt the p-System to a new hardware environment by changing only the BIOS (not the entire PME).

On adaptable systems, the BIOS itself has a standard interface to the SBIOS, or Simplified BIOS. The SBIOS is a set of simple I/O routines, and is intended to allow the user to rapidly adapt the system to a new I/O environment.

The BIOS is dealt with in Chapter 4, "Low-Level I/O." The SBIOS is described in the Adaptable System Installation Manual.

CODE SEGMENT ENVIRONMENTS

Segment Information Blocks (SIBs)

A Segment Information Block (SIB) is a record that contains information about an "active" code segment. A code segment is active if it may be used by a program that is running. A SIB is allocated on the heap, and remains there as long as the segment is active. There is only one SIB for each code segment, no matter how many other segments may be using it.

A code segment need not be in memory to be active; an active code segment may be on disk or in the code pool, but its SIB will always be on the heap. One exception to this is that some operating system segments may be allocated outside of the stack/heap space by the bootstrap.

The following fragment of Pascal code describes a SIB:

```
SIB = RECORD
        Seg_Pool: Pool_Ptr;   {points to the description of the code pool
                               where the segment resides}
        Seg_Base: Mem_Ptr;    {byte offset within code pool of segment's
                               memory location}
        Seg_Refs: integer;    {# of active calls to the seg}
        Time_Stamp: integer;  {memory swap activity}
        Link_Count: integer;  {number of links to the SIB}
        Residency: -1..maxint;{-1 = pos lock, 0 = swap, n = mem lock}
        Seg_Name: PACKED ARRAY [0..7] OF CHAR;
        Seg_Leng: integer;    {# of words in segment}
        Seg_Addr: integer;    {disk address of segment}
        Vol_Info: VIP;        {pointer to disk drive info}
        Data_Size: integer;   {number of words in data segment}
        Res_SIBs: RECORD      {code pool management record}
                    Next_SIB,         {next SIB in list}
                    Prev_SIB: SIB_P; {previous SIB in list}
                    CASE boolean OF  {scratch area}
                      TRUE:  (Next_Sort: SIB_P);  {next SIB in sort list}
                      FALSE: (New_Loc:  Mem_Ptr);     {temporary address}
                  END {of Res_SIBs};
        M_Type: integer;
    END (of SIB);
```

Seg_Pool points to the description of the code pool where the segment resides. If the segment is on the heap (or if there is no external code pool), this value is set to NIL. (A Pool_Ptr is declared as a ^Pool_Des.)

Seg_Base contains the byte offset within the code pool of the code segment. If there is no external code pool, Seg_Base contains the address of the segment within the stack/heap area. If the code segment isn't in memory, Seg_Base contains NIL.

Seg_Refs contains the number of outstanding calls to the segment. It is incremented whenever a routine outside the segment executes an external call to a routine within the segment. It is decremented whenever a routine within the segment returns to a routine outside the segment.

Time_Stamp contains a value which is used by the operating system to determine which segment(s) should be removed from the code pool when more space is required. The Time_Stamp field indicates which segment is least recently used. The SYSCOM area, within the operating system's KERNEL, contains a 16-bit variable which is incremented every time a segment is exited. This value is placed into the Time_Stamp field of the SIB for the segment being exited.

Link_Count contains the number of links to the SIB from other operating system data structures. When Link_Count becomes zero, the SIB is removed from the heap (the space it occupied is available again).

Residency contains a value between -1 and maxint. A -1 indicates that the segment is position-locked (this occurs when the Boolean Relocatable in the segment dictionary is FALSE). A zero indicates that the segment is wwappable (that is, it can be removed from memory if necessary). A value greater than zero indicates that the segment is memory-locked. In this case, the value is a count of the number of memory lock operations that have been applied to that segment. Residency is incremented when a program declares the segment to be memory-locked, and decremented when a program declares it to be swappable. It becomes actually swappable when residency is equal to zero (that is, when no outstanding memory-lock operations remain). Programs can control the residency of segments by using the intrinsics MEMLOCK and MEMSWAP.

Seg_Name contains the first eight characters of the segment's name.

Seg_Leng contains the number of words that the code segment occupies (including any relocation lists, but excluding segment reference lists).

Seg_Addr contains the segment's first block number on disk.

Vol_Info contains a pointer to a volume information record that contains the drive number and volume name of the disk on which the segment is resident.

Data_Size contains the number of words in the code segment's data segment. This only applies to principal segments; otherwise, Data_Size should be zero.

Res_SIBs is used to maintain the code pool. All SIBs of segments in the code pool are on a doubly-linked list formed by the Prev_SIB and Next_SIB pointers. The Sort_SIB and New_Loc fields are used for temporary values while managing the code pool.

The operating system uses several data structures to manage code segments by maintaining active SIBs and managing the code pool. All of these data structures refer to SIBs through pointers.

When a program being prepared for execution requires a code segment that isn't yet active, the appropriate SIB is allocated on the heap and initialized. The operating system creates a pointer to the SIB, and the SIB's Link_Count is incremented. When the segment is no longer needed, the pointer is removed, and the Link_Count is decremented. When Link_Count becomes zero, the SIB is removed from the heap.

Environment Records (E_Recs)

A code segment's "environment" is a mapping of segments it may access to local segment numbers. Segment numbers have local meaning; a segment may refer only to segments that have been assigned local segment numbers—not to segments outside its scope.

For each segment, there is an Environment Record (E_Rec). This record designates an Environment Vector (E_Vect) that describes the mapping of local segment numbers to actual code segments.

The following fragment of pseudo-Pascal describes environment records and vectors:

```
E_Vect_P = ^E_Vect;
E_Rec_P  = ^E_Rec;

E_Vect   = RECORD
             Vec_Length: integer;   {number of local segments}
             Map: ARRAY [1..Vec_Length] OF E_Rec_P;
                                    {local environment mapping}
           END {of E_Vect};

E_Rec    = RECORD
             Env_Data: Mem_Ptr;       {pointer to global data}
             Env_Vect: E_Vect_P;      {pointer to environment}
             Env_SIB:  SIB_P;         {pointer to SIB for seg number}
             CASE boolean OF
               TRUE : (Link_Count: integer;  {number of links to E_Rec}
                       Next_Rec:   E_Rec_P); {next environment record}
           END {of E_Rec};
```

Env_Data points to the segment's global data. (The data is allocated on the heap when the program is executed.)

Env_Vect is an array of pointers to E_Rec's. It is indexed by a segment number—the pointer indicates an E_Rec that describes a code segment. In this way, a mapping from local segment numbers to actual segments is accomplished.

Env_SIB points to the segment's SIB. (It is also placed on the heap when the program is called.)

Link_Count indicates the number of active compilation units that are currently USEing the segment. This only applies to the principal E_Rec of a compilation unit. Link_Count is maintained in the same way a SIB's Link_Count is maintained.

Next_Rec is a pointer on a chain of all active compilation units. This chain is called Unit_List. This field also applies only to the principal E_Rec's of a compilation unit.

In order to minimize index manipulations, the Map array in an E_Vect record starts at 1. Thus, it may be indexed by local segment numbers (these must be 1 or greater). The Vec_Length field of the record may be considered to occupy the zero'th position of the map.

The operating system uses a recursive routine to construct the environments of a program's USEed units, and then its subsidiary segments and principal segment (its "native segments"). The algorithm is roughly:

```
FUNCTION Build_Env (Seg_Dict): E_Rec_P;
  BEGIN
  IF outer block segment E_Rec exists in Unit_List
  THEN
    BEGIN
      increment Link_Count;
      return existing E_Rec_P
    END
  ELSE
    BEGIN
     create E_Vect;
     create Env_Data for outer block data space;
     IF there are USEd units indicated in Seg_Dict THEN
       FOR all USEed units DO
          install Build_Env (New_Seg_Dict) into current E_Vect;
     FOR all native segments DO
     BEGIN
       create E_Rec and SIB for native segment;
       install E_Vect, SIB, and Env_Data in E_Rec;
       install E_Rec for native segments in E_Vect
     END;
     install E_Rec for outer block segment on Unit_List;
     return E_Rec_P for outer block segment
    END
  END
```

The Build_Env function returns a pointer to the E_Rec for the outer block of the program being executed. This pointer is installed into the operating system's User_Program E_Vect entry.

After a program's execution, a recursive routine is used to delink the environment for the program's outer block and all subsidiary units and segments. The algorithm is roughly:

```
PROCEDURE Dump_Env (E_Rec_P);
  BEGIN
  decrement Link_Count;
  IF Link_Count = 0 THEN
  BEGIN
    de_link from Unit_List;
    DISPOSE (Env_Data);
    FOR all E_Rec's on E_Vect whose
      Seg_Vect <> E_Rec.Seg_Vect DO
      Dump_Env (those E_Rec's);
    FOR all E_Rec's on E_Vect whose
      Seg_Vect = E_Rec.Sec_Vect DO
    BEGIN
      de_link E_REC^.SEG_SIB;
      DISPOSE (those E_RECs);
    END;
    DISPOSE (E_Rec.Seg_Vect);
  END
  END
```

The operating system sets its E_Vect entry for the terminating program to NIL, and calls Dump_Env for the outer block's E_Rec. After Dump_Env returns, a pass is made through the Res_SIBs list to find all segments whose Link_Count = 1, and remove them from the heap.

TASK ENVIRONMENTS

A task is a routine that is executed concurrently with other routines. A task is implemented by three data structures: the body, the Task Information Block (TIB), and the task queue. In Pascal, a task is known as a PROCESS.

The "main task" of the p-System is the thread of execution that runs from operating system initialization and all system utility or user program executions to the termination of the operating system. A program may have subsidiary tasks.

During execution, each subsidiary task uses its own stack instead of the system stack. The task's activation record (described later in this chapter) is actually contained in the task stack; both are allocated on the heap, along with an amount of free space into which the stack may grow.

The task body is a portion of a p-code segment. In structure, it's no different from the body of a procedure or function.

The amount of space allocated to the task stack depends on the Stack_Size parameter of the START intrinsic. The default is 200 words.

The main task uses the system stack for expression evaluation and activation records. The heap is shared by the main task and all subsidiary tasks.

The TIB of a subsidiary task is allocated on the heap when the task is started. It contains information about a task's execution environment. This must be maintained, and restored whenever a task is restarted after having been idle.

At any given time, the p-machine may have:

one task running;
several tasks ready to run; and
several tasks waiting for semaphores.

The tasks that are ready to run are organized into a queue. There is also a queue of tasks waiting for each semaphore (the queue may be empty). Tasks in queues are ordered by their priority.

The p-machine register CURTSK always points to the TIB of the currently executing task. The register READYQ points to the first in the list of tasks ready to run.

The following fragment of Pascal code describes a TIB:

```
TIB = RECORD  {Task Information Block}
       Regs: PACKED RECORD
               Wait_Q: TIB_Ptr;
               Prior: byte;
               Flags: byte;
               SP_Low: Mem_Ptr;
               SP_Upr: Mem_Ptr;
               SP: Mem_Ptr;
               MP: MSCW_Ptr;
               Reserved:Integer
               IPC: integer;
               Env: ERec_Ptr;
               Proc_Num: byte;
               TI_BIOResult: byte;
               Hang_Ptr: Sem_Ptr;
               M_Depend: integer;
             END {of Regs}
       Main_Task: Boolean;
       Start_MSCW: MSCW_Ptr;
     END {of TIB}
```

SP is the p-machine stack pointer. SP_Low and SP_Upr are the lower and upper bounds for this task.

MP designates the local activation record for this task.

IPC is the p-code instruction counter (a segment relative byte pointer), and Proc_Num is the number of the executing routine.

Priority contains the task's priority. This is a number from 0 through 255. The higher the value, the greater the priority.

Wait_Q is used when the task is waiting to run, or waiting for a semaphore. Wait_Q is one link in a linked list of TIBs.

When a task is waiting for a semaphore, Hang_Ptr points to that semaphore. If the task isn't waiting for a semaphore, Hang_Ptr is NIL. Hang_Ptr allows a task to be removed from a semaphore's wait queue if the task is being terminated.

Flags are reserved for future use.

Env is a pointer to the task's current E_Rec. The task's current SIB may be found through the E_Rec.

TIBIO_Result is used to save an IORESULT that is local to the task.

M_Depend contains machine-dependent data maintained by the PME. It is initialized to 0.

Main_Task, if TRUE, indicates that this is the TIB of a "root" ("parent") task.

Start_MSCW points to the MSCW (mark stack control word) of the routine that STARTed this task.

Further information about tasks appears below in Chapter 5, "The Operating System." Figure 3-1 shows the layout of main memory while the system is running, including the location of task stacks as discussed in this section.

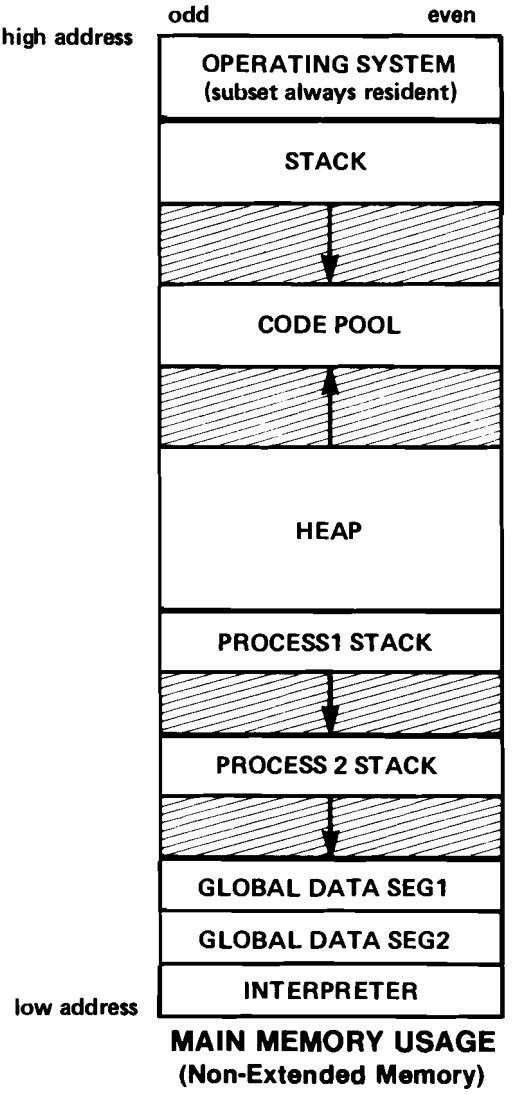


Figure 3-1. Main Memory Usuage

P-MACHINE INSTRUCTIONS

The Intrinsic P_MACHINE

A Pascal compilation unit may directly generate in-line p-code. This is done by calling the intrinsic procedure P_MACHINE. Producing in-line p-code may be useful in very low-level system programming. **Absolutely no protection** is provided by this intrinsic or the system; it can only be used at the user's risk, and extreme caution should be exercised.

The form of a call to P_MACHINE may be sketched as follows:

```
P_MACHINE ( <p-machine item> {, <p-machine item>} )
```

that is, the parameters to the procedure are a list of one or more <p-machine item>s. A <p-machine item> describes a portion of p-code, and causes one or more bytes to be generated.

There are three varieties of <p-machine item>:

1. P-code syllable: The simplest item is a (non-real) scalar constant. This item produces a single byte of p-code, which is the least significant byte of the specified constant.

2. Expression value: If the item is an expression enclosed in parentheses, then a p-code sequence is generated which will compute the value of the expression and leave it on the stack.

3. Address reference: If the first token of the item is '^', then the item is the specification of a variable, and p-code is generated which leaves the address of that variable on the stack.

A <p-machine item> may not be a string constant.

EXAMPLE:

Given these declarations:

```
CONST STO = 196;

TYPE Records  = RECORD
                  First_Field, Second_Field: integer
                END;
     P_Records = ^Records;

VAR Vector: ARRAY [0..9] OF P_Record;
    i: integer;
```

... the following call to P_MACHINE ...

```
P_MACHINE ( ^Vector[5]^.First_Field, (i*i), STO)
```

would cause the square of "i" to be stored in the first field of the record designated by the sixth element of the array Vector.

P-MACHINE REGISTERS

Like other processors, the p-machine has registers which are a fundamental part of its architecture. Since the p-machine is usually emulated by a program on a host processor, these registers may or may not correspond to actual host processor registers.

Unlike most processors, the p-machine doesn't allow its registers to be used in a general fashion. All registers have specific uses. The p-machine stack takes the place of general purpose registers; all temporary data is stored there.

Here is a list of the p-machine registers, along with a description of how they are used. All the registers listed below are required registers that will be found on each p-machine. They are the registers that may be accessed via the LPR (Load Processor Register) and SPR (Store Processor Register) instructions. Some p-machines may have additional registers for optimization. These will be mentioned below where appropriate.

CURPROC

The CURPROC register contains the procedure number of the currently executing procedure. It changes whenever a procedure call is made. There is a maximum of 255 procedures per segment, so CURPROC will have a value in the range 1 through 255.

CURTASK

The CURTASK register is a pointer to the TIB of the currently executing task. It changes whenever a task switch occurs.

EREC

The EREC register is a pointer to the E_Rec (Environment Record) of the current environment. It changes whenever a call or return is made to a procedure in a different segment. The E_Rec contains pointers to the global data, EVEC and SIB. Often the pointer to the global data is kept as a register called BASE. Also, a pointer to the SIB may be kept as a register called SIB, and the location in memory of the current segment (found in the SIB) may be kept as a register called CURSEG. If BASE, SIB, or CURSEG are kept as auxiliary p-machine registers, they must be updated whenever EREC is changed.

EVEC

The EVEC register is a pointer to the E_Vect (Environment Vector) of the current environment. It changes whenever a call or return is made to a procedure in a different segment. The EVEC is a redundant register, because the E_Vect may be found through the EREC register. The EVEC register is used to find the E_Rec of a different segment in order to access its data or to call a procedure in that segment.

IORESULT

IORESULT contains the code resulting from the last I/O operation. This is the only register that may be accessed directly (without a Load Processor Register instruction) by the operating system since it is located in SYSCOM. IORESULT changes whenever it is modified by the operating system, or whenever an I/O operation occurs. IORESULT is limited in size to the range 0 through 255.

IPC

The IPC register is a pointer to the next p-code that will be executed relative to the currently executing segment. It changes during each p-code execution.

MP

The MP register points to the current activation record (MSCW). It changes whenever a procedure call or return is made. All variables (except those that have been dynamically allocated on the heap) are accessed from an MSCW. Local variables are accessed from MP, global variables from BASE (see EREC, above), and intermediate variables from an intermediate MSCW.

READYQ

The READYQ register points to the TIB at the head of the queue of tasks ready to be run. It may change on a SIGNAL or WAIT p-code, or when an attached semaphore is signalled.

SP

The SP register points to the word that is on the top of the p-machine stack. It changes on nearly every p-code, whenever an item is pushed on or popped off the stack.

FAULTS

A fault is a special condition recognized by the PME during execution of a p-code that requires operating system assistance to fix. After handling the problem, the operating system returns to p-code execution where the fault was detected. The p-code where the fault was detected is reexecuted.

Two types of faults may be issued by the PME: segment faults and stack faults. A segment fault is issued when a segment that must be accessed isn't in memory. A stack fault is issued if there isn't enough room on the stack for a p-code to perform its operation. Stack height checking is done only on p-codes that will place multiple words on the stack, except in the case of real number operations, which do no stack checking.

When the fault is detected, the p-machine must be returned to the state it was in prior to execution of the p-code. This is so that the p-code may be reexecuted on return from the fault.

The fault is issued by saving information in SYSCOM, then signalling the semaphore Real_Sem in SYSCOM. This signal starts the high priority task Fault_Handler in the KERNEL, which processes the fault.

The following fields in SYSCOM with the indicated byte offsets are used in handling faults:

```
14   Real_Sem      semaphore to start the faulthandler
18   Fault_TIB     TIB of faulting task
20   Fault_EREC    E_REC of segment to read
22   Fault_Words   number of words needed on stack
24   Fault_Type    80H=segment fault, 81H=stack fault
```

A stack fault sets Fault_EREC to the current p-machine EREC. A segment fault sets Fault_Words to zero. All other parameters are set up as described above.

The following p-codes may issue a segment fault:

CAP, CSP, CXL, SCXGn, CXG, CXI, CFP, RPU, SIGNAL (if a task switch occurs), WAIT (if a task switch occurs)

The following p-codes may issue a stack fault:

LDC, LDM, ADJ, SRS, CLP, CGP, SCIPn, CIP, CXL, SCXGn, CXG, CXI, CFP

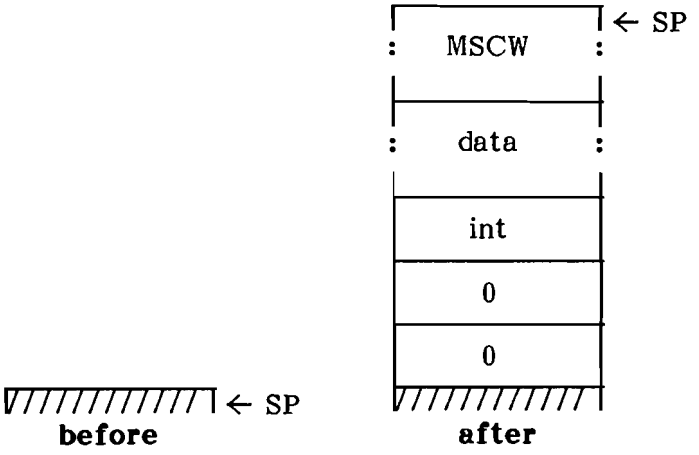
EXECUTION ERRORS

An execution error is a special error condition that may be recognized during execution of a p-code. When an execution error is detected, the PME calls the operating system routine Exec_Error to report the error. On the call, no stack checking should be done in order to prevent a stack fault.

Under normal circumstances, Exec_Error won't return and continue p-code execution where the execution error was detected. Instead, the system will be reinitialized. However, it is possible for you to specify that execution should continue. Thus it is highly desirable that each p-code that can cause an execution error leave the p-machine in a consistent state on detection of the error. Usually it is best to leave IPC pointing to the next p-code, putting "dummy" results on the stack; that way the p-code won't be reexecuted on return.

The call to Exec_Error is made by performing a CXG 1,2 (an external call to KERNEL, procedure 2) with the stack as follows:

Stack:



TOS and TOS-1 are the usual elements placed on the stack for a procedure call. TOS-2 is the execution error number. TOS-3 and TOS-4 are dummy parameters. (TOS indicates the element at the top of the stack. TOS-1 indicates the second element from the top, and so forth.)

Below is a list of the execution errors, along with the execution error number, the p-codes that may issue the error and a description of what the error means.

Value Range Error

Number: 1

P-Codes: CHK, CSTR

Description:

A value range error is issued if an array index or scalar is out of bounds. This is detected only with one of the special check instructions.

No Proc in Seg Table

Number: 2

P-Codes: CLP, CGP, SCIPn, CIP, CXL, SCXGn, CXG, CXI, CFP

Description:

A no-proc-in-seg-table error is issued whenever a call is made to a procedure whose procedure dictionary entry is zero. This condition indicates that the procedure hasn't been linked into the host program.

Integer Overflow

Number: 5

P-Codes: <long integer routines>

Description:

An integer overflow error is issued on a conversion from long integer to integer, where the resulting integer is too large to fit into 16-bits. It will also be generated if a long integer on the stack is too large to be stored.

Divide by Zero

Number: 6

P-Codes: DVI, MODI, DVR, <long integer routines>

Description:

A divide-by-zero error is issued whenever division or the remainder function is attempted with a zero denominator.

Program Interrupted by User

Number: 8

P-Codes: <none>

Description:

A program-interrupted-by-user error is issued whenever the BIOS informs the PME that the break key has been pressed and break hasn't been disabled.

I/O Error

Number: 10

P-Codes: <IOCHECK>

Description:

An I/O error is issued on the IOCHECK standard procedure when IORESULT is nonzero.

Unimplemented Instruction

Number: 11

P-Codes: <any unimplemented p-code>

Description:

An unimplemented-instruction error is issued if an attempt is made to execute an illegal or reserved p-code.

Floating Point Error

Number: 12

P-Codes: LDCRL, LDRL, STRL, FLT, TNC, RND, ABR, NGR, ADR, SBR, MPR, DVR, EQREAL, LEREAL, GEREAL, <POWEROFTEN>

Description:

A floating point error is issued if the result of a floating point calculation isn't a legal floating point number. This may happen on floating point overflow. This error is also issued if a floating point p-code is executed with a PME that doesn't support floating point.

String Overflow

Number: 13

P-Codes: CSP, ASTR, <long integer routines>

Description:

A string overflow error is issued on string assignment to a string that is too small to hold the source string.

Break Point

Number: 16

P-Codes: BPT

Description:

A break point error is issued when a break point p-code is executed. This error will result in entering the Debugger if the Debugger is currently running.

Set Too Large

Number: 18

P-Codes: SRS

Description:

A set-too-large error is issued if an attempt is made to create a set that is larger than the largest allowed set size.

Segment Too Large

Number: 19

P-Codes: <READSEG>

Description:

A segment-too-large error is issued if an attempt is made to read, with the standard procedure READSEG, a segment that is too large. This execution error is generated only by p-machines that have an unusual restriction on segment size.

P-CODE INSTRUCTIONS

Instructions for the p-machine consist of an opcode, which is one or two bytes long, followed by zero to three parameters. There are 217 p-code instructions (255 minus 38 currently unused instructions). P-machine instructions are described at the end of this section. They are also listed alphabetically in Appendix A and numerically in Appendix B.

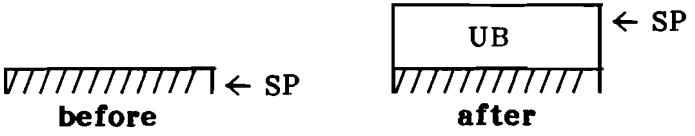
Here is a description of a typical p-code instruction. (The format of the description is the same for all p-codes.)

LDCB Load Constant Byte

Opcode: 128 80

Operation: LDCB UB

Stack:



Description:

The constant UB with high byte zero is pushed onto the stack. LDCB is used to load a constant in the range 0 through 255.

The top line shows the mnemonic of the p-code (LCDB) and its name (Load Constant Byte); the second line shows its value, both decimal (128) and hexadecimal (80).

The third line defines the operation of the instruction. The mnemonic is followed by the instruction's parameter--more specifically, the format of the instruction's parameter. Here the format is UB, meaning unsigned byte. UB and the four other paramter formats are discussed below.

NOTE: Most p-machine instructions don't have specific parameters but instead deal with operands that are on the stack.

The stack part of the description shows the contents of the stack before and after execution of the instruction. SP means stack pointer; it points to the top of the stack (TOS).

Instruction Parameters

The parameters to a p-code instruction contain information about the size and number of the instruction's operands. (In some cases, the parameter may be an operand itself, as in the case of LDCB, shown above.)

The five parameter formats are:

1. **UB — Unsigned Byte**

   Represents a positive integer in the range 0 through 255. When convereted to a 16-bit value, the most significant byte is zeroed.

2. **SB — Signed Byte**

   Represents a two's complement 8-bit integer in the range -128 through 127. When converted to a 16-bit two's complement value, the most significant byte is a sign extension (all bits equal bit 7 of the low byte (SB)).

3. **DB — Don't Care Byte**

   Represents a positive integer in the range 0 through 127. Bit 7 is always 0. When converted to a 16-bit value, the most significant byte is zeroed.

4. **B — Big**

   This is a parameter with variable length. If bit 7 of the first byte is 0, the remaining 7 bits represent a positive integer in the range 0 through 127. If bit 7 of the first byte is 1, then bit 7 is cleared; the first byte is the high-order byte of a 16-bit word, and the following byte is the low-order byte of that word. The big format may represent positive integers in the range 0 through 32767.

5. **W — Word**

   This is a 2-byte paramter. It is a 16-bit two's complement value that represents an integer in the range -32768 through 32767. The word is always represented as least-significant-byte in the code stream.

Dynamic Operands

This section describes the stack-oriented dynamic operands of p-machine instructions.

Activation Record

An activation record is created for each invocation of an active routine. Figure 3-2 illustrates an activation record.

Addr (address

A 16-bit p-machine pointer. It may be a byte address, in which case it is restricted to even values. On word-addressed processors, a pointer may be a word address.

Bool (boolean)

A 16-bit quantity treated as a logical value. If bit 15 is 0, the value is FALSE. If bit 15 is 1, the value is TRUE.

Byte-ptr (byte pointer)

A 32-bit quantity. TOS is an index into an array of bytes. TOS-1 is the word address of the base of the byte array. Two words are used in a byte-ptr so that individual bytes may be specified on word-addressed processors.

Int (integer)

A 16-bit two's complement integer.

Nil

A constant that references an invalid address. The actual value varies from processor-to-processor. (See the p-code description of LDCN for a table of NIL values for various processors.)

Offset

An offset into a code segment. This is either a word or a byte offset, depending on the natural addressing unit of the host processor.

Pack-ptr (packed array pointer)

Three words that designate a bit field within a 16-bit word. TOS is the number of the rightmost bit of the field, TOS-1 is the number of bits in the field, and TOS-2 is the address of the word.

Real

A 32-bit or 64-bit floating point quantity.

Set

A set is 0 through 255 words of bit flags, preceded by a word that contains the number of words in the set.

Word

A 16-bit quantity that may be treated in any way—as an integer, boolean, address, and so forth.

Word-block

A group of Zero or more words.

Activation Records

An activation record is created for each invocation of an active routine. Figure 3-2 illustrates an activation record.

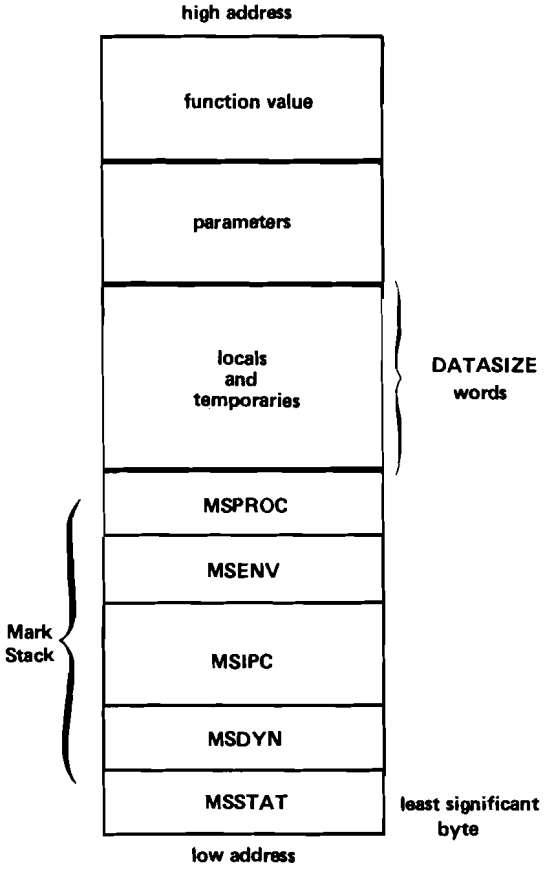


Figure 3-2. Procedure Activation Record

The parts of an activation record are:

1. Mark stack. This area contains five (full) words of housekeeping information:

   a. MSSTAT — pointer to the activation record of the lexical parent.

   b. MSDYN — point to the activation record of the caller.

   c. MSIPC — segment relative byte pointer to point of call in the caller.

   d. MSENV — E_Rec pointer of the caller.

   e. MSPROC — procedure number of caller.

2. Local and temporary variables. This area is Data_Size words long.

3. Parameters. This area (which may be empty) contains:

   a. Addresses — for VAR parameters, and record and array value parameters.

   b. Values — for other value parameters.

4. Function value. This area is present only for functions, and is either one or two words (or four words, if reals are that size).

p-code Descriptions

P-CODE DESCRIPTIONS

ABI Absolute Value Integer

Opcode: 224 E0

Operation: ABI

Stack:

before **after**

Description:

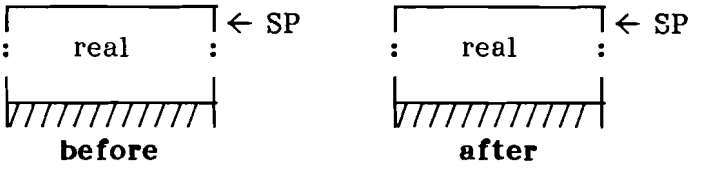
TOS is replaced by the absolute value of TOS. If TOS was initially -32768 the result should be -32768.

ABR Absolute Value Real

Opcode: 227 E3

Operation: ABR

Stack:



before **after**

Description:

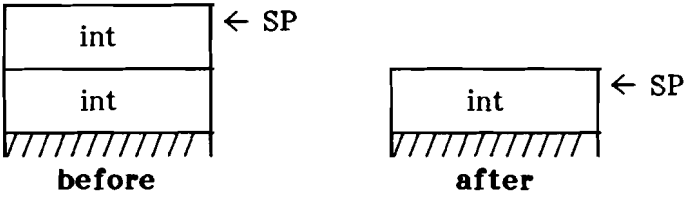
TOS is replaced by the absolute value of TOS.

ADI Add Integers

Opcode: 162 A2

Operation: ADI

Stack:



Description:

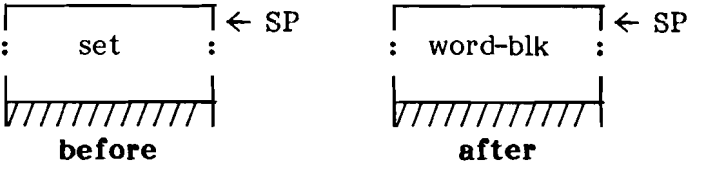
TOS is replaced by TOS-1 + TOS. The result should be computed as if it were an unsigned operation on 32-bit operands, and only the lowest 16 bits were retained for the result. Thus, overflow or underflow will "wrap around" to the opposite sign.

ADJ Adjust Set

Opcode: 199 C7

Operation: ADJ UB

Stack:



before **after**

Description:

If less than 20 words on the stack will be available after the completion of the adjust, a stack fault is issued.

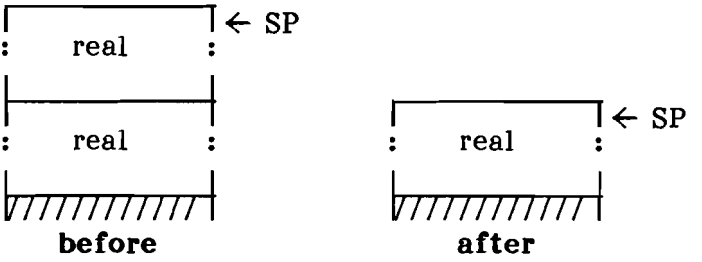
The set TOS is stripped of its length word and then expanded or compressed so that it is UB words in size. Expansion is done by adding words of zeros "between" TOS and TOS-1. Compression is done by removing high words of the set. It is legal for adjust to remove "significant" words of the set during compression.

ADR Add Real

Opcode: 192 C0

Operation: ADR

Stack:



Description:

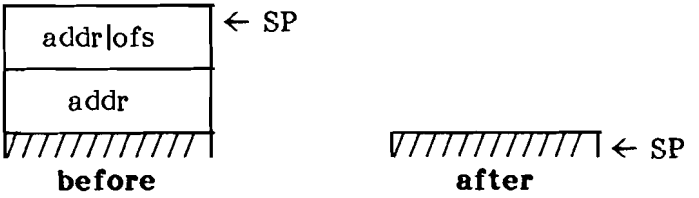
TOS is replaced by the value TOS-1 + TOS. The result should be zero on underflow. A floating point execution error is issued on overflow.

ASTR Assign String

Opcode: 235 EB

Operation: ASTR UB1, UB2

Stack:



Description:

TOS-1 is the address of the destination string variable. UB2 is the declared size of that string (the number of characters it may hold). TOS is either the address of a string variable (if UB1 is zero), or the offset of a string constant in the constant pool of the current segment.

A string overflow execution error is issued if the dynamic size of the source string is greater than the declared size of the destination string.

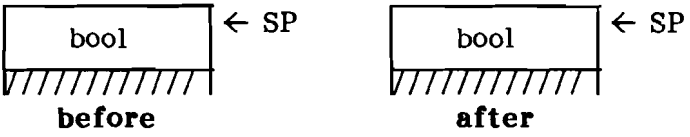
Otherwise, the source string is copied to the destination string.

BNOT Boolean Not

Opcode: 159 9F

Operation: BNOT

Stack:



before **after**

Description:

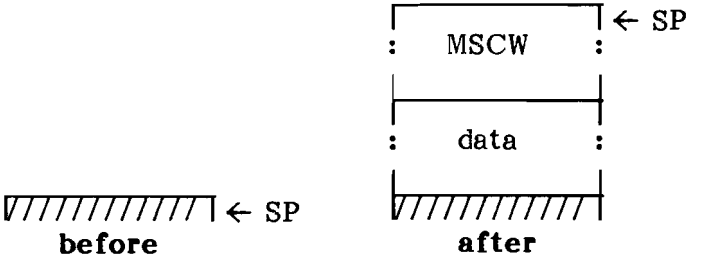
The one's complement of TOS is masked to one bit, and the result is pushed on the stack. BNOT produces a 1 (TRUE) or a 0 (FALSE) on the stack, regardless of how many bits were set in TOS.

BPT Break point

Opcode: 158 9E

Operation: BPT

Stack:



Description:

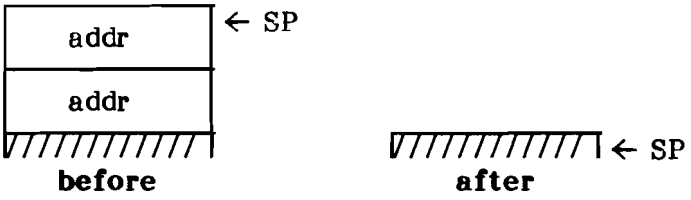
A break point execution error is issued unconditionally.

CAP Copy Array Parameter

Opcode: 171 AB

Operation: CAP B

Stack:



Description:

TOS is the address of a parameter descriptor for a packed array of characters. The parameter description is a two word record. The first (low) word is either NIL, or a pointer to an E_Rec. If the first word is NIL, the second word is the address of the paramter. If the first word points to an E_Rec, the second word is an offset relative to the segment indicated by the E_Rec. This offset was created with an LCO instruction.

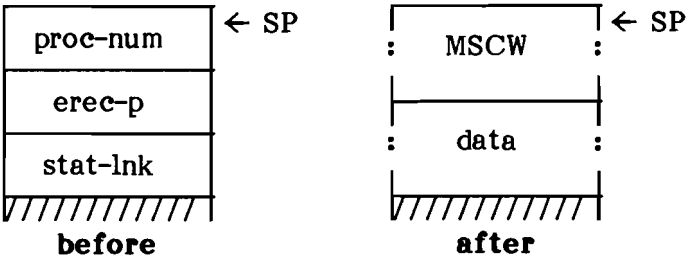
A segment fault is issued if the parameter descriptor indicates a nonresident segment. Otherwise, the array (which is B words big), is copied to the destination at address TOS-1.

CFP Call Formal Procedure

Opcode: 151 97

Operation: CFP UB

Stack:



Description:

TOS contains a procedure number. TOS-1 contains an E_Rec pointer, TOS-2 contains a static link. The procedure TOS in the segment indicated by TOS-1 is called.

If the segment indicated by TOS-1 isn't in memory, a segment fault is issued.

If the Data_Size word for procedure UB is negative, nothing is allocated on the stack and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to TOS-2. MP is set to point to the new MSCW, CURPROC is set to TOS, and IPC is set to the first p-code of procedure UB2. E_Rec and E_Vect are set to reflect the new environment.

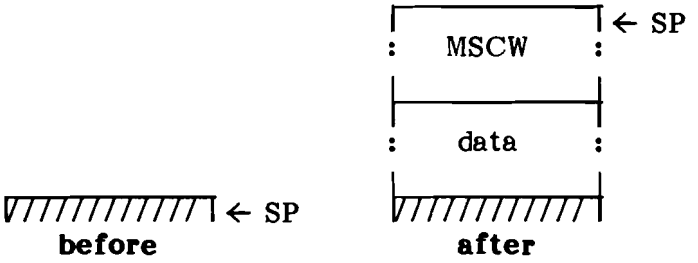
If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

CGP Call Global Procedure

Opcode: 145 91

Operation: CGP UB

Stack:



Description:

Global procedure UB in the currently executing segment is called.

If the Data_Size word for procedure UB is negative, nothing is allocated on the stack and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to the old value of BASE (the global data MSCW). MP is set to point to the new MSCW, CURPROC is set to UB, and IPC is set to the first p-code of procedure UB.

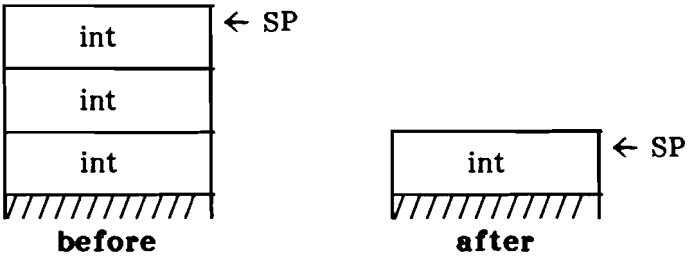
If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

CHK Check Subrange Bounds

Opcode: 203 CB

Operation: CHK

Stack:



Description:

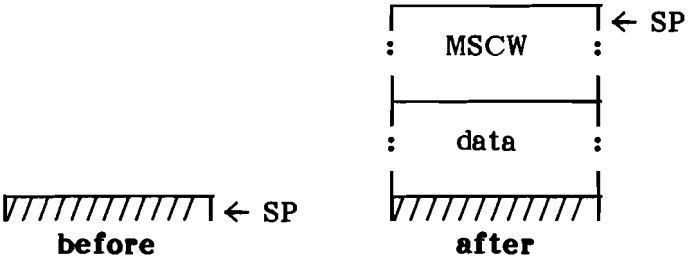
TOS is an upper-bound. TOS-1 is a lower-bound. If it isn't the case that TOS-1 <= TOS-2 <= TOS, a value range execution error is issued. TOS-2 remains on the stack.

CIP Call Intermediate Procedure

Opcode: 146 92

Operation: CIP DB, UB

Stack:



Description:

Intermediate procedure UB in the currently executing segment is called.

If the Data_Size word for procedure UB is negative, nothing is allocated on the stack and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to the intermediate MSCW that is DB lexical levels above the current MSCW. MP is set to point to the new MSCW, CURPROC is set to UB, and IPC is set to the first p-code of procedure UB.

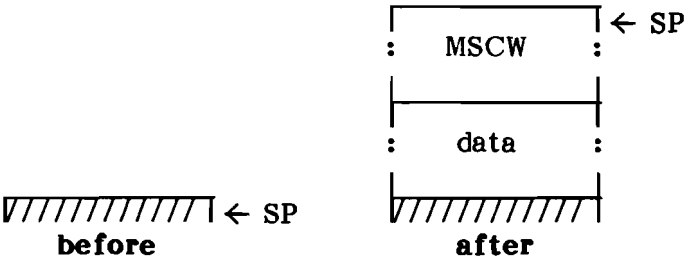
If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

CLP Call Local Procedure

Opcode: 144 90

Operation: CLP UB

Stack:



Description:

Local procedure UB in the currently executing segment is called.

If the Data_Size word for procedure UB is negative, nothing is allocated on the stack and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to the old value of MP. MP is set to point to the new MSCW, CURPROC is set to UB, and IPC is set to the first p-code of procedure UB.

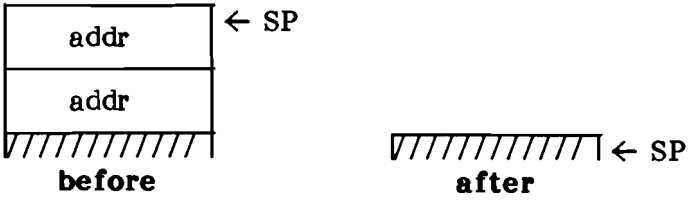
If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

CSP Copy String Parameter

Opcode: 172 AC

Operation: CSP UB

Stack:



Description:

TOS is the address of a parameter descriptor for a packed array of characters. The parameter description is a two word record. The first (low) word is either NIL, or a pointer to an E_Rec. If the first word is NIL, the second word is the address of the paramter. If the first word points to an E_Rec, the second word is an offset relative to the segment indicated by the E_Rec. This offset was created with an LCO instruction.

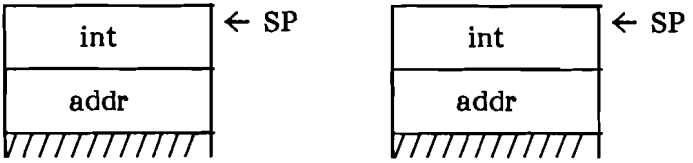
A segment fault is issued if the parameter descriptor indicates a nonresident segment. Otherwise, the dynamic length of the designated string is compared to UB (the declared size of the destination formal parameter). If the string is larger than the destination size, a string overflow execution error is issued. Otherwise, the string is copied to the address TOS-1.

CSTR Check String Index

Opcode: 236 EC

Operation: CSTR

Stack:



Description:

TOS-1 is the address of a string variable. TOS is an index into that variable.

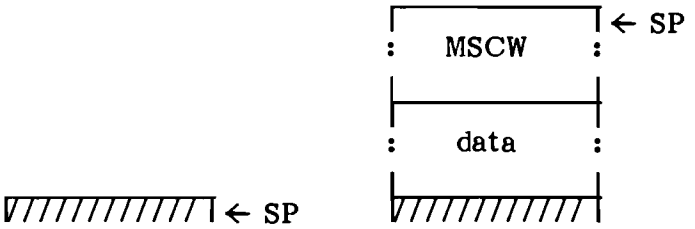
If the index is less than 1 or greater than the dynamic length of the string variable, a value range execution error is issued.

CXG Call External Global

Opcode: 148 94

Operation: CXG UB1, UB2

Stack:



Description:

The global procedure UB2 in segment UB1 is called.

If segment UB1 isn't in memory, a segment fault is issued.

If the UB1 is 1 and the procedure number matches one of the standard procedure numbers, the p-code performs the standard procedure instead of the call. See the section describing the standard procedures, at the end of this chapter.

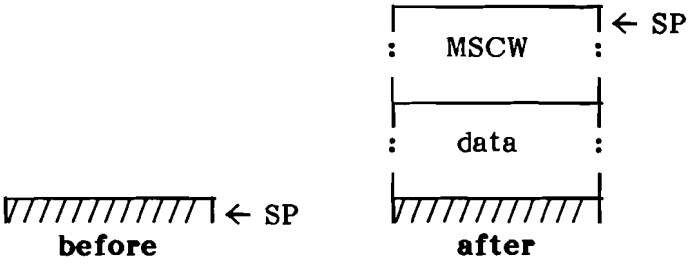
If the Data_Size word for procedure UB is negative, nothing is allocated on the stack and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to the new BASE (the global data MSCW). MP is set to point to the new MSCW, CURPROC is set to UB, and IPC is set to the first p-code of procedure UB2. E_Rec and E_Vect are set to reflect the new environment.

If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

CXI	Call External Intermediate	
Opcode:	149	95
Operation:	CXI	UB1, DB, UB2

Stack:



Description:

The global procedure UB2 in segment UB1 is called.

If segment UB1 isn't in memory, a segment fault is issued.

If the Data_Size word for procedure UB is negative, nothing is allocated on the stack and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to the MSCW, that is, DB lexical levels above the current MSCW. MP is set to point to the new MSCW, CURPROC is set to UB, and IPC is set to the first p-code of procedure UB2. E_Rec and E_Vect are set to reflect the new environment.

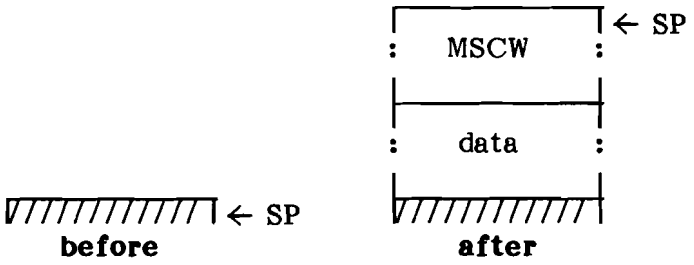
If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

CXL Call External Local

Opcode: 147 93

Operation: CXL UB1, UB2

Stack:



Description:

The local procedure UB2 in segment UB1 is called.

If segment UB1 isn't in memory, a segment fault is issued.

If the Data_Size word for procedure UB is negative, nothing is allocated on the stack and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to the old value of MP. MP is set to point to the new MSCW, CURPROC is set to UB, and IPC is set to the first p-code of procedure UB2. E_Rec and E_Vect are set to reflect the new environment.

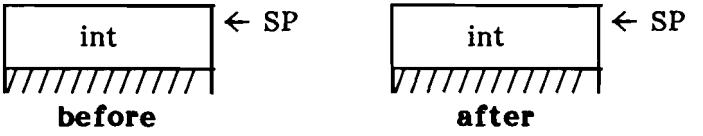
If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

DECI Decrement Integer

Opcode: 238 EE

Operation: DECI

Stack:



before **after**

Description:

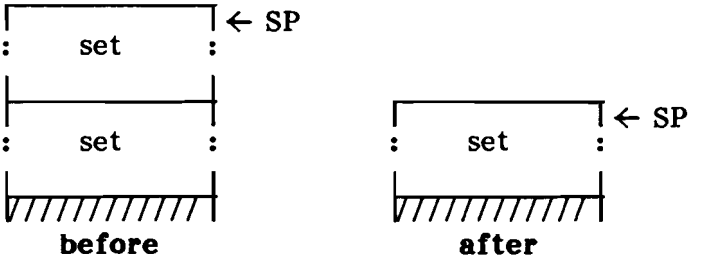
TOS is decremented by 1. If TOS was initially -32768, the result should be 32767.

DIF Set Difference

Opcode: 221 DD

Operation: DIF

Stack:



Description:

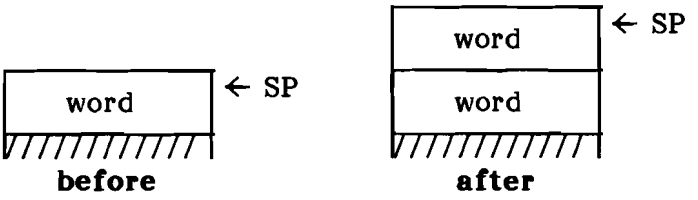
The difference of sets TOS-1 and TOS is pushed onto the stack. The difference is computed as bit-wise (TOS-1 AND NOT TOS).

DUP1 Duplicate One Word

Opcode: 226 E2

Operation: DUP1

Stack:



before **after**

Description:

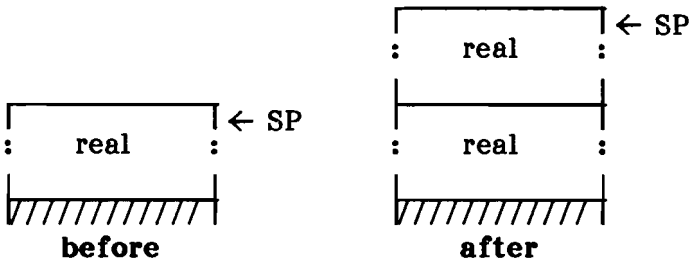
The word TOS is duplicated on top of the stack.

DUPR Duplicate Real

Opcode: 198 C6

Operation: DUPR

Stack:



Description:

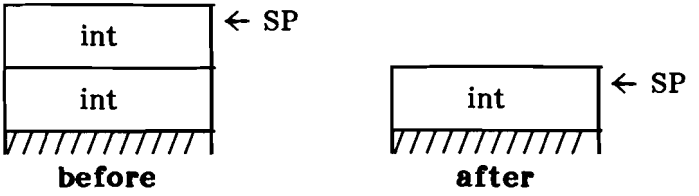
The real TOS is duplicated on top of the stack. If the p-machine supports 2-word reals, two words are duplicated. If the p-machine supports 4-word reals, four words are duplicated. If the p-machine doesn't support reals, a floating point execution error is issued.

DVI Divide Integer

Opcode: 141 8D

Operation: DVI

Stack:



Description:

If TOS is zero, a divide-by-zero execution error occurs.

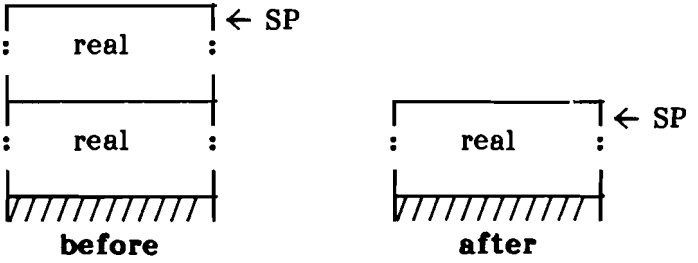
Otherwise, TOS is replaced by TOS-1 DIV TOS. The division operation is an integer division truncated toward zero.

DVR Divide Real

Opcode: 195 C3

Operation: DVR

Stack:



Description:

If TOS is zero, a divide-by-zero execution error is issued.

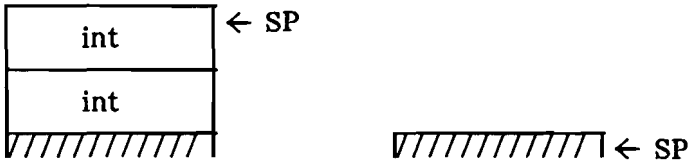
Otherwise, TOS is replaced by the value TOS-1 / TOS. The result should be zero on underflow. A floating point execution error is issued on overflow.

EFJ Equal False Jump

Opcode: 210 D2

Operation: EFJ SB

Stack:



Description:

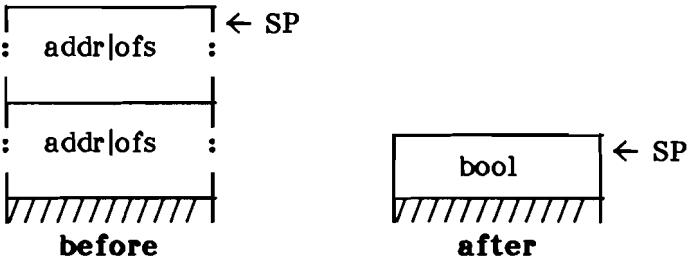
If TOS <> TOS-1, a jump is made, relative to the next instruction, by the byte offset SB.

EQBYT Equal Byte Array

Opcode: 185 B9

Operation: UB1, UB2, B

Stack:



Description:

UB1 and UB2 are mode flags. They refer to TOS and TOS-1, respectively. TOS and TOS-1 each point to a byte array (if the corresponding UB is zero) or to the offset of the byte array in the current segment. B is the size (in bytes) of the arrays.

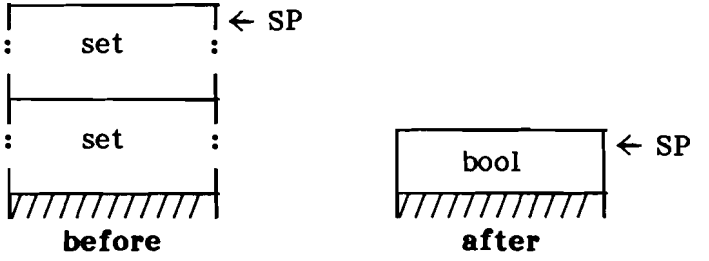
The boolean result of the comparison TOS-1 = TOS is pushed onto the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the arrays is reached. If there is a mismatch in any character position, FALSE is pushed onto the stack. Otherwise, TRUE is pushed.

EQPWR Equal Set

Opcode: 182 B6

Operation: EQPWR

Stack:



Description:

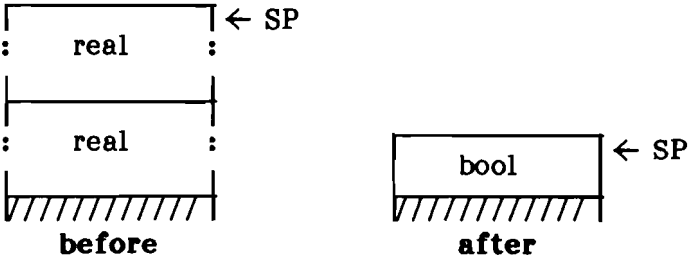
The boolean result of the comparison TOS-1 = TOS is pushed onto the stack. The sets need not be the same size—only the elements must match.

EQREAL Equal Real

Opcode: 205 CD

Operation: EQREAL

Stack:



Description:

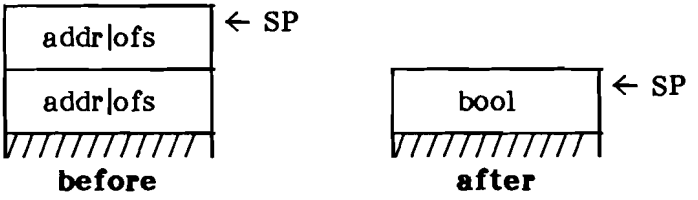
The boolean result of the comparison TOS-1 = TOS is pushed onto the stack.

EQSTR Equal String

Opcode: 232 E8

Operation: EQSTR UB1, UB2

Stack:



Description:

UB1 and UB2 are mode flags. They refer to TOS and TOS-1, respectively. TOS and TOS-1 each point to a string (if the corresponding UB is zero) or to the offset of the string in the current segment.

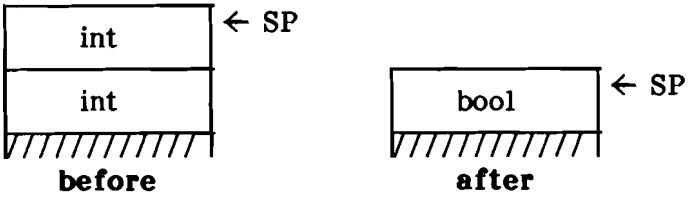
The boolean result of the comparison TOS-1 = TOS is pushed onto the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the shorter string is reached. The comparison begins at the second element of the strings. If there is a mismatch in any character position, FALSE is pushed on the stack. Otherwise, the lengths of the strings are compared, and the boolean result of the comparison length(TOS-1) = length(TOS) is pushed.

EQUI Equal Integer

Opcode: 176 B0

Operation: EQUI

Stack:



Description:

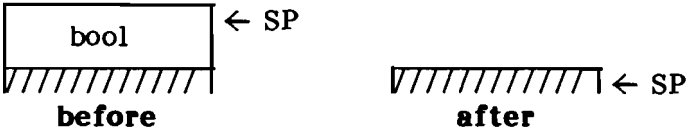
The boolean result of the comparison TOS-1 = TOS is pushed onto the stack.

FJP False Jump

Opcode: 212 D4

Operation: FJP SB

Stack:



Description:

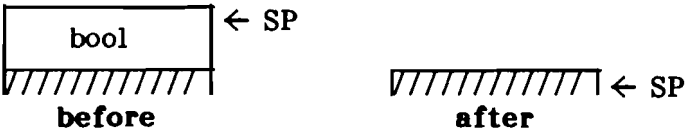
If TOS is FALSE, a jump is made, relative to the next instruction, by the byte offset SB.

FJPL False Jump Long

Opcode: 213 D5

Operation: FJPL W

Stack:



Description:

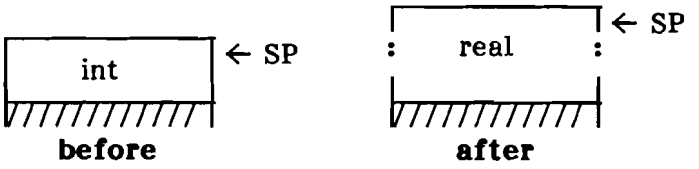
If TOS is FALSE, a jump is made, relative to the next instruction, by the byte offset W.

FLT Float

Opcode: 204 CC

Operation: FLT

Stack:



Description:

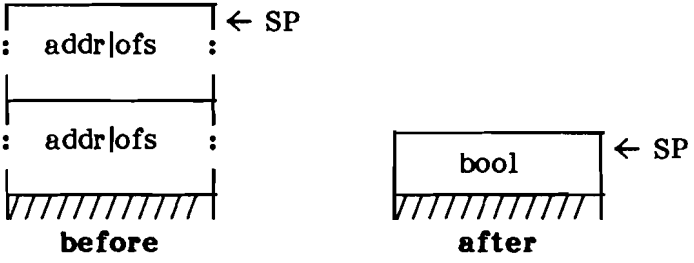
Integer TOS is converted to a floating point number, and the result is pushed onto the stack.

GEBYT Greater Than or Equal Byte Array

Opcode: 187 BB

Operation: GEBYT UB1, UB2, B

Stack:



Description:

UB1 and UB2 are mode flags. They refer to TOS and TOS-1, respectively. TOS and TOS-1 each point to a byte array (if the corresponding UB is zero) or to the offset of the byte array in the current segment. B is the size (in bytes) of the arrays.

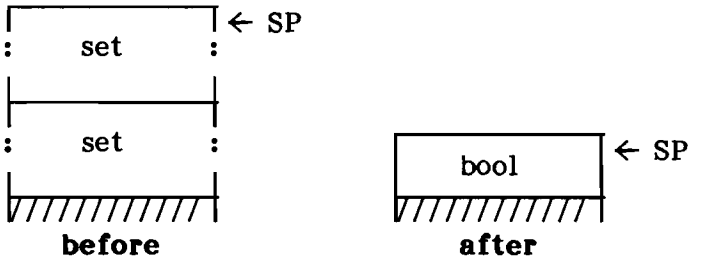
The boolean result of the comparison TOS-1 >= TOS is pushed on the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the arrays is reached. If there is a mismatch and the character in TOS-1 < the character in TOS, FALSE is pushed onto the stack. Otherwise, TRUE is pushed.

GEPWR Greater Than or Equal Set

Opcode: 184 B8

Operation: GEPWR

Stack:



Description:

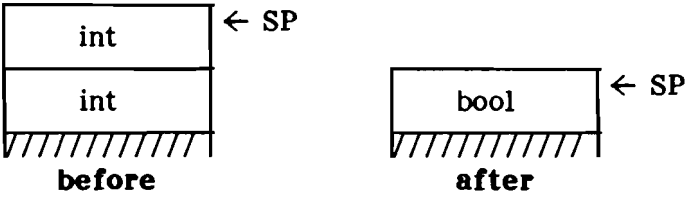
TRUE is pushed if TOS-1 is a superset of TOS. Otherwise, FALSE is pushed.

GEQI Greater Than or Equal Integer

Opcode: 179 B3

Operation: GEQI

Stack:



Description:

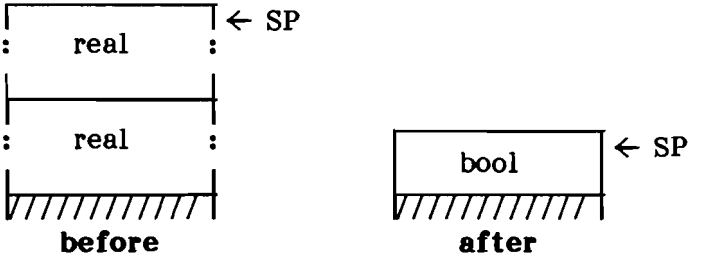
The boolean result of the signed comparison TOS-1 >= TOS is pushed onto the stack.

GEREAL Greater Than or Equal Real

Opcode: 207 CF

Operation: GEREAL

Stack:



Description:

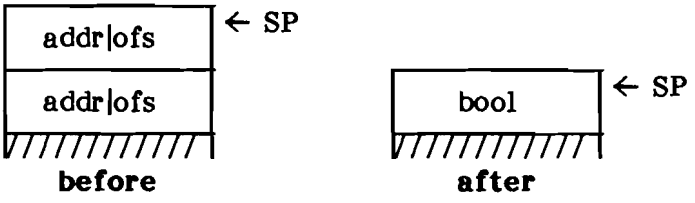
The boolean result of the comparison TOS-1 >= TOS is pushed onto the stack.

GESTR Greater Than or Equal String

Opcode: 234 EA

Operation: GESTR UB1, UB2

Stack:



Description:

UB1 and UB2 are mode flags. They refer to TOS and TOS-1, respectively. TOS and TOS-1 each point to a string (if the corresponding UB is zero) or to the offset of the string in the current segment.

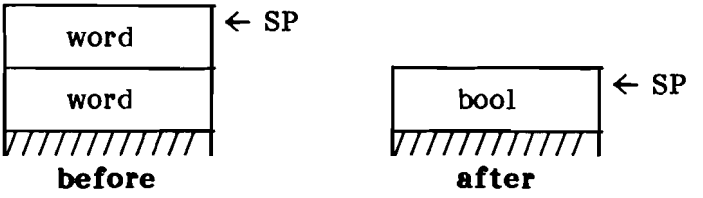
The boolean result of the comparison TOS-1 >= TOS is pushed on the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the shorter string is reached. The comparison begins at the second element of the strings. If there is a mismatch in any character position and the character in TOS-1 < the character in TOS, FALSE is pushed on the stack. Otherwise, the lengths of the strings are compared, and the boolean result of the comparison length(TOS-1) >= length(TOS) is pushed.

GEUSW Greater Than or Equal Unsigned

Opcode: 181 B5

Operation: GEUSW

Stack:



Description:

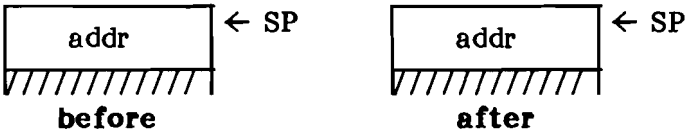
The boolean result of the unsigned comparison TOS-1 >= TOS is pushed onto the stack.

INC Increment

Opcode: 231 E7

Operation: INC B

Stack:



Description:

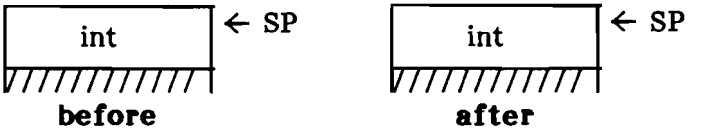
The word pointer TOS is indexed by B words, and the resulting pointer is pushed.

INCI Increment Integer

Opcode: 237 ED

Operation: INCI

Stack:



before **after**

Description:

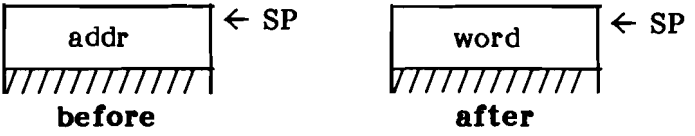
TOS is incremented by 1. If TOS was initially 32767, the result should be -32768.

IND Index

Opcode: 230 E6

Operation: IND B

Stack:



Description:

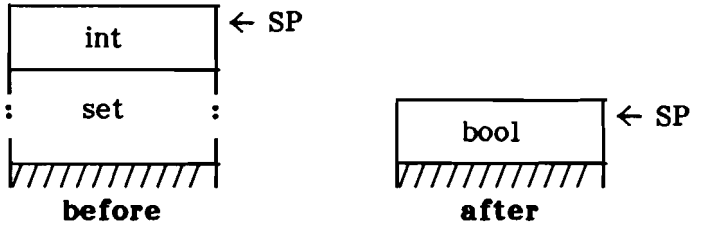
TOS is the address of a record. TOS is replaced with word B of the record.

INN Set Membership

Opcode: 218 DA

Operation: INN

Stack:



Description:

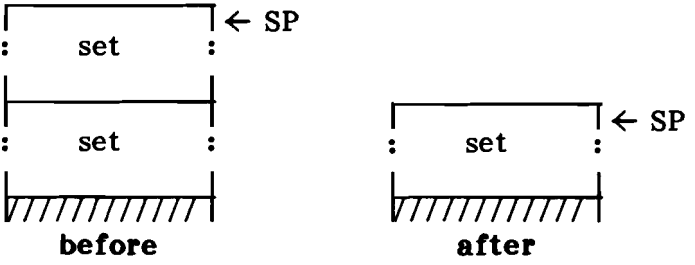
The boolean result of the check whether TOS is contained in the set TOS-1 is pushed onto the stack.

INT Set Intersection

Opcode: 220 DC

Operation: INT

Stack:



Description:

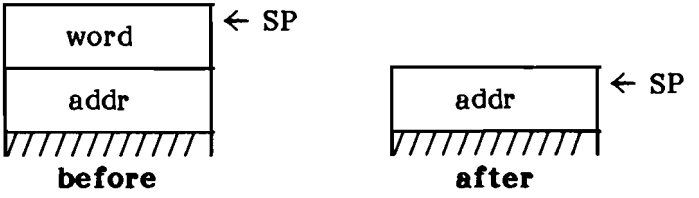
The intersection (bit-wise AND) of sets TOS and TOS-1 is pushed onto the stack.

IXA Index Array

Opcode: 215 D7

Operation: IXA B

Stack:



Description:

TOS is an integer index. TOS-1 is the array base pointer. B is the size (in words) of an array element. The word pointer to the indexed element is pushed.

Algorithm:

addr:= TOS-1 + TOS*B * x

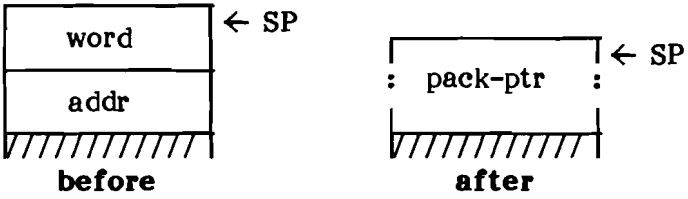
(Where x=2 for byte-addressed PMEs, and x=1 for word-addressed PMEs.)

IXP Index Packed Array

Opcode: 216 D8

Operation: IXP UB1, UB2

Stack:



Description:

TOS is an integer index. TOS-1 is the array base pointer. UB1 is the number of elements per word. UB2 is the field width (in bits). A packed field pointer to the indicated field is pushed.

Algorithm:

pack-ptr.right_bit:= (TOS mod UB1) * UB2
pack-ptr.field_width:= UB2
pack-ptr.addr:= TOS-1 + (TOS div UB1) * x

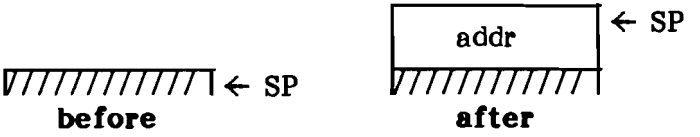
(Where x=2 for byte-addressed PMEs, and x=1 for word-addressed PMEs.)

LAE Load Extended Address

Opcode: 155 9B

Operation: LAE UB, B

Stack:



Description:

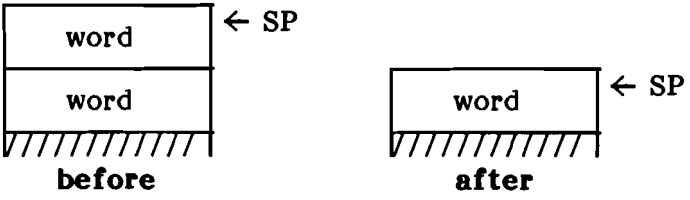
The address of the variable with offset B in the global activation record of local segment UB is pushed onto the stack.

LAND Logical And

Opcode: 161 A1

Operation: LAND

Stack:



Description:

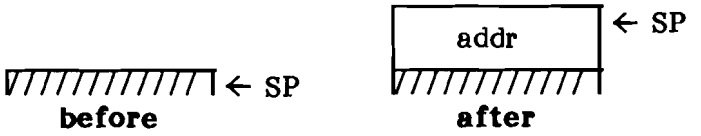
TOS and TOS-1 are ANDed bit-wise, and the result is placed on the stack.

LAO Load Global Address

Opcode: 134 86

Operation: LAO B

Stack:



Description:

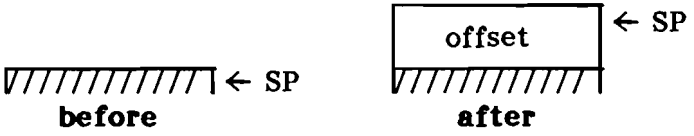
The address of the variable with offset B in the global activation record is pushed onto the stack.

LCO Load Constant Offset

Opcode: 130 82

Operation: LCO B

Stack:



Description:

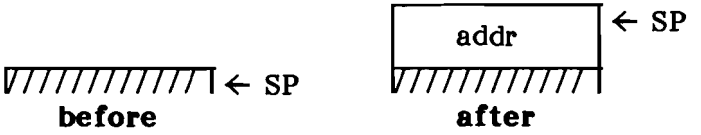
B is a word offset into the constant pool of the current segment. The address of the indicated constant is converted into a segment relative offset. This offset is a word offset on word-addressed PMEs and a byte offset on byte-addressed PMEs. The computed offset is pushed onto the stack.

LDA Load Intermediate Address

Opcode: 136 88

Operation: LDA DB, B

Stack:



Description:

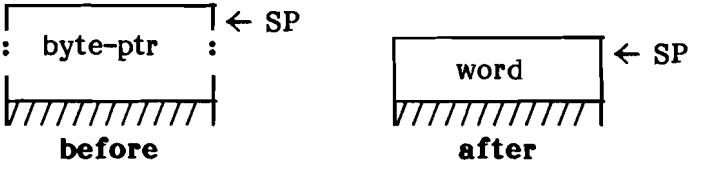
DB indicates the number of static links to traverse to find the activation record to use. (DB=0 indicates the local activation record; DB=1 indicates the parent activation record; and so forth.) The address of the variable with offset B in the indicated activation record is pushed onto the stack.

LDB Load Byte

Opcode: 167 A7

Operation: LDB

Stack:



Description:

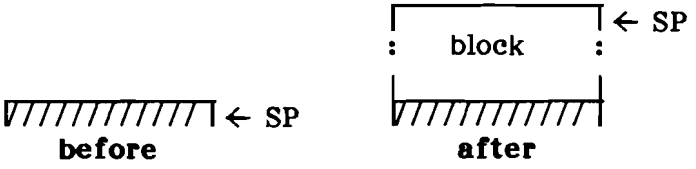
TOS is a byte pointer. TOS is replaced by the indicated byte with the high byte zero.

LDC Load Constant

Opcode: 131 83

Operation: LDC UB1, B, UB2

Stack:



Description:

If less than UB2+20 words are available on the stack, a stack fault is issued.

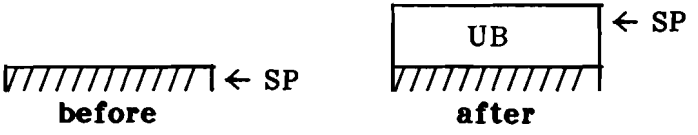
B is a word offset into the constant pool of the currently executing segment. UB2 words starting at that offset are pushed onto the stack, preserving the order of the words. If UB1, the mode, is 2, and the current segment is of opposite byte sex from the host processor, the bytes of each word are swapped as they are loaded.

LDCB Load Constant Byte

Opcode: 128 80

Operation: LDCB UB

Stack:



Description:

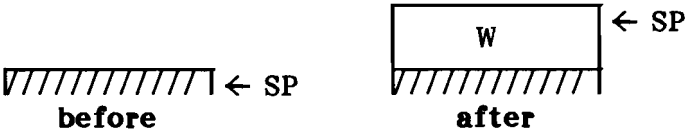
The constant UB with high byte zero is pushed onto the stack. LDCB is used to load a constant in the range 0 through 255.

LDCI Load Constant Integer

Opcode: 129 81

Operation: LDCI W

Stack:



Description:

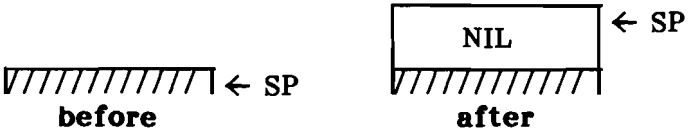
The constant word W is pushed onto the stack.

LDCN Load Constant NIL

Opcode: 152 98

Operation: LDCN

Stack:



Description:

The processor-dependent value NIL is pushed onto the stack. See the table below for the value of NIL for each processor.

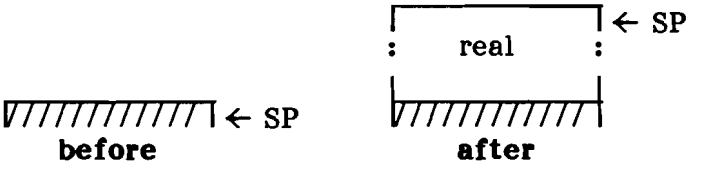
	NIL
Z80	0001
8080	0001
6502	0000
6809	0000
68000	0000
HP-87	0000
PDP-11	F001
9900	0000
8086	0000

LDCRL Load Constant Real

Opcode: 242 F2

Operation: LDCRL B

Stack:



Description:

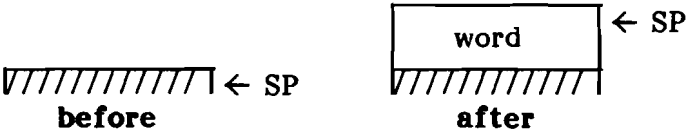
The real constant at offset B in the constant pool of the currently executing segment is loaded onto the stack.

LDE Load Extended

Opcode: 154 9A

Operation: LDE UB, B

Stack:



Description:

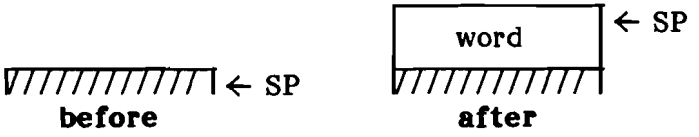
The word at offset B in the global activation record of local segment UB is pushed onto the stack.

LDL Load Local

Opcode: 135 87

Operation: LDL B

Stack:



Description:

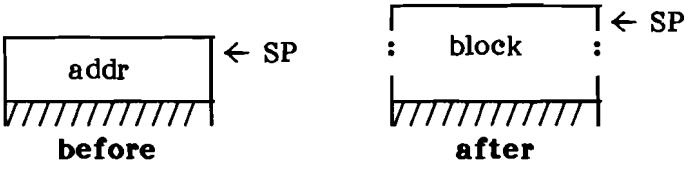
The word with word offset B in the local activation record is pushed onto the stack.

LDM Load Multiple

Opcode: 208 D0

Operation: LDM UB

Stack:



Description:

If less than UB+20 words are available on the stack, a stack fault is issued.

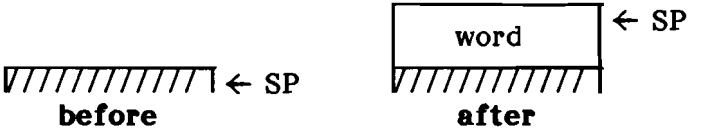
TOS is a pointer to a block of UB words. The block is pushed onto the stack, preserving the order of the words.

LDO Load Global

Opcode: 133 85

Operation: LDO B

Stack:



Description:

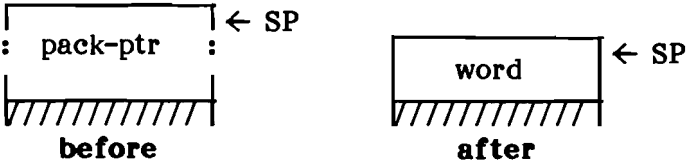
The word with offset B in the global activation record is pushed onto the stack.

LDP Load Packed

Opcode: 201 C9

Operation: LDP

Stack:



Description:

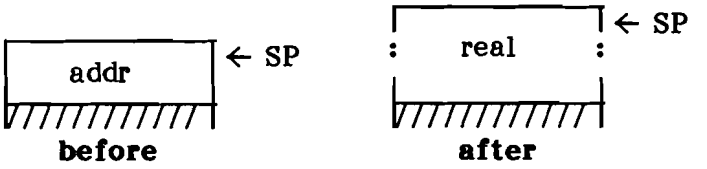
The packed field pointer TOS is replaced with the field it designates. Before being pushed onto the stack, the field is right-justified and zero-filled.

LDRL Load Real

Opcode: 243 F3

Operation: LDRL

Stack:



Description:

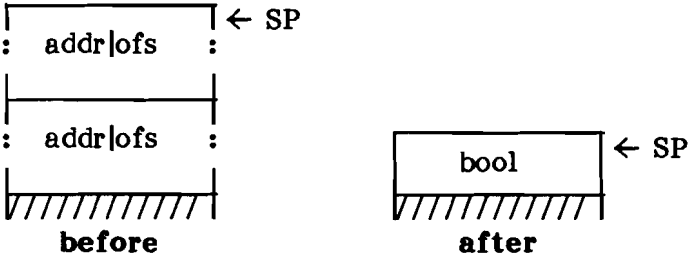
TOS is the address of a real variable. TOS is replaced with the indicated real.

LEBYT Less Than or Equal Byte Array

Opcode: 186 BA

Operation: LEBYT UB1, UB2, B

Stack:



Description:

UB1 and UB2 are mode flags. They refer to TOS and TOS-1, respectively. TOS and TOS-1 each point to a byte array (if the corresponding UB is zero) or to the offset of the byte array in the current segment. B is the size (in bytes) of the arrays.

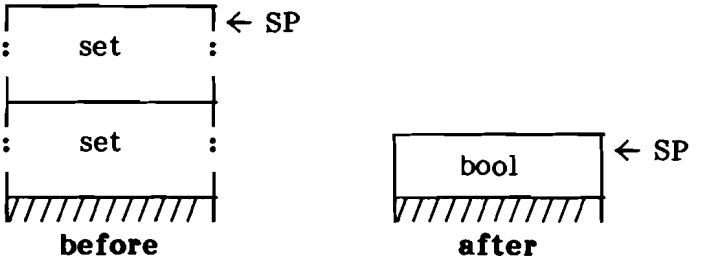
The boolean result of the comparison TOS-1 <= TOS is pushed on the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the arrays is reached. If there is a mismatch and the character in TOS-1 > the character in TOS, FALSE is pushed onto the stack. Otherwise, TRUE is pushed.

LEPWR Less Than or Equal Set

Opcode: 183 B7

Operation: LEPWR

Stack:



Description:

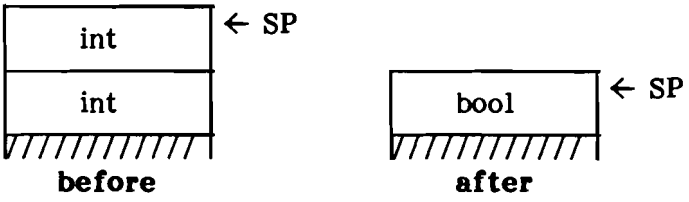
TRUE is pushed if TOS-1 is a subset of TOS. Otherwise, FALSE is pushed.

LEQI Less Than or Equal Integer

Opcode: 178 B2

Operation: LEQI

Stack:



Description:

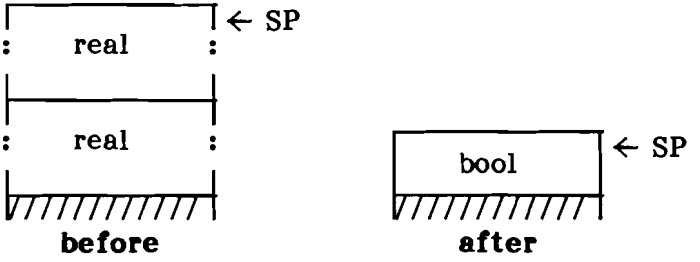
The boolean result of the signed comparison TOS-1 <= TOS is pushed onto the stack.

LEREAL Less Than or Equal Real

Opcode: 206 CE

Operation: LEREAL

Stack:



Description:

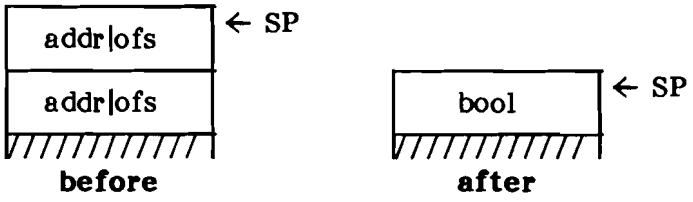
The boolean result of the comparison TOS-1 <= TOS is pushed onto the stack.

LESTR Less Than or Equal String

Opcode: 233 E9

Operation: LESTR UB1, UB2

Stack:



Description:

UB1 and UB2 are mode flags. They refer to TOS and TOS-1, respectively. TOS and TOS-1 each point to a string (if the corresponding UB is zero) or to the offset of the string in the current segment.

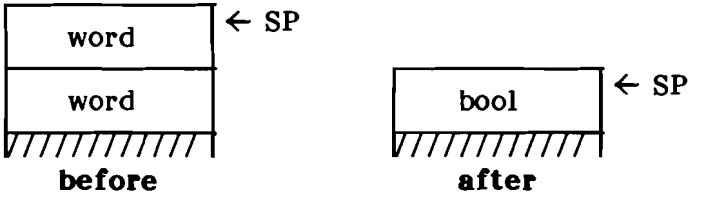
The boolean result of the comparison TOS-1 <= TOS is pushed onto the stack. The bytes are compared one by one in the natural byte order of the processor until a mismatch is found or the end of the shorter string is reached. The comparison begins at the second element of the strings. If there is a mismatch in any character position and the character in TOS-1 > the character in TOS, FALSE is pushed onto the stack. Otherwise, the lengths of the strings are compared, and the boolean result of the comparison length(TOS-1) <= length(TOS) is pushed.

LEUSW Unsigned Less Than or Equal

Opcode: 180 B4

Operation: LEUSW

Stack:

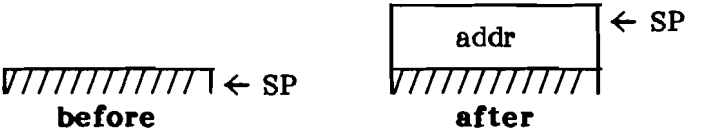


Description:

The boolean result of the unsigned comparison TOS-1 <= TOS is pushed onto the stack.

LLA	Load Local Address	
Opcode:	132	84
Operation:	LLA	B

Stack:



Description:

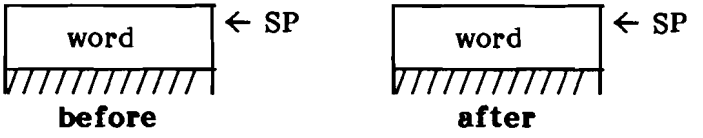
The address of the variable with offset B in the local activation record is pushed onto the stack.

LNOT Logical Not

Opcode: 229 E5

Operation: LNOT

Stack:



before **after**

Description:

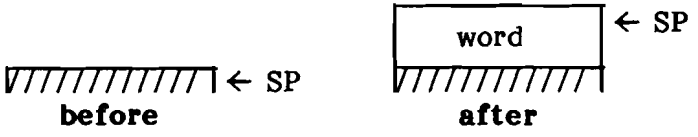
TOS is replaced by its one's complement.

LOD Load Intermediate

Opcode: 137 89

Operation: LOD DB, B

Stack:



Description:

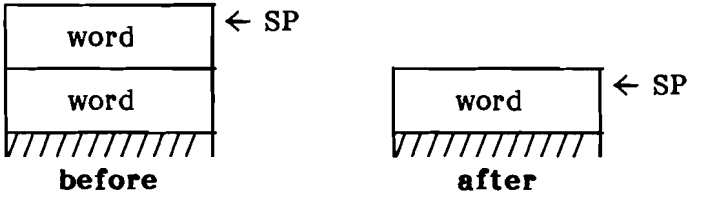
DB indicates the number of static links to traverse to find the activation record to use. (DB=0 indicates the local activation record; DB=1 indicates the parent activation record; and so forth.) The word with offset B in the indicated activation record is pushed onto the stack.

LOR Logical Or

Opcode: 160 A0

Operation: LOR

Stack:



Description:

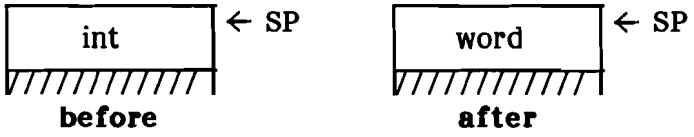
TOS and TOS-1 are ORed bit-wise, and the result is placed on the stack.

LPR Load Processor Register

Opcode: 157 9D

Operation: LPR

Stack:



Description:

TOS is a register number. The value of the register indicated in TOS is pushed onto the stack. If TOS is negative, the following table indicates which register is pushed:

-1	CURTASK
-2	EVEC
-3	READYQ

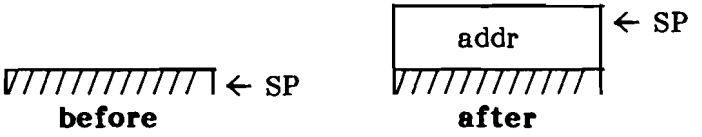
If TOS is positive, the current p-machine registers are saved in the TIB, and TOS is the word index of the register in the TIB to be pushed. If TOS is less than -3 or greater than the size of a TIB, the result of LPR is undefined.

LSL Load Static Link

Opcode: 153 99

Operation: LSL DB

Stack:



Description:

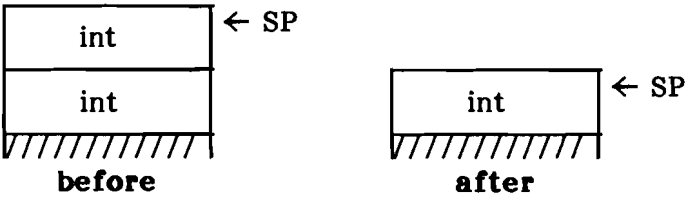
DB indicates the number of static links to traverse. A pointer to the MSCW that is DB links above the current MSCW is pushed onto the stack.

MODI Modulo Integers

Opcode: 143 8F

Operation: MODI

Stack:



Description:

If TOS is zero, a divide-by-zero execution error occurs.

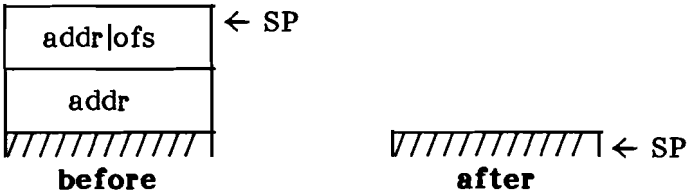
Otherwise, TOS is replaced by TOS-1 MOD TOS, where the MOD operation is as follows. The MOD operation is undefined if TOS is negative, but the PME shouldn't cause an execution error. The result of MOD is always an integer between 0 and (TOS) - 1. This result is calculated as if TOS were added or subtracted from TOS-1 until the result is in the desired range. Note: the result of the MOD operation isn't really the remainder of the DIV operation if TOS-1 is negative.

MOV Move

Opcode: 197 C5

Operation: MOV UB, B

Stack:



Description:

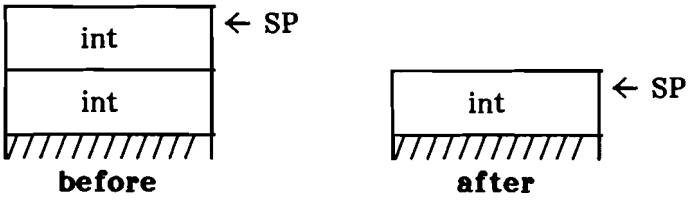
TOS is either the address of a word block (if UB=0) or the offset of a constant word block in the current segment (if UB<>0). B words are moved from the source designated by TOS to the destination address TOS-1. IF UB=2, and the current segment has opposite byte sex from the host processor, the bytes of each word are swapped as the words are moved.

MPI Multiply Integer

Opcode: 140 8C

Operation: MPI

Stack:



Description:

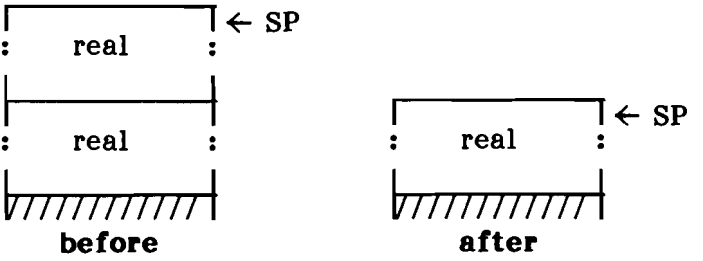
TOS is replaced by TOS-1 * TOS. The result should be computed as if it were an unsigned operation on 32-bit operands, and only the lowest 16 bits were retained for the result.

MPR Multiply Real

Opcode: 194 C2

Operation: MPR

Stack:



Description:

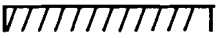
TOS is replaced by the value TOS-1 * TOS. The result must be zero on underflow. A floating point execution error is issued on overflow.

NAT Enter Native Code

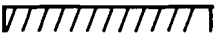
Opcode: 168 A8

Operation: NAT

Stack:



before



after

Description:

Control is transferred to the native code that begins directly after the NAT instruction. It may be necessary to increment IPC to a word boundary on word-oriented processors.

NAT-INFO Native Code Information

Opcode: 169 A9

Operation: NAT-INFO B

Stack:

before **after**

Description:

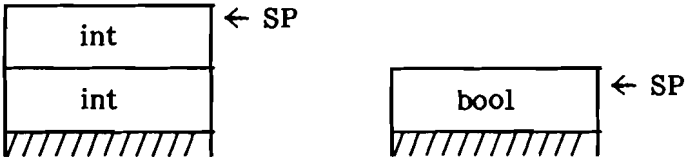
IPC is incremented to B bytes beyond the byte starting just after B in the p-code stream. The bytes after B contain information for the native-code generators. This instruction acts like a long form of NOP or a forward jump.

NEQI Not Equal Integer

Opcode: 177 B1

Operation: NEQI

Stack:



before **after**

Description:

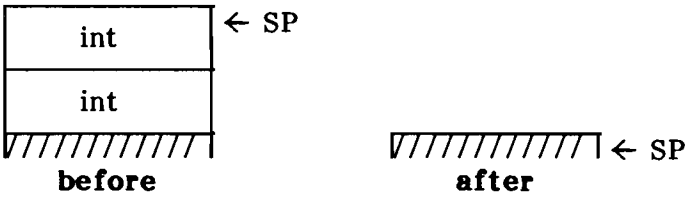
The boolean result of the comparison TOS-1 <> TOS is pushed onto the stack.

NFJ Not Equal False Jump

Opcode: 211 D3

Operation: NFJ SB

Stack:



Description:

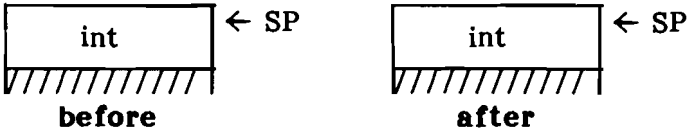
If TOS = TOS-1, a jump is made, relative to the next instruction, by the byte offset SB.

NGI Negate Integer

Opcode: 225 E1

Operation: NGI

Stack:



before **after**

Description:

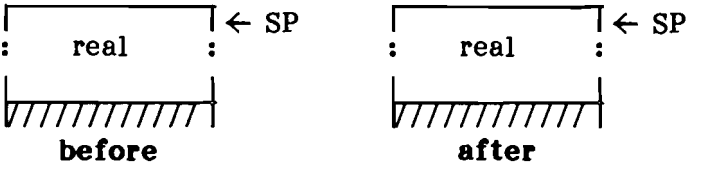
TOS is replaced by the negative (two's complement) of TOS. If TOS was initially -32768, the result should be -32768.

NGR Negate Real

Opcode: 228 E4

Operation: NGR

Stack:



Description:

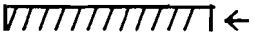
TOS is replaced by the inverse of TOS.

NOP No Operation

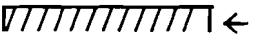
Opcode: 156 9C

Operation: NOP

Stack:



before



after

Description:

No operation is performed. Execution continues.

RESERVE1..RESERVE6 **Reserved**

Opcode: 250..255 FA..FF

Operation: RESERVEn

Stack:

before **after**

Description:

RESERVEn generates an unimplemented instruction execution error.

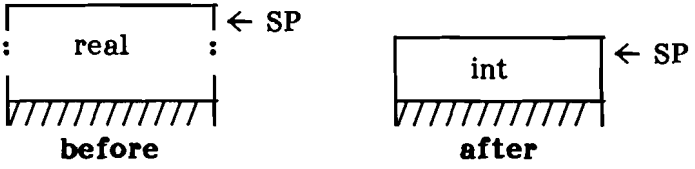
These opcodes are reserved for internal use by the compilers.

RND Round Real

Opcode: 191 BF

Operation: RND

Stack:



Description:

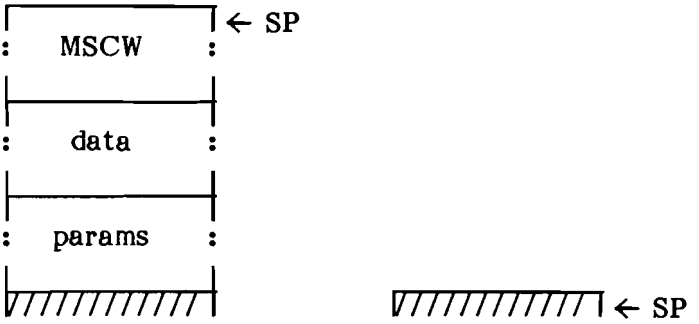
Real TOS is converted to an integer by rounding, and the result is pushed on the stack. If the result isn't in the range -32768 to 32767, a floating point execution error is issued.

RPU Return from Procedure

Opcode: 150 96

Operation: RPU B

Stack:



Description:

Execution returns to the calling procedure.

The E_Rec pointer in the MSCW indicates the segment to return to. If the segment isn't in memory, a segment fault is issued.

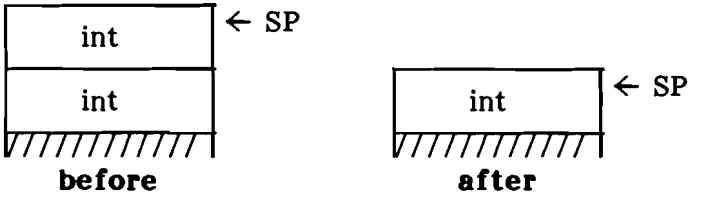
Otherwise, MP is set to the Dynamic_Link field of the MSCW. If the Proc field of the MSCW is positive, IPC is restored from the MSCW. Otherwise, IPC is set to the Exit_IC value found just before the procedure code in the segment. CURPROC is restored from the MSCW (negating the value, if necessary). If the E_Rec pointer of the MSCW differs from EREC, EREC and EVEC are set to reflect the new segment.

SBI Subtract Integer

Opcode: 163 A3

Operation: SBI

Stack:



Description:

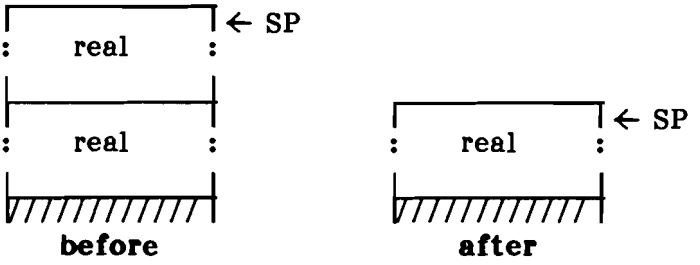
TOS is replaced by TOS-1 - TOS. The result should be computed as if it were an unsigned operation on 32-bit operands, and only the lowest 16 bits were retained for the result. Thus, overflow or underflow will "wrap around" to the opposite sign.

SBR Subtract Real

Opcode: 193 C1

Operation: SBR

Stack:



Description:

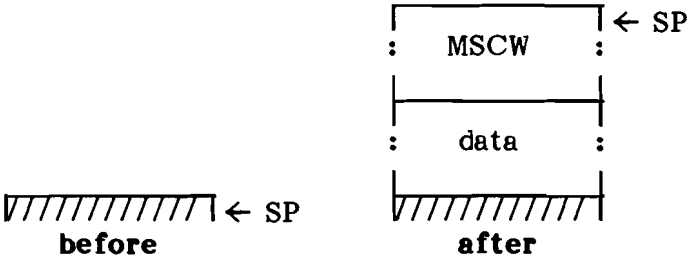
TOS is replaced by the value TOS-1 - TOS. The result should be zero on underflow. A floating point execution error is issued on overflow.

SCIP1..SCIP2 Short Call Intermediate Procedure

Opcode: 239..240 EF..F0

Operation: SCIPn UB

Stack:



Description:

Intermediate procedure UB in the currently executing segment is called.

If the Data_Size word for procedure UB is negative, nothing is allocated on the stack, and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to the lexical parent (SCIP1) or grandparent (SCIP2) of the current MSCW. MP is set to point to the new MSCW, CURPROC is set to UB, and IPC is set to the first p-code of procedure UB.

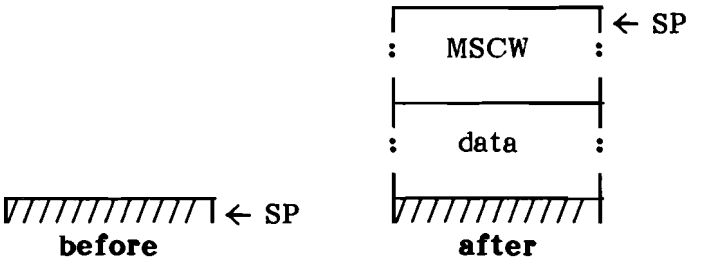
If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

SCXG1..SCXG8 Short Call External Global

Opcode: 112..119 70..77

Operation: SCXGn UB

Stack:



Description:

The global procedure UB in segment n is called.

If segment UB isn't in memory, a segment fault is issued.

If the instruction is SCXG1 and the procedure number matches one of the standard procedure numbers, the p-code performs one of these standard procedures, instead of the call. See the section describing the standard procedures.

If the Data_Size word for procedure UB is negative, nothing is allocated on the stack, and a native code call is made. P-code execution resumes with the following p-code.

Otherwise, Data_Size words and an MSCW are allocated on the stack. The Static_Link field of the MSCW is set to the new BASE (the global data MSCW). MP is set to point to the new MSCW, CURPROC is set to UB, and IPC is set to the first p-code of procedure UB. EREC and EVEC are set to reflect the new environment.

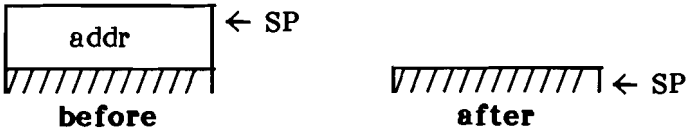
If there aren't 40 words left on the stack after the MSCW and data are allocated, a stack fault is issued.

SIGNAL Signal

Opcode: 222 DE

Operation: SIGNAL

Stack:



Description:

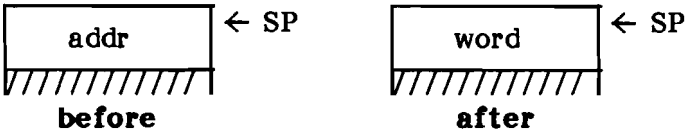
TOS is the address of a semaphore. If the semaphore's wait queue is empty or the count is negative, the count is incremented by one. Otherwise, the TIB at the head of the semaphore's wait queue is put on the ready queue, and its Hang_Ptr is set to NIL. If the new task has a higher priority than the current task, a task switch occurs.

SIND0..SIND7 Short Index

Opcode: 120..127 78..7 F

Operation: SINDn

Stack:



Description:

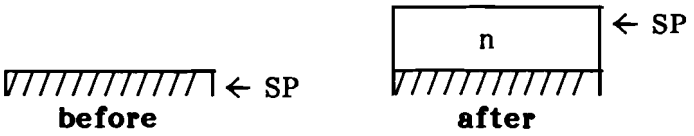
TOS is the address of a record. TOS is replaced with word n of the record. SINDn is used to index into the first eight words of a record. The value of n is <opcode>-120.

SLDC0..SLDC31 Short Load Constant

Opcode: 0..31 00..1F

Operation: SLDCn

Stack:



Description:

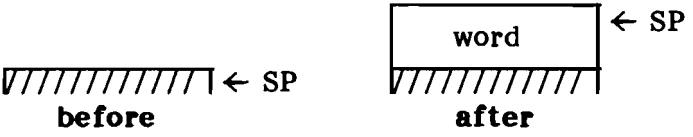
The constant word whose value is encoded in the opcode is pushed onto the stack. The value n is the value of the opcode itself. SLDCn is used to load a constant between 0 and 31.

SLDL1..SLDL16 Short Load Local

Opcode: 32..47 20..2F

Operation: SLDLn

Stack:



Description:

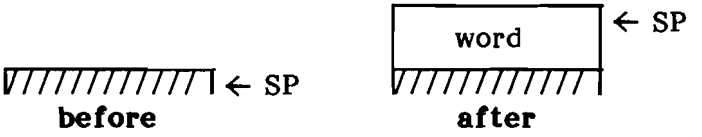
The word with word offset n in the local activation record is pushed onto the stack. SLDLn is used to load one of the first 16 local words. The value of n is <opcode>-31.

SLDO1..SLDO16 Short Load Global

Opcode: 48..63 30..3F

Operation: SLDOn

Stack:



Description:

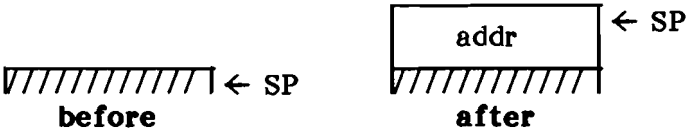
The word with offset n in the global activation record is pushed onto the stack. SLDOn is used to load global words with offsets between 1 and 16. The value of n is <opcode>-47.

SLLA1..SLLA8 Short Load Local Address

Opcode: 96..103 60..67

Operation: SLLAn

Stack:



Description:

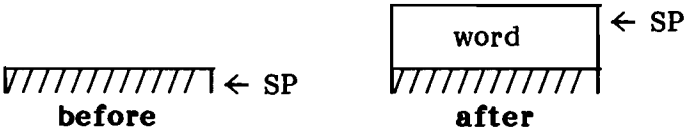
The address of the variable with offset n in the local activation record is pushed onto the stack. SLLAn is used to load the address of local variables with offsets between 1 and 8. The value of n is <opcode>-95.

SLOD1..SLOD2 Short Load Intermediate

Opcode: 173..174 AD..AE

Operation: SLODn B

Stack:



Description:

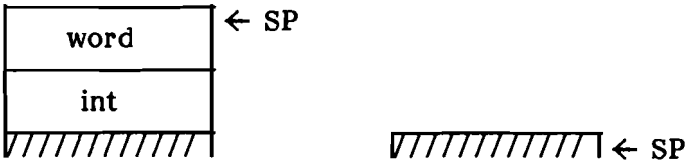
The word with offset B in the activation record of the parent (SLOD1) or grandparent (SLOD2) of the local activation record is pushed onto the stack. The value of n is <opcode>-172.

SPR Store Processor Register

Opcode: 209 D1

Operation: SPR

Stack:



Description:

TOS-1 is a register number. If TOS-1 is negative, TOS is stored in one of the following registers:

-1	CURTASK
-2	EVEC
-3	READYQ

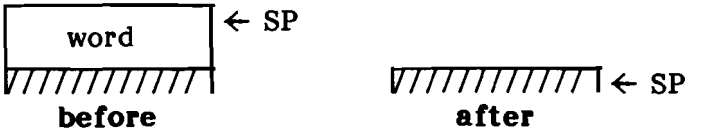
Otherwise, the current p-machine registers are stored in the TIB. TOS is stored in the TIB at offset TOS-1. Finally, the p-machine registers are restored from the TIB.

SRO Store Global

Opcode: 165 A5

Operation: SRO B

Stack:



Description:

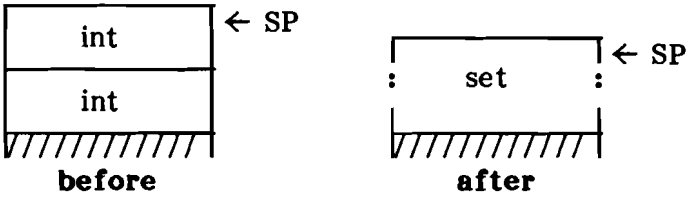
TOS is stored in the word with offset B in the global activation record.

SRS Subrange Set

Opcode: 188 BC

Operation: SRS

Stack:



Description:

If less than 20 words will be available on the stack after this operation, a stack fault is issued.

The integers TOS and TOS-1 must be in the range [0 through 4097]. If not, a value range execution error is issued.

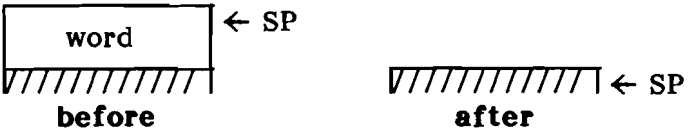
If TOS-1 > TOS, the empty set is pushed. Otherwise, a set is created containing the elements between TOS-1 and TOS, inclusive, as members. This set is pushed on the stack.

SSTL1..SSTL8 Short Store Local

Opcode: 104..111 68..6F

Operation: SSTLn

Stack:



Description:

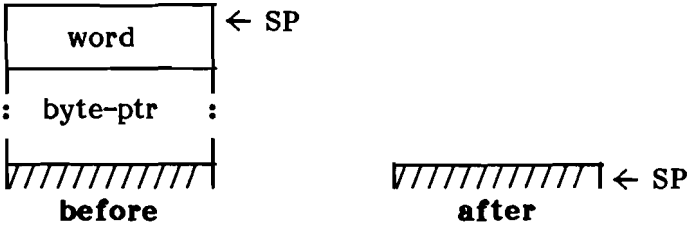
TOS is stored in the word with offset n in the local activation record. SSTLn is used to store in one of the first eight local words. The value of n is <opcode>-103.

STB Store Byte

Opcode: 200 C8

Operation: STB

Stack:



Description:

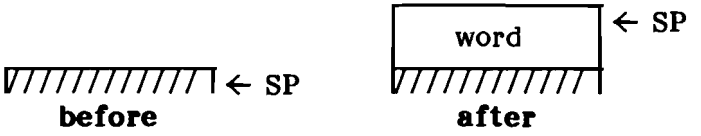
The low byte of TOS is stored in the location pointed to by byte pointer TOS-1.

STE Store Extended

Opcode: 217 D9

Operation: STE UB, B

Stack:



Description:

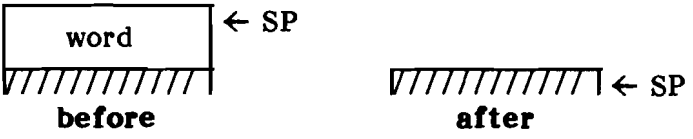
TOS is stored in the word with offset B in the global activation record of local segment UB.

STL Store Local

Opcode: 164 A4

Operation: STL B

Stack:



Description:

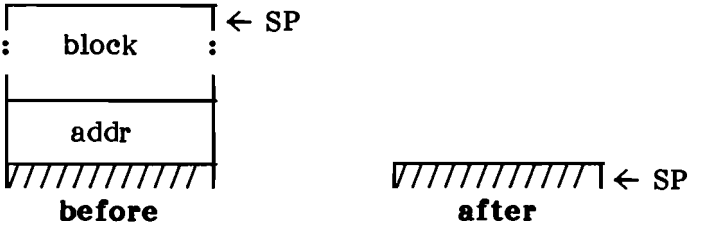
TOS is stored in the word with offset B in the local activation record.

STM Store Multiple

Opcode: 142 8E

Operation: STM UB

Stack:



Description:

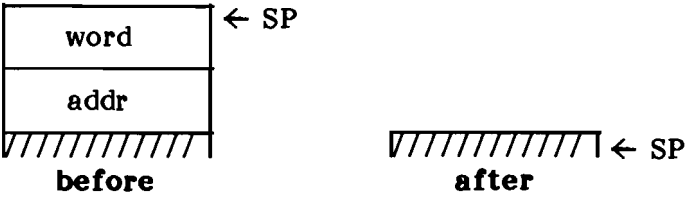
TOS is a block of UB words. The block is stored at address TOS-1, preserving the order of the words.

STO Store

Opcode: 196 C4

Operation: STO

Stack:



Description:

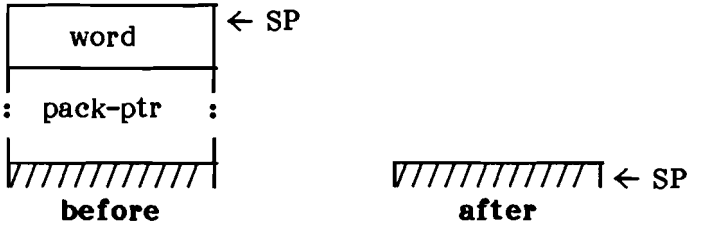
TOS is stored in the word pointed to by TOS-1.

STP Store Packed

Opcode: 202 CA

Operation: STP

Stack:



Description:

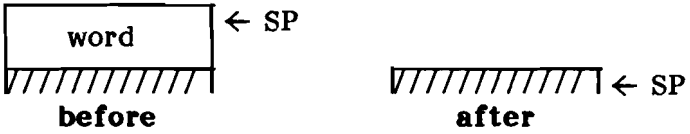
TOS contains right-justified data. TOS-1 is a packed field pointer. TOS is masked to the field width indicated in TOS-1, then stored into the field described by TOS-1.

STR Store Intermediate

Opcode: 166 A6

Operation: STR DB, B

Stack:



Description:

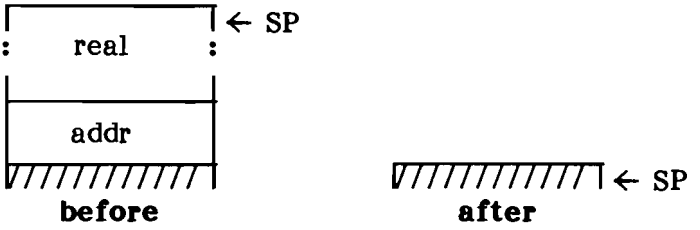
DB indicates the number of static links to traverse to find the activation record to use. (DB=0 indicates the local activation record; DB=1 indicates the parent activation record; and so forth.) TOS is stored into the word with offset B in the indicated activation record.

STRL Store Real

Opcode: 244 F4

Operation: STRL

Stack:



Description:

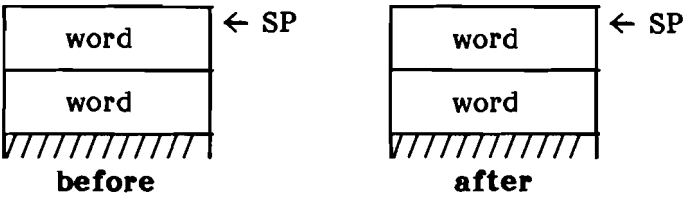
TOS is a real value. TOS-1 is an address. TOS is stored at the address TOS-1.

SWAP Swap

Opcode: 189 BD

Operation: SWAP

Stack:



Description:

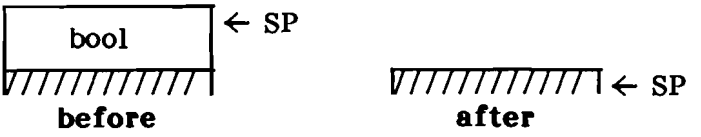
Word TOS is swapped with word TOS-1 on the stack.

TJP True Jump

Opcode: 241 F1

Operation: TJP SB

Stack:



Description:

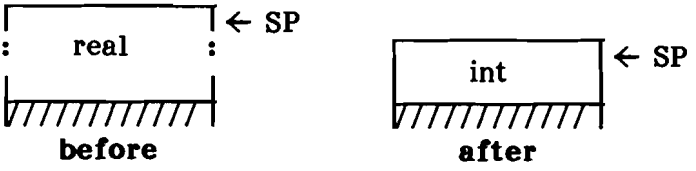
If TOS is TRUE, a jump is made, relative to the next instruction, by the byte offset SB.

TNC Truncate Real

Opcode: 190 BE

Operation: TNC

Stack:



Description:

Real TOS is converted to an integer by truncating, and the result is pushed onto the stack. If the result isn't in the range -32768 to 32767, a floating point execution error is issued.

UJP Unconditional Jump

Opcode: 138 8A

Operation: UJP SB

Stack:

before **after**

Description:

A jump is made, relative to the next instruction, by the byte offset SB.

UJPL Unconditional Jump Long

Opcode: 139 8B

Operation: UJPL W

Stack:

before **after**

Description:

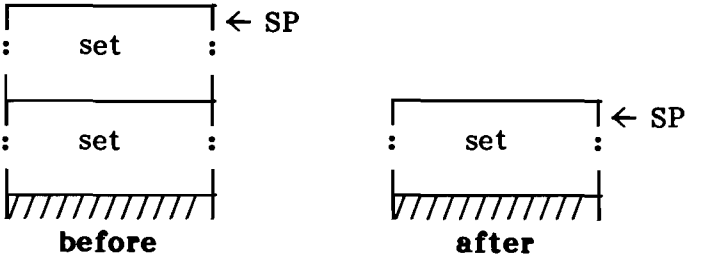
A jump is made, relative to the next instruction, by the byte offset W.

UNI Set Union

Opcode: 219 DB

Operation: UNI

Stack:



Description:

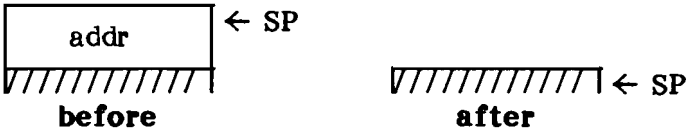
The union (bit-wise OR) of the sets TOS and TOS-1 is pushed onto the stack.

WAIT Wait

Opcode: 223 DF

Operation: WAIT

Stack:



Description:

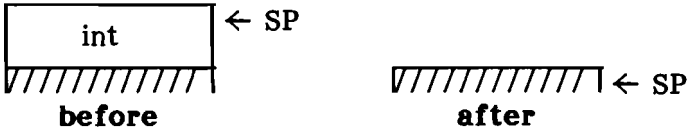
TOS is the address of a semaphore. If the semaphore's count is greater than zero, the count is decremented by one. Otherwise, the current TIB is put on the semaphore's wait queue, its Hang_Ptr is set to TOS, and a task switch occurs.

XJP Case Jump

Opcode: 214 D6

Operation: XJP B

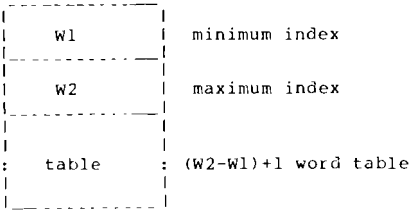
Stack:



Description:

B is the offset of the case jump table within the constant pool of the current segment. TOS is the index.

The case jump table is structured as follows:



Every entry is an integer quantity in the byte sex of the current segment. If the current segment has opposite byte sex from the host processor, each word in the table must be flipped before it is used.

If TOS is in the range W1 through W2, inclusive, a jump is made, relative to the next instruction, by the quantity in word (TOS - W1) in the table. (The table is word-indexed starting with zero, and is found just after word W2 in the case jump table.) Otherwise, no jump is performed.

STANDARD PROCEDURES

The standard procedures are procedures that are implemented in the PME directly, either for speed or because the nature of the procedure requires that it be written in native code. A standard procedure is called via a CXG or SCXG1 instruction. Which procedure is executed is based on the procedure number.

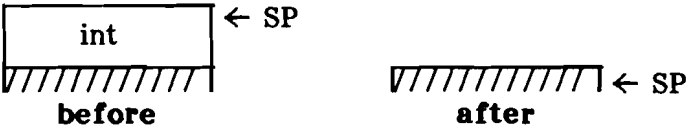
Most of the standard procedures require parameters on the stack, and some expect a function return value to be passed back. In some sense they act more like individual p-codes than procedures, because no RPU instruction is executed to return control to the caller. For this reason, the procedure descriptions that follow are presented in the same format as were p-code descriptions—showing the stack before and after execution.

UNIT I/O PROCEDURES

UNITCLEAR (UNIT)

Procedure: 34

Stack:



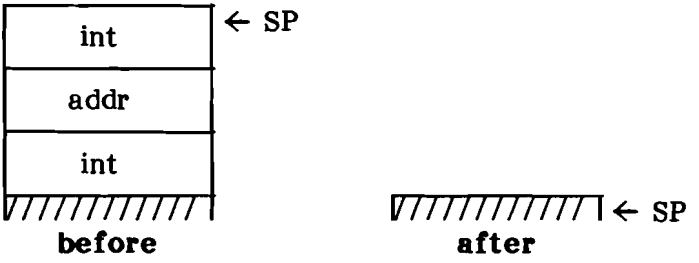
Description:

TOS is a unit number. The device with unit number TOS is initialized to its "power-up" state. On return, IORESULT contains status information. For more information on UNITCLEAR, refer to the BIOS documentation in Chapter 4.

UNITSTATUS (UNIT, STAT_REC, CONTROL)

Procedure: 36

Stack:



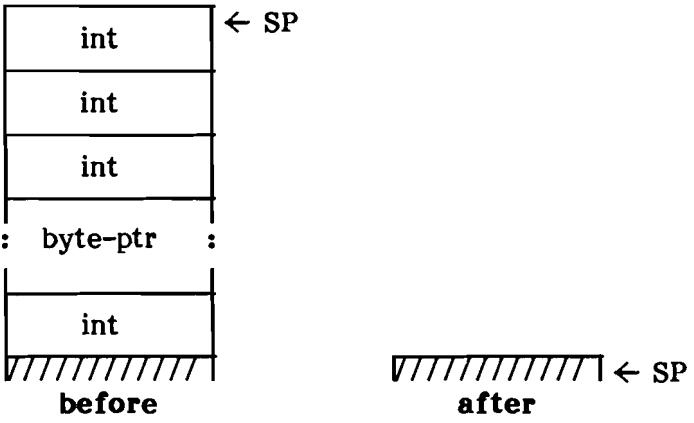
Description:

TOS is the control word. TOS-1 is the address of a buffer. TOS-2 is the unit number. Status information about the unit described in TOS-2 and the direction described in TOS is written to the buffer, TOS-1. If TOS = 0 the direction is output. If TOS = 1 the direction is input. On return, IORESULT contains status information. For more information about UNITSTATUS, refer to the BIOS documentation.

UNITREAD (UNIT, BUF, LEN, BLOCK, CTRL)

Procedure: 18

Stack:



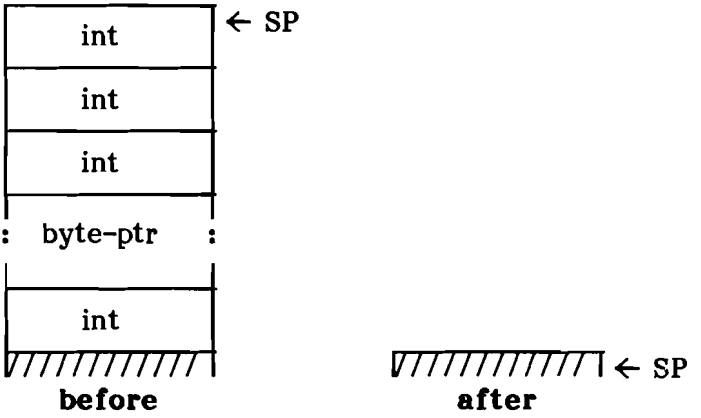
Description:

TOS is the control word. TOS-1 is the block number. TOS-2 is the number of bytes to read. TOS-3 is the address of the destination buffer. TOS-4 is the unit number. The number of bytes specified in TOS-2 is read from unit TOS-4 at block TOS-1 into buffer TOS-3. The control word selects different modes of UNITREAD. On return, IORESULT contains status information. For more information about UNITREAD, refer to the BIOS documentation.

UNITWRITE (UNIT, BUF, LEN, BLOCK, CTRL)

Procedure: 19

Stack:



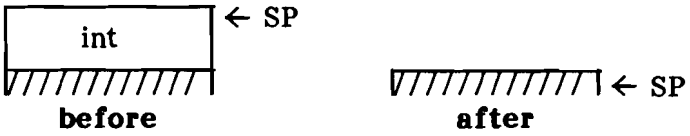
Description:

TOS is the control word. TOS-1 is the block number. TOS-2 is the number of bytes to write. TOS-3 is the address of the source buffer. TOS-4 is the unit number. The number of bytes specified in TOS-2 is written to unit TOS-4 at block TOS-1 from buffer TOS-3. The control word selects different modes of UNITWRITE. On return, IORESULT contains status information. For more information about UNITWRITE, refer to the BIOS documentation.

UNITWAIT (UNIT)

Procedure: 33

Stack:



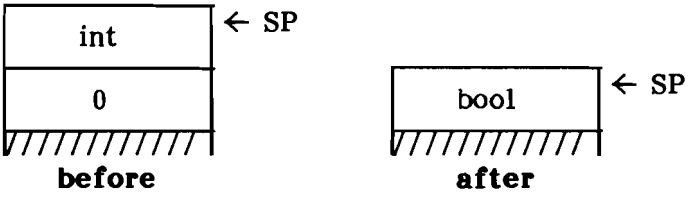
Description:

TOS is a unit number. The p-machine waits until all I/O on unit TOS is completed. On return, IORESULT contains status information. For more information about UNITWAIT, refer to the BIOS documentation.

UNITBUSY (UNIT): BOOLEAN

Procedure: 31

Stack:



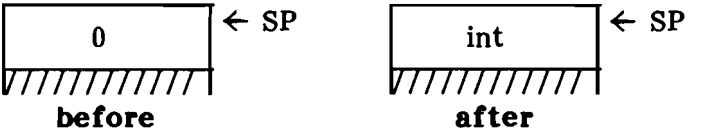
Description:

TOS is the unit number. TOS-1 is a function return word. UNITBUSY returns TRUE if there is any outstanding I/O on device TOS, and FALSE otherwise. On return, IORESULT contains status information. For more information about UNITBUSY, refer to the BIOS documentation.

IORESULT :INTEGER

Procedure: 30

Stack:



Description:

TOS is a return word. IORESULT returns the value of the p-machine register IORESULT.

IOCHECK

Procedure: 23

Stack:

Description:

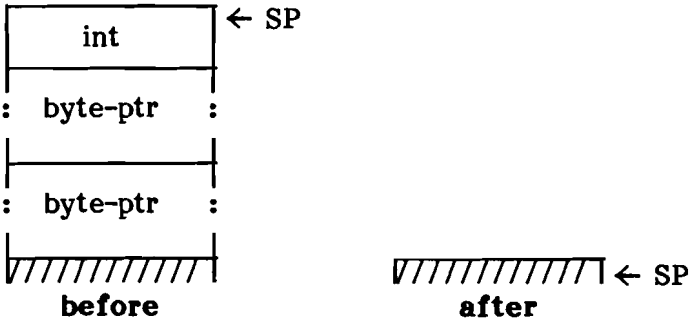
IOCHECK tests the p-machine register IORESULT for zero. If the register is nonzero, an I/O execution error is issued.

STRING PROCEDURES

MOVELEFT (SOURCE, DEST, LENGTH)

Procedure: 15

Stack:



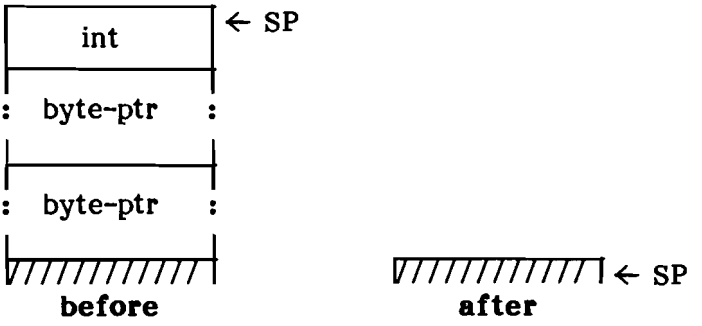
Description:

TOS is the number of bytes to move. TOS-1 is a pointer to the destination. TOS-2 is a pointer to the source. If TOS is zero or negative, no bytes are moved. Otherwise, the bytes are moved one at a time starting from the left (low order byte).

MOVERIGHT (SOURCE, DEST, LENGTH)

Procedure: 16

Stack:



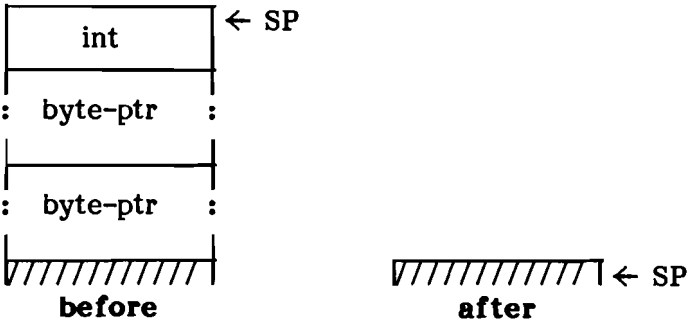
Description:

TOS is the number of bytes to move. TOS-1 is a pointer to the destination. TOS-2 is a pointer to the source. If TOS is zero or negative, no bytes are moved. Otherwise, the bytes are moved one at a time starting from the right (high order byte).

FILLCHAR (DEST, LENGTH, CHAR)

Procedure: 21

Stack:



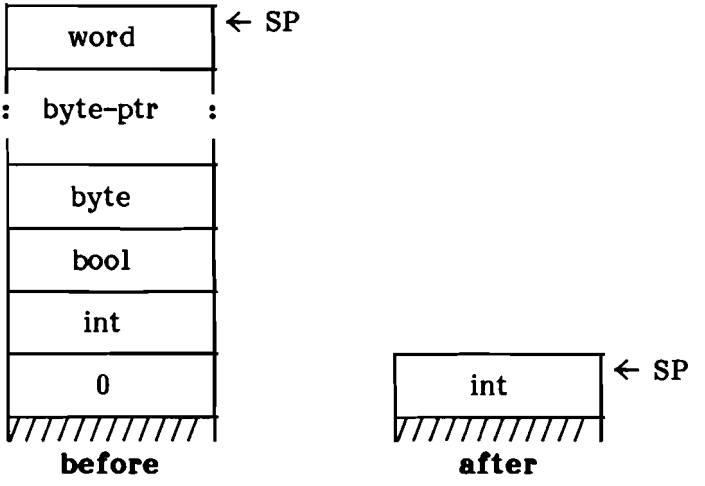
Description:

TOS is the character. TOS-1 is the length to fill. TOS-2 is the starting address for the fill. If TOS-1 is zero or negative, no filling is done. Otherwise, memory is filled with the byte TOS for TOS-1 bytes starting at address TOS-2.

SCAN (LEN, EXP, SOURCE): INT

Procedure: 22

Stack:



Description:

TOS is a mask field (unused). TOS-1 is a pointer to the array to scan. TOS-2 is the byte to look for. TOS-3 is the scan kind (0 means until equal, 1 means until not equal). TOS-4 is the length to scan. If TOS-4 is negative, the scan proceeds to the left. TOS-5 is the function result word.

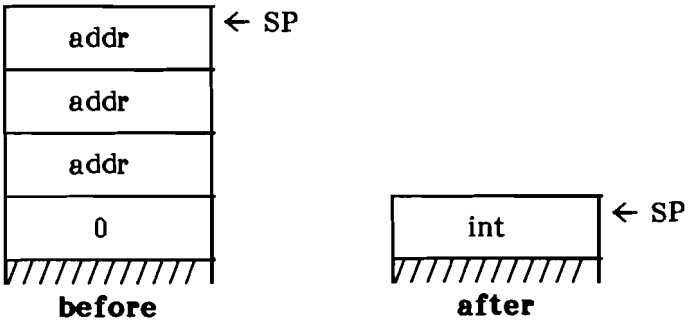
The array at TOS-1 is scanned in the direction indicated in TOS-4 until the character TOS-2 is found (TOS-3 = 0) or a nonmatching character is found (TOS-3 = 1) or until the length in TOS-4 is exhausted. The distance between the character where SCAN stopped and the start character is passed back as the function result.

COMPILER PROCEDURES

TREESEARCH (ROOT, FOUNDP, TARGET): INT

Procedure: 38

Stack:



Description:

TOS is a pointer to the target string, which is a packed array of eight characters. TOS-1 is a pointer to where the result of the search will be saved. TOS-2 is a pointer to the root of the identifier tree to be searched. TOS-3 is the return word.

TREESEARCH searches the symbol table tree TOS-2 for the target string TOS, returning a pointer to where the target was found in the variable pointed to by TOS-1. If the target wasn't found, the variable pointed to by TOS-1 will point to the leaf node of the tree that was searched last. The function result returns status information:

0	target was found
1	target is to the right
-1	target is to the left

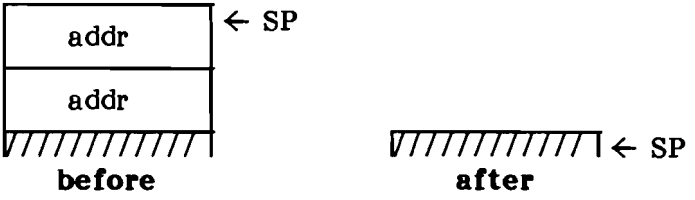
Each node of the tree contains the following fields at the indicated byte offsets:

0	name (8 characters)
8	right link (pointer)
10	left link (pointer)

IDSEARCH (SYMREC, BUFFER)

Procedure: 37

Stack:



Description:

TOS is the address of a buffer. TOS-1 is the address of a record that has the following fields at the indicated byte offsets:

0	SYMCURSOR
2	SY
4	OP
6	ID

IDSEARCH scans the buffer at byte offset SYMCURSOR for an identifier (string beginning with a letter, containing letters, digits and underscores), ignoring underscores and masking lowercase to uppercase. The identifier is blank-filled to eight characters, then placed in ID for a maximum of eight characters. SYMCURSOR is updated to point just past the identifier.

Finally, the identifier is looked up in a table of reserved words, and its two characteristics are filled into SY and OP. If the identifier is not found in the table, SY is set to 0 and OP is set to 15.

Here is the table of reserved words, along with the SY and OP values for each one:

ID	SY	OP
AND	39	2
ARRAY	44	15
BEGIN	19	15
CASE	21	15
CONST	28	15
DIV	39	3
DO	6	15
DOWNTO	8	15
ELSE	3	15
END	9	15
EXTERNAL	53	15
FOR	2	15
FILE	46	15
FORWARD	34	15
FUNCTION	32	15
GOTO	26	15
IF	20	15
IMPLEMEN	52	15
IN	41	14
INTERFAC	51	15
LABEL	27	15
MOD	39	4
NOT	38	15
OF	11	15
OR	40	7
PACKED	43	15
PROCEDUR	31	15

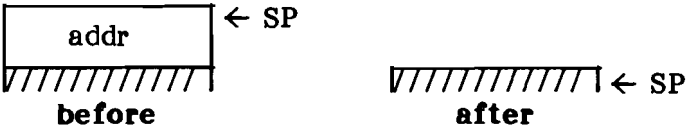
PROCESS	56	15
PROGRAM	33	15
REPEAT	22	15
RECORD	45	15
SET	42	15
SEGMENT	33	15
SEPARATE	54	15
THEN	12	15
TO	7	15
TYPE	29	15
UNIT	50	15
UNTIL	10	15
USES	49	15
VAR	30	15
WHILE	23	15
WITH	25	15

CODE POOL PROCEDURES

RELOCSEG (EREC)

Procedure: 4

Stack:



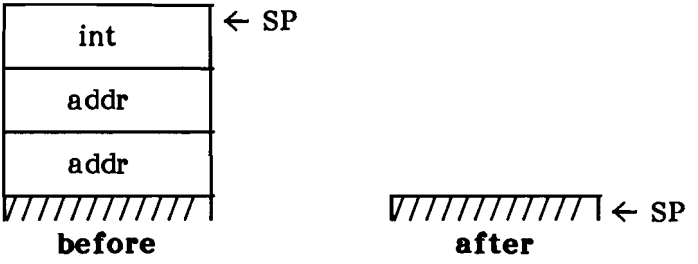
Description:

TOS is the address of an E_Rec. RELOCSEG relocates the segment pointed to by the ERec. Since RELOCSEG is called after a segment is first read into memory, all relocation is performed.

MOVESEG (SIB, SRCPOOL, SRCOFFSET)

Procedure: 14

Stack:



Description:

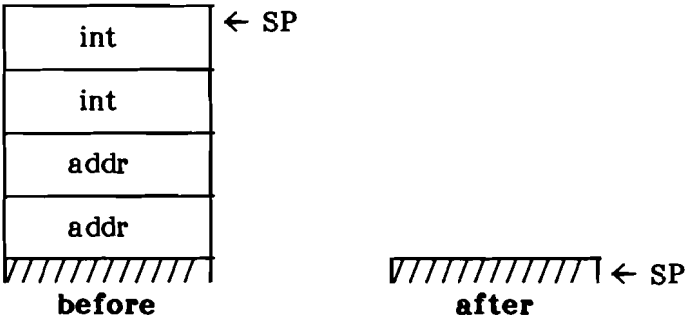
TOS is an offset to the segment within a code pool. TOS-1 is a pointer to a code pool descriptor of which the first two words are the pointer to the base of the pool. TOS-2 is the address of a SIB.

MOVESEG moves the segment at offset TOS in the pool described by TOS-1 to the location specified in the SIB pointed to by TOS-2, and relocates it. Only segment-relative relocation is performed.

GETPOOLBYTES
(DEST, POOLDESC, OFFSET, NBYTES)

Procedure: 24

Stack:



Description:

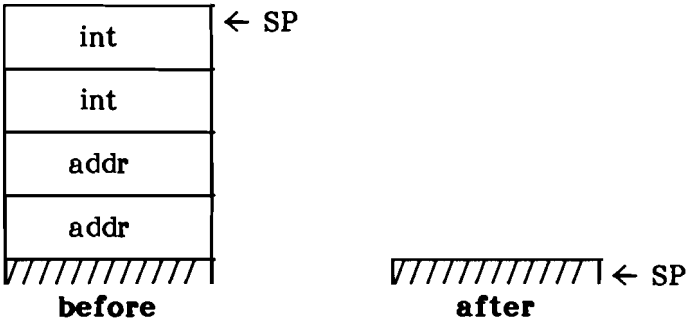
TOS is the number of bytes to get. TOS-1 is the offset within the pool of the bytes to get. TOS-2 points to a code pool descriptor of which the first two words point to the pool. TOS-3 points to the buffer where the bytes will be placed.

GETPOOLBYTES get TOS bytes from the pool described by TOS-2 at offset TOS-1, and places them at TOS-3.

PUTPOOLBYTES
(SOURCE, POOLDESC, OFFSET, NBYTES)

Procedure: 25

Stack:



Description:

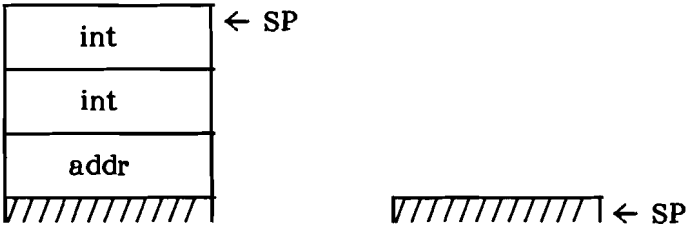
TOS is the number of bytes to put. TOS-1 is the offset within the pool of the bytes to put. TOS-2 points to a code pool descriptor of which the first two words point to the pool. TOS-3 points to the source buffer.

PUTPOOLBYTES writes TOS bytes from the buffer TOS-3 to the pool described by TOS-2 at offset TOS-1.

FLIPSEGBYTES (EREC, OFFSET, NWORDS)

Procedure: 26

Stack:



Description:

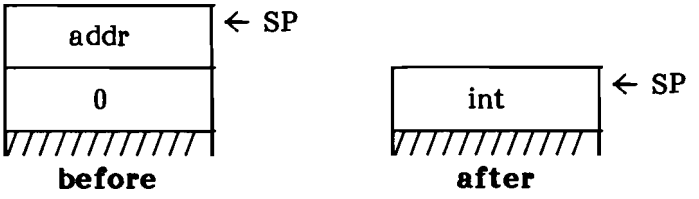
TOS is the number of words to flip. TOS-1 is the word offset within the segment where flipping will take place. TOS-2 is the address of the E_Rec describing the segment.

FLIPSEGBYTES flips TOS words starting at word offset TOS-1 in the segment described by TOS-2.

READSEG (EREC): INTEGER

Procedure: 39

Stack:



Description:

TOS is the address of the E_Rec describing a segment. TOS-1 is the return word.

READSEG reads the segment described by TOS into memory at the location described in the SIB. The completion code in p-machine register IORESULT is returned as the function result.

CONCURRENCY PROCEDURES

QUIET

Procedure: 27

Stack:

Description:

QUIET must disable all p-machine events such that no attached semaphore is signalled until the corresponding call to ENABLE is made.

ENABLE

Procedure: 28

Stack:

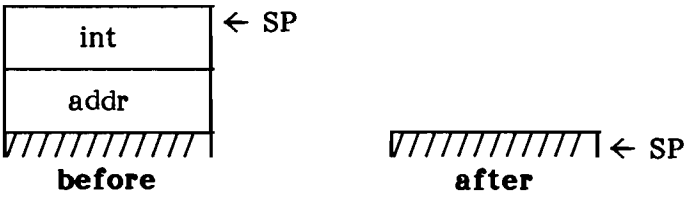
Description:

ENABLE reenables p-machine events that have been disabled by QUIET.

ATTACH (SEMAPHORE, VECTOR)

Procedure: 29

Stack:



Description:

TOS is the number of a p-machine event vector. It must be in the range 0 through 63. TOS-1 is the address of a semaphore.

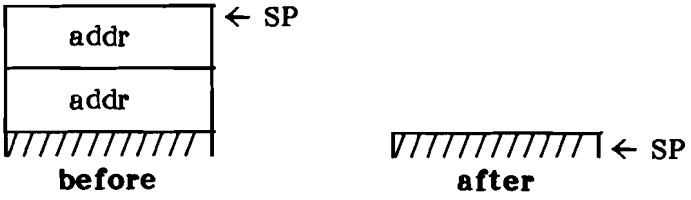
ATTACH associates the semaphore pointed to by TOS-1 with the vector TOS such that whenever the event TOS is recognized, the semaphore is signaled. If the semaphore pointer is NIL, vector TOS must be unattached from any sempahore it was formerly attached to. If TOS isn't in the range 0 through 63, no operation is performed.

MISCELLANEOUS PROCEDURES

TIME (HIWORD, LOWORD)

Procedure: 20

Stack:



Description:

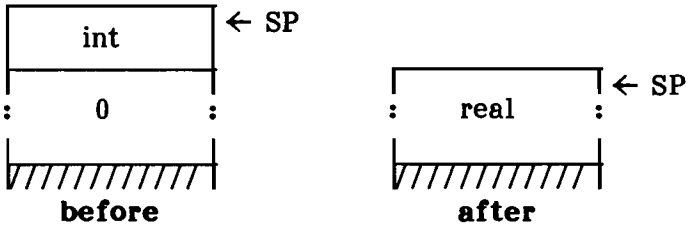
TOS is a pointer to where the high word of the time will be saved. TOS-1 is a pointer to where the low word of the time will be saved.

TIME saves the high and low words of the system clock (a 32-bit 60 Hz clock) in the indicated words.

POWEROFTEN (POWER): REAL

Procedure: 32

Stack:



Description:

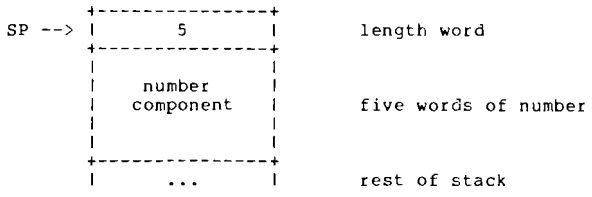
TOS is a positive integer power. POWEROFTEN returns the real value ten to the power of TOS. If TOS < 0 or TOS > the largest expressible power, a floating point execution error is issued.

LONG INTEGERS

The long integer data type is a nonscalar data type unique to UCSD Pascal. Long integers may be up to 36 decimal digits long. Although they lack some of the flexibility of scalar types, long integers allow operations on integers outside the range of most microcomputers (generally, -32768 to +32767 on a 16-bit machine). In computations, long integers act like real numbers; however, they act more like sets in the way they are implemented and in the way they are passed as parameters.

Number Format

On the stack (when used in calculations), long integers are of variable length, and consist of a length word followed by a number component. An integer five words long that is on the top of stack looks like this:



Every long integer on the stack has this format, regardless of the processor. However, the actual number encoding varies from processor to processor. The number format for different processors is given later in this section.

When a long integer is assigned to a variable, or stored in a file on disk, only the number component is present. The length word is present only when the number is on the stack. Each long integer variable is allocated a fixed number of words. When a long integer is assigned to a variable, the number must be coerced to the storage size of the variable. If this can't be done, an integer overflow execution error occurs.

The storage size for long integers is from two to ten words, based on the number of digits specified in the declaration statement. The following table shows the allocation size for each declared size.

digits	size (words)
1..4	2
5..8	3
9..12	4
13..16	5
17..20	6
21..24	7
25..28	8
29..32	9
33..36	10

The declaration size reflects the approximate number of digits that may be stored in the number. Since long integer formats vary from processor to processor, more digits than the declared number of digits may sometimes be stored in a long integer variable. As a result, the overflow value for a long integer may vary from processor to processor. The fact that more digits than the declared size may be stored in a long integer variable shouldn't be relied upon, since it would reduce the portability of a program. The number of digits specified in the declaration of a long integer should be treated as the maximum number of digits that the number will ever hold.

Long Integer Constants

Long integer constants are constructed at run-time by code generated by the compiler. This code builds each constant by doing a series of calculations on integers, thus sidestepping the problems associated with the fact that long integer formats are processor-dependent.

Example 1. To build the long integer constant 12, the compiler generates code to do the following:

```
CVT(12)
```

where 12 is an integer constant, and CVT is the routine to convert an integer to a long integer.

Example 2. To build the long integer constant 235543, the compiler generates code to do the following:

```
CVT(23554)*CVT(10) + CVT(3)
```

Example 3. To build -8733442, the compiler generates code to do the following:

```
-( CVT(8733)*CVT(1000) + CVT(442) )
```

Here is a listing of the actual p-code generated for the last example. The long integer routines called to do each operation are described in detail later.

```
42(02A):   LDCI      8733            811D22
45(02D):   SLDC        18            12          CVT
46(02E):   SCXG    LONGOPS     2      7202
48(030):   LDCI      1000            81E803
51(033):   SLDC        18            12          CVT
52(034):   SCXG    LONGOPS     2      7202
54(036):   SLDC         8            08          *
55(037):   SCXG    LONGOPS     2      7202
57(039):   LDCI       442            81BA01
60(03C):   SLDC        18            12          CVT
61(03D):   SCXG    LONGOPS     2      7202
63(03F):   SLDC         2            02          +
64(040):   SCXG    LONGOPS     2      7202
66(042):   SLDC         6            06          NEG
67(043):   SCXG    LONGOPS     2      7202
69(045):   SLDC        10            0A
```

LONGOPS Routines

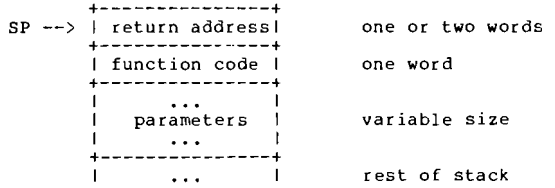
LONGOPS is the Pascal UNIT that implements the long integer functions. LONGOPS contains three procedures: FREADDEC reads a long integer; FWRITEDEC writes a long integer; and DECOPS performs the long integer arithmetic functions.

Although LONGOPS isn't part of the p-machine (it is in SYSTEM.LIBRARY), it isn't a normal Pascal UNIT either. Normally, a Pascal procedure or function must have fixed size parameters, where the parameter size is known at compile-time. There is one procedure in LONGOPS (DECOPS) that takes variable size parameters. One way to view this is that each call to DECOPS is like the execution of a single p-code in the PME. Different functions of DECOPS take different parameters and return different results, just as different p-codes do. In fact, DECOPS performs functions very similar to the set p-codes in the PME.

DECOPS Routines

DECOPS is an external (assembly language) procedure in UNIT LONGOPS that performs the long integer functions.

Parameters are passed to DECOPS on the stack. On every call, the stack looks as follows:



The return address is the standard return information for any assembly-language routine. Refer to the assembler documentation for a description of this information. (The return address isn't always on top of the stack. The HP-87, for example, uses a separate stack.)

The function code is a word that describes the function that is to be performed. The actions performed by each function are discussed below, along with the numeric value of the associated function code. Function codes are even integers between 0 and 20. Even integers are used to facilitate jumping indirectly through a word array of addresses.

The parameters vary for each long integer function. The parameter requirements for each routine are included in the description of the routine.

Usually, DECOPS returns to the PME as a normal assembly-language routine does. However, in certain error conditions, DECOPS must return to a different location in the PME so that an execution error may be recognized. This method of returning is usually via the .INTERP jump vector, but there are exceptions to this.

Below are the descriptions of each routine in DECOPS. The first line of each description contains the function code, followed by a descriptive name of the function. Next are descriptions of the parameters and the return value(s). Finally, there is a detailed description of the function, including any error conditions that should be recognized.

In the parameter and return value descriptions, the top of stack (TOS) is on the left, and items deeper on the stack are on the right. Here are the abbreviations used in the descriptions:

```
LI          Long Integer.  A variable-length long
              integer, containing a length word.
ALI         Adjusted Long Integer.  A fixed length
              long integer that does not contain a
              length word.
W           Word.  A 16-bit quantity.
B           Boolean.  A boolean quantity.  1=TRUE,
              0=FALSE.
```

0 — Adjust

parameters: <W> <LI>
return val: <ALI>

Adjusts <LI> into an adjusted long integer <ALI> suitable for assignment to a variable. It does this by stripping off the size word from <LI>, then expanding or contracting <LI> until it is <W> words in length. If a contraction can't be done because of overflow, an integer overflow execution error occurs.

2 — Add

parameters: <LI 2> <LI 1>
return val: <LI 3>

Adds <LI 1> and <LI 2>, placing the result on the stack as <LI 3>. If the result <LI 3> has more than 36 digits, an integer overflow execution error may occur.

4 — Subtract

parameters: <LI 2> <LI 1>
return val: <LI 3>

Subtracts <LI 2> from <LI 1>, placing the result on the stack as <LI 3>. If the result <LI 3> has more than 36 digits, an integer overflow execution error may occur.

6 — Negate

parameters: <LI 1>
return val: <LI 2>

Negates <LI 1>, placing the result on the stack as <LI 2>.

8 — Multiply

parameters: <LI 2> <LI 1>
return val: <LI 3>

Multiplies <LI 1> and <LI 2>, placing the result on the stack as <LI 3>. If the result <LI 3> has more than 36 digits, an integer overflow execution error may occur.

10 — Divide

parameters: <LI 2> <LI 1>
return val: <LI 3>

Divides <LI 1> by <LI 2>, placing the result on the stack as <LI 3>. If the result <LI 3> has more than 36 digits, an integer overflow execution error may occur. If <LI 2> is zero, a divide-by-zero execution error occurs.

12 — Long Integer to String

parameters: <W 1> <W 2> <LI>
return val:

Converts <LI> into a string, placing the result at the location pointed to by <W 2>. <W 1> is the maximum length of the string. If <LI> requires more than <W 1> characters to represent, a string overflow execution error occurs.

14 — TOS-1 Integer to Long Integer

parameters: <LI 1> <W>
return val: <LI 1> <LI 2>

Converts the integer at TOS-1 <W> into a long integer <LI 2>, leaving the long integer at TOS <LI 1> unchanged.

16 — Compare

parameters: <W> <LI 2> <LI 1>
return val: <B>

value of <W>	comparison
0	less than
1	less than or equal
2	greater than or equal
3	greater than
4	not equal
5	equal

Performs the comparison encoded in <W> of the long integers <LI 1> and <LI 2>, placing the boolean result <B> on the stack. In all cases, <LI 1> is compared to <LI 2>, in that order.

18 — TOS Integer to Long Integer

parameters: <W>
return val: <LI>

Converts the integer at TOS <W> into a long integer <LI>.

20 — Long Integer to Integer

parameters: <LI>
return val: <W>

Converts a long integer <LI> into an integer <W>. If the conversion can't be made (<LI> isn't in the range -32768..32767), an integer overflow execution error occurs.

Processor-Specific Information

8086/8088/LSI-11/6809/9900

These processors use a two's complement format that is stored in natural byte order with the most significant bits appearing at the lowest byte address. The most significant bit represents the sign of the number; a 0 means positive, and a 1 means negative.

```
Examples (hexadecimal):

0000 0000 6C33      is 27699
FFFF FFF6           is -10
0000 0000           is 0
```

68000

The 68000 uses a sign-magnitude Binary-Coded Decimal (BCD) format with the first word a sign word. The magnitude part of the long integer is stored in natural byte order, with the most significant digits in the byte with the lowest address, and the number is right-justified within the field. In the sign word, 0 means positive, and FFFF means negative.

```
Examples:

00 00  00 02 76 99   is 27699
FF FF  00 10         is -10
00 00  00 00         is 0
FF FF  00 00         is also 0
```

Z80/8080/6502

These processors have the same format as the 68000 except that only the least significant byte of the sign word is used (that is, 0000 means positive, and FF00 means negative).

```
Examples:

0000 00 02 76 99     is 27699
FF00 00 10           is -10
0000 00 00           is 0
```

HP-87

The HP-87 uses a ten's complement BCD format with the least significant digit appearing at the lowest address. A 9 in the most significant digit means negative, a 0 means postive.

```
Examples (least significant digit first):

99 67 20 00          is 27699
09 99 99 99          is -10
00 00 00 00          is 0
```

Low-Level I/O

CHAPTER 4

LOW-LEVEL I/O

THE I/O SUBSYSTEM

Besides emulating the p-machine, each PME must contain some native code to perform certain time-critical operations and deal with hardware dependencies such as I/O devices. The body of code that is not devoted to emulating p-code is called the Run-Time Support Package (RSP). The portion of the RSP that is responsible for I/O is called the RSP/IO.

To make the system as portable as possible, the RSP/IO is machine-independent, except for a portion called the Basic Input/Output Subsystem (BIOS). The BIOS must vary depending on the hardware in use, but the interface between the BIOS and the RSP/IO is standard—calls to routines in the BIOS are clearly defined.

Thus, we have the I/O hierarchy shown in Figure 4-1, the user's I/O calls (that is, READLN, WRITELN) are mapped by the compiler and operating system into calls to the RSP (that is, UNITREAD, UNITWRITE). The RSP/IO itself calls the BIOS which controls the actual device operations. It is important for the reader to recognize that here we are discussing a synchronous I/O system. In other words, when an I/O request has been initiated by your program, control doesn't return to that program until the I/O operation is completed.

This chapter describes the behavior and interfaces of the RSP/IO and BIOS. The SBIOS (Simplified BIOS) is described in the Adaptable System Installation Manual. The easiest way to describe its relation to the BIOS and RSP/IO is to sketch the history of I/O support within the p-System.

The first implementation was for the PDP-11, which has well-established standard interfaces to peripheral devices (regardless of manufacturer). In this environment, there was no need for I/O adaptation.

When the p-System was adapted to the 8080 and Z80, the p-System used CP/M BIOS run-time to perform low-level I/O. As adaptations for additional processors (that is, the 9900 and 6502) were begun, however, it became clear that the p-System needed some analog to the CP/M BIOS. It was at this point that the p-System BIOS, essentially as described in this chapter, was created and standardized.

The final step in this I/O development took place at SofTech Microsystems, where it was realized that:

1. The BIOS definition didn't address the problem of standardizing bootstrap mechanisms; and

2. Implementing a BIOS was a difficult task, and virtually required the use of an already running p-System.

The adaptable system was created to address these problems. The SBIOS is as simple a hardware interface as possible. It is called from a unit of "interface code" that accepts BIOS-style calls and emits SBIOS routine calls. This interface code allows the PME/SBIOS interface to be simpler than the BIOS interface. The RSP/IO is essentially unchanged.

The adaptable system also addresses the bootstrap problem by defining a hierarchy of bootstrap components, only some of which need to be implemented by the user installing a p-System.

A user who has access to a running p-System and the source code for the PME and SBIOS interface code may wish to implement a BIOS-level I/O interface. This is potentially more efficient than an SBIOS-level adaptation, since the more elaborate BIOS interface allows the implementor to take advantage of such performance characteristics as Direct Memory Access (DMA) support in the disk interface.

Both BIOS and SBIOS I/O interfaces have been created as the system was adapted to new environments.

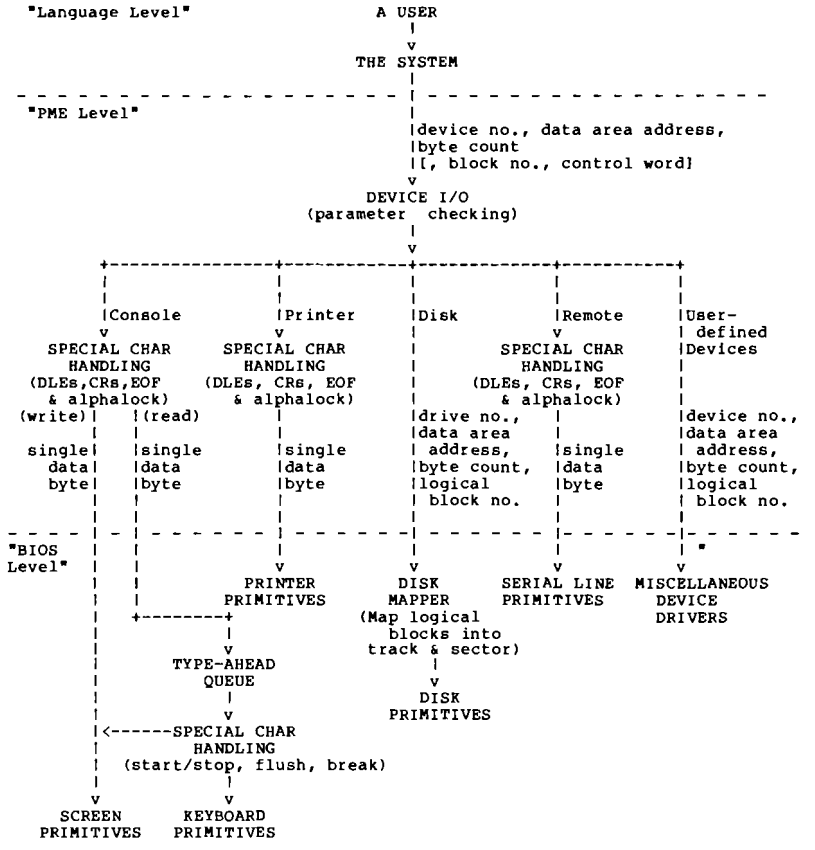


Figure 4-1. I/O Subsystem Hierarchy

DEVICE I/O ROUTINES

As mentioned above, all language-level I/O requests are eventually mapped by the compiler and operating system into calls to a group of intrinsic routines known as the device I/O routines. The programmer may call the device routines directly, or may use the standard I/O syntax of the language in use. The exact details of how this mapping is accomplished don't concern us here. The device I/O routines aren't written in Pascal, but are the native code procedures that constitute the RSP/IO.

Throughout this chapter, it is assumed that all I/O support at or below the device I/O level is implemented in assembly language. If p-code is the native language of the host processor, these routines may be implemented in Pascal.

The RSP/IO routines are implemented and accessed as routines of the operating system's unit KERNEL. KERNEL is accessible as segment 1 of every compilation unit. The actual code for the routines may reside in the PME itself, instead of in KERNEL.

Calling the RSP/IO

If you make direct calls to device I/O routines, they look like any other intrinsic routine. If they actually were declared in Pascal, the declarations would have the following format (allowing a few illegitimate constructs such as optional parameters and variable-length arrays):

```
PROCEDURE UNITREAD( UNITNUMBER : INTEGER;
        VAR DATAAREA : PACKED ARRAY [0..BYTESTOTRANSFER-1]
                                    OF 0..255;
             BYTESTOTRANSFER : INTEGER
             [; LOGICALBLOCK : INTEGER]
             [; CONTROL : INTEGER] );

   PROCEDURE UNITWRITE( <same as for UNITREAD> );

   FUNCTION UNITBUSY( UNITNUMBER : INTEGER ) : BOOLEAN;

   PROCEDURE UNITWAIT( UNITNUMBER : INTEGER );

   PROCEDURE UNITCLEAR( UNITNUMBER : INTEGER );

   PROCEDURE UNITSTATUS( UNITNUMBER : INTEGER;
        VAR STATUSWORDS : ARRAY [0..29] OF INTEGER;
             CONTROL : INTEGER );
```

Remember that no such declarations actually exist in the system. They are intended to model the parameters passed and returned by the native code RSP/IO routines.

Devices and Device Numbers

Each device is referred to in the system by a given number. The formal parameter UNITNUMBER in the declarations above determines which physical device the operation is intended for. Thus, the device I/O routines are device-transparent to the Pascal programmer; the same procedure will handle any physical device. Figure 4-2 is a list of the predefined device numbers associated with each physical device. The meaning of the other parameters is discussed later in this chapter.

Device Number	Volume Name
0	<Reserved for the system>
1	CONSOLE
2	SYSTERM
3	<Reserved for the system>
4	disk0
5	disk1
6	PRINTER
7	REMIN
8	REMOUT
9	disk2
10	disk3
11	disk4
12	disk5
13 - 127	Additional disks, subsidiary volumes, and user-defined serial devices

Figure 4-2. Device Numbers

User-Defined Devices

The system reserves all device numbers above 127 for user-defined devices. They have no preassigned names, but can be accessed through the UNIT intrinsics as can devices with preassigned numbers.

CONTROL Parameters

The CONTROL parameter to UNITREAD, UNITWRITE, and UNITSTATUS is a word used to pass special information to the RSP/IO and BIOS regarding the handling of the I/O request. The formats of the CONTROL words are shown in Figures 4-3 and 4-4. ("Set" equals 1; "reset" equals 0.)

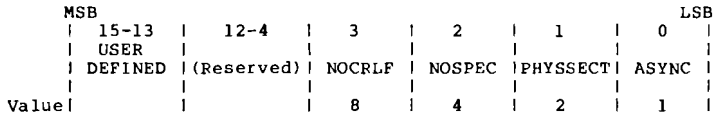


Figure 4-3. CONTROL Word Format for UNITREAD and UNITWRITE

Bit 0 ASYNC — Set implies asynchronous I/O request. Reset implies synchronous I/O request. (This bit should always be reset.)

Bit 1 PHYSSECT — Set implies "physical sector mode" for disk I/O. Reset implies "logical block mode" for disk I/O.

Bit 2 NOSPEC — Set implies "no special character handling." Reset implies "special character handling."

Bit 3 NOCRLF — Set implies that no line-feeds (LFs) are to be appended to carriage returns (CRs) during nondisk I/O. Reset implies LFs are to be appended to CRs during nondisk I/O.

Bits 4-12 — Reserved for future expansion.

Bits 13-15 — User-defined functions.

The default setting for all these bits is reset (0).

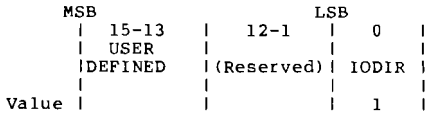


Figure 4-4. CONTROL Word Format for UNITSTATUS

Bit 0 IODIR — Set implies the status of the input channel is to be returned. Reset implies the status of the output channel is to be returned.

Bits 1-12 — Reserved for future expansion.

Bits 13-15 — User-defined functions.

IORESULT and Completion Codes

At times, an I/O request will terminate abnormally. To handle error conditions, a program may use the intrinsic IORESULT. The integer value returned by IORESULT describes the status of the last I/O request.

Each call to UNITREAD, UNITWRITE, UNITCLEAR, or UNITSTATUS causes a "completion code" to be set in the SYSCOM data area (SYSCOM, for SYStem COMmunication area, is conventionally the only data space that may be directly accessed by both the operating system and the PME). Programmers may test the completion code by using IORESULT.

The standard completion codes are given in Figure 4-5 below.

Code	Meaning
0	No error
1	Bad block, CRC error (parity)
2	Bad device number
3	Illegal I/O request
4	Data-com timeout
5	Volume is no longer on-line
6	File is no longer in directory
7	Illegal file name
8	No room; insufficient space on disk
9	No such volume on-line
10	No such file name in directory
11	Duplicate file
12	Not closed: attempt to open an open file
13	Not open: attempt to access a closed file
14	Bad format: error reading real or integer
15	Ring buffer overflow
16	Write attempt to protected disk
17	Illegal block number
18	Illegal buffer address
19	Bad text file size
20 - 127	Reserved for future expansion

Codes 128 through 255 are available for non-predefined, device-dependent errors.

Figure 4-5. I/O Completion Codes

Logical Disk Structure

The system views a disk as a zero-based linear array of 512-byte logical blocks. All disks in the system have this logical structure, regardless of their physical format. The physical allocation units of a disk are commonly known as sectors; these may vary widely from one model of drive to another. The BIOS is responsible for mapping the logical structure of a system disk onto the physical structure of the device; that is, mapping logical blocks onto physical sectors.

Physical Sector Addressing Mode

To provide enhanced flexibility for systems programming at a machine-specific level, a mechanism has been provided for directly accessing the physical sectors of the disk. When the PHYSSECT bit (bit 1, value 2) of the CONTROL word is set on a call to UNITREAD or UNITWRITE involving a disk unit, the I/O is performed in physical sector mode. This has the following effects:

1. The parameter LOGICALBLOCK is interpreted by the BIOS as the physical sector number (PSN). (In the future, this may become the least significant 15 or 16 bits of the PSN.)

2. The parameter BYTESTOTRANSFER must be 0. (In the future, this may become the most significant 16 bits of the PSN.)

Physical Sector Numbers

Typically, the physical sectors of a disk are addressed by specifying both track and sector numbers. That is, the disk is viewed as an array of tracks where each track is an array of sectors. If this data structure were declared in Pascal, it would look like this:

```
type
  BYTE = 0..255;
  SECTOR = array [0..(BYTESperSECTOR-1)] of BYTE;
  TRACK = array [1..SECTORSperTRACK] of SECTOR;
  DISK = array [0..(TRACKSperDISK-1)] of TRACK;
```

NOTE: Here you should be using the convention that track numbers are zero-based but sector numbers start from one.

You can convert the type DISK into a linear array of SECTOR as follows:

```
type
  DISK = array [0..(TRACKSperDISK*SECTORSperTRACK)-1]  of SECTOR;
```

You can use this linear representation for addressing the disk by PSN. The relations between the PSN, and track and sector numbers are:

```
PSN = (TRACKNUMBER*SECTORSperTRACK) + SECTORNUMBER-1;
 TRACKNUMBER = PSN div SECTORSperTRACK;
 SECTORNUMBER = (PSN mod SECTORSperTRACK) + 1;
```

Physical Sector Size

Any physical sector size may be accommodated. An I/O request in physical sector mode simply causes a full sector to be transferred. The programmer is responsible for ensuring that the data area is at least large enough for one physical sector.

Programs written using physical sector mode aren't expected to be portable to different disk hardware without some modification.

THE RSP

This section details the design and operation of the Input/Output portion of the Run-Time Support Package (RSP/IO). While the design itself is processor- and hardware-independent, it is intended to be realized in native code. Thus, the final product will be processor-specific but still independent of the exact peripherals used.

Calling Mechanisms

The following section discusses how each routine in the RSP/IO is called from the Pascal level (or the level of another compiled language). The level of detail is intended to be such that an implementor of the RSP will know how to pop parameters off the stack when the RSP is called, and how the stack should look when the RSP returns.

UNITREAD and UNITWRITE

```
PROCEDURE UNITREAD( UNITNUMBER : INTEGER;
      VAR DATAAREA : PACKED ARRAY [0..BYTESTOTRANSFER-1]
                              OF 0..255'
          BYTESTOTRANSFER : INTEGER
          [; LOGICALBLOCK :INTEGER]
          [; CONTROL : INTEGER]  );

PROCEDURE UNITWRITE( <same as for UNITREAD> );
```

Parameter Description

UNITNUMBER was discussed under "Devices and Device Numbers" above.

DATAAREA is the programmer's buffer area for transferring data. Describing it as a VAR parameter signifies that UNITREAD and UNITWRITE are passed a pointer to the start of the data area. This pointer is actually represented as an address couple consisting of a word base and a byte offset. On processors which use byte addressing, the effective address is computed by simply adding the base and the offset, since both quantities are in bytes. For processors using word addressing, the effective address is computed by indexing byte-wise from the base address (always toward higher locations). Generally, the address of the start of the data area may or may not be on a word boundary. In the case of disk units, however, it is only defined if it is on a word boundary; that is, a Pascal programmer must not allow actual parameters with odd numbered indexes (like A[3]) to occur when transferring to or from the disk. The reason for this inconsistency is to avoid restricting disk data to being moved byte by byte.

The third item in the parameter list, BYTESTOTRANSFER, contains the number of bytes to move between your data area and the physical unit.

Two optional parameters follow for UNITREAD and UNITWRITE: LOGICALBLOCK and CONTROL. These parameters are optional for the Pascal programmer; the compiler will assign a default value of zero to both LOGICALBLOCK is relevant only for disk reads or writes, as discussed in "Logical Disk Structure," above. It specifies the Pascal logical block to be accessed. The CONTROL

word has been discussed above in "Control Parameters."

Parameter Stack Format

UNITREAD and UNITWRITE receive their parameters on the evaluation stack in the following order (each box represents a 16-bit quantity):

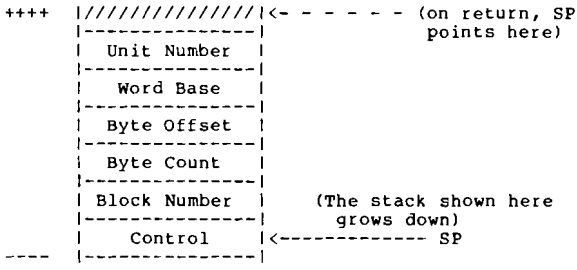


Figure 4-6. Stack State on Entering UNITREAD or UNITWRITE

Like ordinary Pascal procedures, these RSP routines pop their parameters from the stack when they are finished.

UNITBUSY

```
FUNCTION UNITBUSY( UNITNUMBER : INTEGER ) : BOOLEAN
```

The UNITBUSY function has meaning only in an asynchronous environment and thus will always return FALSE (0) for this synchronous specification. The use of the stack is illustrated in Figure 4-7.

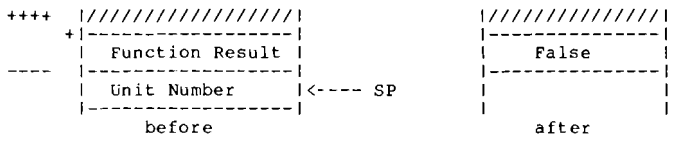


Figure 4-7. Stack State Before and After UNITBUSY

UNITWAIT

```
PROCEDURE UNITWAIT( UNITNUMBER : INTEGER );
```

Like UNITBUSY, UNITWAIT is only useful in an asynchronous environment. In a synchronous system, as described here, UNITWAIT becomes essentially a no-op, since no unit will have a I/O request pending. A single parameter is on the top of stack when the procedure is called and is popped off before the procedure returns. The use of the stack is illustrated in Figure 4-8.

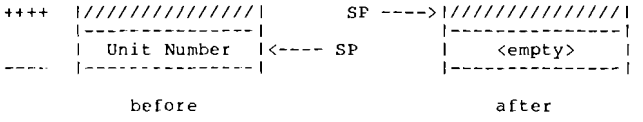


Figure 4-8. Stack State Before and After UNITWAIT and UNITCLEAR

UNITCLEAR

```
PROCEDURE UNITCLEAR( UNITNUMBER : INTEGER );
```

The purpose of UNITCLEAR is to restore the specified device to its "initial" state. At the RSP level, this means clearing any state flags pertaining to the specified device. The "initial" state for each device at the BIOS level is defined in the section, "BIOS Responsibilities," below. The stack format is identical to that of UNITWAIT (see Figure 4-8 above).

UNITSTATUS

```
PROCEDURE UNITSTATUS( UNITNUMBER : INTEGER;
     VAR STATUSWORDS : ARRAY [0..29] OF INTEGER;
         CONTROL : INTEGER  );
```

The purpose of UNITSTATUS is to acquire various device-dependent information from the specified UNIT. The procedure is passed a pointer to a status record (whose length is a maximum of 30 words) into which the status words are sequentially stored (note that users may define words starting at word 29 and allocating toward word 0, to allow for the system's use of the first words of the record) and a CONTROL word.

UNITSTATUS receives its parameters on the evaluation stack in the following order (each box represents a 16-bit quantity):

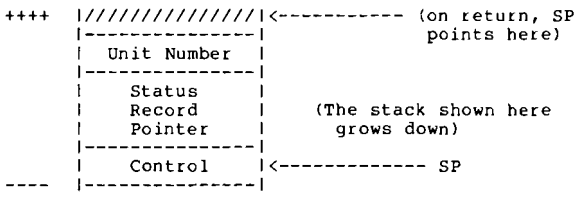


Figure 4-9. Stack State Before and After UNITSTATUS

RSP Responsibilities

This section details the processing to be performed by the RSP/IO. The primary function of the RSP/IO is to manage calls to the BIOS. Secondarily, the RSP/IO is responsible for handling certain special functions, which are described here.

Special Character Output Handling

Output to the printer, console, remote, or serial units must properly handle blank compression codes and carriage returns.

Blank Compression Code (DLE's)

The system supports text files that contain a two-byte blank compression code (only at the beginning of a line). It is the responsibility of the RSP/IO to decode the blank compression code and send an appropriate number of blanks. The first byte is an ASCII DLE (decimal 16), which signals that the next byte contains the excess-32 number of blanks to insert (that is, it should be interpreted as being the <number of blanks to be sent>+32). Therefore, the next byte following the DLE should be processed by subtracting 32 from its value and sending that number of blanks. Note that negative results, obviously in error, are translated to a value of zero. Also note that the blank-count byte may not be the next input byte processed, because of device switching. This, therefore, requires the maintenance of a flag for each device to indicate that it is currently processing a DLE. The DLE character and the blank-count byte aren't normally sent to the device (see the paragraph, "NOSPEC Bit," below).

Carriage Return — Line Feed

Text files contain ASCII CR's (decimal 13) at the end of lines. We define this character as meaning "New Line"; that is, a carriage return followed by a line feed. Thus, it is the responsibility of the RSP/IO to send an ASCII LF (decimal 10) after sending each CR.

NOCRLF Bit

When bit 3 (value 8) of the CONTROL parameter is set, the special handling accorded a CR is turned off; that is, a LF isn't automatically appended, and the CR is sent out like another character.

Special Character Input Handling

There are several characters which should receive special treatment when received from the console, the printer, the remote or the serial devices, in a complete implementation of this I/O system. All but two of them, however, are handled by the BIOS. Those which are handled in the RSP/IO are the EOF and ALPHALOCK characters.

EOF Character

The EOF character, when received from the console, printer or remote devices, signals that the end-of-the-file has been reached on that particular unit. Rather than being a fixed ASCII code, EOF is a "soft character." That is, the exact character code which will be interpreted as end-of-file may be changed during system execution by the Pascal user. Further discussion of the soft characters used by the I/O subsystem may be found in the section, "Character Codes," below. The EOF character is in the SYSCOM data area and must be accessed by the RSP/IO to determine what character to look for. When the EOF character is found in the input stream, the action to be taken depends somewhat upon which device was referenced. If you are reading from UNIT 1 (CONSOLE:), then the rest of your buffer, starting at the current position, is packed with nulls (decimal 0). For UNIT 2 (SYSTERM:), the printer and the remote, the EOF character is put into your buffer. In all cases, no further characters are transferred to the buffer and control returns immediately.

ALPHALOCK Character

The ALPHALOCK character, when received from a device by the RSP/IO, signals a default case change for all alphabetic characters. All lowercase alphabetic characters (that is, 'a' to 'z') received after the ALPHALOCK character will be converted to uppercase. Receipt of another ALPHALOCK character will cause the case to revert back to nonconverting mode (the default mode). As for DLE handling described above, a flag, for each device to indicate that it is currently in the ALPHALOCK state, should be maintained to ensure proper handling when devices are switched. The ALPHALOCK character is not normally returned in the buffer (see the paragraph, "NOSPEC Bit" below).

Other Characters

The remaining special input characters—BREAK, START/STOP, FLUSH, and CHARMASK—are used only for input from the console, not from the printer or remote devices. They are handled by the BIOS and are described under "Input Options," below.

NOSPEC Bit

When bit 2 (value 4) of the CONTROL parameter is set, the special handling accorded DLE's, and the EOF and ALPHALOCK sensing functions described above are turned off. These characters are then transferred as are any other characters. The BIOS functions aren't affected.

Translation for Subsidiary Volumes

The RSP is also responsible for converting disk read/write calls to subsidiary volumes into disk calls for accessing the physical disk drive (instead of the virtual subsidiary volume).

The SYSCOM area contains a pointer to the unitable which contains a record for each p-System unit. Each record for storage devices contains a block offset and a physical disk unit number. The RSP must look up calls to subsidiary volumes and give the physcial disk number and correct block number when the call to the BIOS is made.

The subsidiary volume requires some special checking in the RSP. The following Pascal code fragment describes how the RSP handles subsidiary volumes.

```
if unit# in [syscom^.subsidstart..
             syscom^.subsidstart + syscom^.unitdivision.subsidmax -1]
 then {translate svol parameters}
      with syscom^.unitable^[unit#] do
        begin
          if ueovblk=0 then return_ioresult( 9 );
          if block# >= ueovblk then return_ioresult( 17 );
          block# := block# + ublkoff;
          unit#  := uphysvol;
        end
 else {no translation for other volumes needed};
```

BIOS

As explained above, the BIOS is responsible for providing the actual access to I/O devices. Both the design and implementation of the BIOS are specific to a given processor and I/O configuration. In this section, we will attempt to specify the nature of the BIOS in detail sufficient for an experienced programmer to write the code for a given processor and a set of peripherals.

The general scheme discussed below uses vectors from the RSP/IO to the BIOS subroutines for reading, writing, initializing and controlling, and answering status requests. The exact vector scheme and means of passing parameters must be worked out separately for each processor. Arrangements that have already been worked out for certain processors are illustrated in "Processor-Specific BIOS Calls," below.

Design Goals

The speed at which BIOS code executes is fairly insignificant compared to the (slow) speed of the I/O devices that it handles. When peripherals are changed, which may occur frequently, it often proves that only minor changes need to be made to an existing BIOS to service the new hardware. Also, since the BIOS always resides in main memory, each byte it occupies means one byte less is available to the programmer. For these reasons, we suggest that major design goals (assuming correctness!) be: (1) compactness; and (2) clarity.

Like the rest of the PME, the BIOS should be ROM-able. Obviously, it will also require access to some RAM. The addresses that the BIOS references should be specified in the assembly code by equates, so that it is a simple matter to change them and reassemble the BIOS whenever there is a change in the I/O ports or the memory configuration.

Completion Codes

All read, write, initialization and control, and status calls to the BIOS must return a byte to the RSP that contains status information about the I/O request just serviced. The value of this byte is the "completion code" discussed in the section, "IORESULT and Completion Codes," above. Most of the standard completion codes aren't relevant to the BIOS—they are returned by the operating system for file errors and the like. The following standard errors can be returned by the BIOS:

```
 0   No error
 1   CRC error
 2   Illegal device number
 3   Illegal operation on device
 4   Undefined hardware error
 9   Device not on line
15   Ring buffer overflow
16   Write protect; write attempt to protected disk
17   Illegal block number
18   Illegal buffer address
```

All other errors are considered hardware-dependent. For these, the BIOS should return codes in the range 128 through 255. The selection of appropriate codes is left to the BIOS writer.

NOTE: Any predefined devices not implemented must arrange to return a completion code of 9 ("device not on-line") when an attempt is made to initialize or use them.

Any user-defined devices not implemented should return a completion code of 2 ("illegal device number") when an attempt is made to access them.

Calling Mechanisms

In this section, we discuss the parameters required in the BIOS calls for each device. Each device has four BIOS calls associated with it: READ, WRITE, CONTROL, and STATUS. Each device has varying needs for information associated with these functions. Remember that all calls must return a completion-code byte. The BIOS calling requirements are summarized below.

Console

Only one parameter is needed for reading and writing—the data byte to be transferred. The status request requires two parameters: the CONTROL word and the pointer to the status record. For initialization and control of the console, the BIOS requires a number of special control characters. These are provided by passing the BIOS console initialization routine a pointer to the base of the SYSCOM data area, and a pointer to a break-handler routine.

Printer

To read from and write to the printer, a single parameter is required—the byte that contains the data. To check the status, the CONTROL word and the pointer to the status record are required. For initialization and control, no parameters are needed.

Disks

Reading and writing with disk devices requires five parameters:

1. A starting logical block number as described above.

2. A count of the number of bytes to transfer (positive signed 16 bits; that is, 0 to 32K-1).

3. The address of the data area to transfer to or from.

4. A drive number (0 through n-1, given n drives. Currently n=6 is assumed).

5. The CONTROL parameter.

To check the status, the CONTROL word and a pointer to the status record are passed as parameters. For initialization and control, the drive number is passed.

Remote

The remote device requires a single parameter for reading and writing—a byte that contains the data being transferred. As with the devices just described, the status requires the CONTROL word and the pointer to the status record. Initialization and control of the remote device requires no parameters.

User-Defined Devices

Reading and writing with a user-defined device requires five parameters:

1. A starting logical block number as described above.

2. A count of the number of bytes to transfer (positive signed 16 bits, that is, 0 to 32K-1).

3. The address of the data area to transfer to or from.

4. A device number (this will be the same as UNITNUMBER).

5. The CONTROL parameter.

The native code in the BIOS may choose to ignore some of this information, of course.

When checking status, the CONTROL word, device number, and a pointer to the status record are passed. For initialization and control, the device number is passed. It is left to the device handler to determine the specific device from its device number.

Character Codes

The system assumes that the printer and console devices will support the use of printable ASCII characters and a few standard control codes (CR, LF, SP, NUL and BEL). The remaining control codes that may be useful (such as cursor positioning and screen erasure) are "soft" characters that may be changed by the user (using the utility SETUP) to meet the requirements of some particular hardware.

These soft characters, along with all other information that is entered using SETUP, are stored in the file *SYSTEM.MISCINFO. SYSTEM.MISCINFO is read into a portion of the global data area SYSCOM whenever the system is booted or reinitialized.

The reason for keeping this hardware-dependent information at such a high level to be able to change a terminal (as happens fairly often) without creating a new BIOS. One way to do this is to map logical control symbols into control codes that are recognized by the hardware.

Suppose, for example, that there is a predeclared procedure CURSORBACK which causes the cursor on a screen terminal to move left one column. Somewhere in the system, CURSORBACK must cause a control code to be sent to the terminal, which will cause the desired response; control-U, control-H, or an escape sequence. One way to do this would be for the compiler to emit a standard code which the BIOS then translates into whatever is correct for the current terminal. This has the disadvantage of requiring a new BIOS for every slightly different terminal. The approach which we have taken sees to it that the correct code is sent to the BIOS for the terminal that is currently on-line.

Since many devices can make use of 8-bit control codes, the system makes no assumptions about the relevance of the high-order bit, and transfers the whole byte unchanged. When using 7-bit ASCII, the value of the high-order bit is defined to be zero. This means that the BIOS must mask all characters from the console with the character mask in the SYSCOM, which is 127 (decimal) if 7-bit ASCII is being used.

The RSP sends both uppercase and lowercase characters to the BIOS. If a device can handle only uppercase characters, the BIOS must map lowercase into uppercase.

BIOS Responsibilities

Console

In the following discussion, the console device is assumed to be a CRT terminal.

Console Output Requirements

As a minimum, every console device should provide the following keyboard functions:

CR <carriage return> (hexadecimal 0D) — Moves the cursor to the beginning of the current line (column 0).

LF <line feed> (hexadecimal 0A) — Moves the cursor down one line while the column position remains the same. Starting from any but the last line on the screen, the contents of the screen should remain the same while the cursor moves downward. If the cursor is on the last line when the LF is issued, it should remain in the same position while the rest of the display scrolls upward one line and the bottom line clears.

BEL <bell> (hexadecimal 07) — If an audio signal is available, it will sound when the appropriate key is pressed. If one isn't available, the terminal will do nothing.

SP <space> (hexadecimal 20) — Writes a space at the current cursor position (erasing whatever is there) and advance the cursor position by one column. If the cursor is already at the last position in a line, the position of the cursor after the SP is undefined. (We prefer that the cursor remain in its prior position in this case. If the cursor is in the last column of the last line on the screen, not only is the position of the cursor undefined after the SP, but so is the state of the screen—maybe it scrolled and maybe it didn't. We prefer that the cursor remain where it was and that the screen not scroll.)

NUL <null> (00) — Causes a delay of the time required to write one character. The state of the console shouldn't change.

Printable Characters (hexadecimal 21-7E) — Same as the discussion for SP, except, of course, write the character.

NOTE: The effect of sending nonprintable characters other than those described above isn't defined here since it varies from terminal to terminal.

Console Output Options

The following set of cursor and screen functions should be provided if possible. The control characters or sequences of characters that perform these functions are left unspecified (they are soft characters). If a stand-alone ASCII terminal is connected to the host system, these functions may be provided by the terminal itself. In this case, all the BIOS need do is pass the appropriate control characters.

Reverse Line Feed: Moves the cursor to the next line higher on the screen without changing the column or the contents of the screen. If the cursor is already on the top line, the result is undefined. If possible, the screen should reverse-scroll in such a case, or if that isn't feasible, the cursor and screen should just remain as they were.

Non-Destructive Forward and Backward Space: Moves the cursor in the direction indicated without changing the contents of the screen (that is, moves it non-destructively). The position of the cursor is undefined if an attempt is made to move it beyond the beginning or the end of a line. The preferred result is that cursor and screen remain unchanged in such a case.

Cursor HOME: Moves the cursor to the upper left-hand corner of the screen without changing the contents of the screen.

Cursor X,Y Positioning: Moves the cursor to some absolutely determined row and column without disturbing the contents of the screen. The result is undefined if an attempt is made to move the cursor to a nonexistent position.

Erase to End of Screen: Erases from the cursor position to the end of the screen, leaving the cursor where it started and the other contents of the screen undisturbed.

Erase to End of Line: Erases from the cursor position to the end of the current line, leaving the cursor where it started and the rest of the screen undisturbed.

Console Input Requirements

Input from the console should not be echoed to the screen by the BIOS; this function is handled by RSP/IO. Keys which represent ASCII characters should generate 8-bit codes between 0 and 127. Other (non-ASCII, that is, special function) keys can generate codes between 128 and 255, if desired.

Console Input Options

If possible, we recommend that the console input BIOS be responsible for the following special functions.

START/STOP

The START/STOP character is used to control console output. When START/STOP (a soft character) is received, console output is suspended until: (1) another START/STOP character is received; (2) a FLUSH character is received; (3) the console BIOS is reinitialized; or (4) the BREAK character is received. The action to take in the last three cases is discussed below. Should another START/STOP character be received, the suspended activities should resume exactly as they left off. The chief benefit of this arrangement is that you can suspend output processes which are proceeding too fast (for example, a text file scrolling across the screen at 9600 baud). The suspension process takes place wholly within the BIOS, and requires no communication to the RSP. (Note that the START/STOP character is never returned to the RSP. The queueing of keyboard input, if implemented, should continue during the suspension.)

FLUSH

FLUSH is another soft control character. When FLUSH is typed, the console output BIOS discards all output characters (that is, doesn't display them) until: (1) FLUSH is entered again; (2) input is requested from the console BIOS; (3) the console BIOS is reinitialized; or (4) the BREAK character is received. The FLUSH character is never returned to the RSP. If FLUSH is received while a START/STOP suspension is pending, the suspension is canceled and FLUSH has its usual effect. This feature is useful when a long text file is being displayed on the console and you're tired of looking at it. If you push FLUSH, it terminates rather quickly. It is also useful when a process is generating console output that is irrelevant, but slows down the process. Note that FLUSH applies only to console output.

BREAK

When BREAK (also a soft character) is entered, the console input BIOS should check the state of the NOBREAK flag bit in the SYSCOM data area. If the NOBREAK flag is a 1, then the BREAK key should be ignored (the console input routine should go back to waiting for a character from the console). If the NOBREAK flag is a 0, then the BIOS should immediately give control to a special PME routine. The vector to this routine is passed at console initialization time. After execution of the BREAK routine, the BIOS should continue as before. The BREAK routine is responsible for notifying the PME that a BREAK should be executed before the next p-code is interpreted. (Note that the BREAK character is never returned to the RSP. Receipt of BREAK should terminate any START/STOP or FLUSH suspension pending.)

The system stores the NOBREAK boolean in the data area called SYSCOM. A pointer to SYSCOM is passed to the console initialization routine. The byte containing the NOBREAK boolean must be masked with 01000000 binary (40 hexadecimal) before examining the NOBREAK boolean, the other bits aren't necessarily zero.

Type-Ahead

When nonspecial characters (ones not described in the sections above) are received from the keyboard, and when a no read request is pending, they should be queued until the next read request. The next read request should remove the first character from the queue. When characters in excess of the maximum queue size are received, they should be ignored; the queue should remain intact. While a type-ahead of even 1 character is better than none at all, we recommend a minimum queue size of about 20 characters. If possible, the bell should be sounded for each character entered from the keyboard after no room remains in the queue.

Input Character Mask

In the p-System, prior to version IV.1, all characters input from the console were masked with 7F (hexadecimal) to clear the parity bit in bit 7. This changed, in version IV.1, to allow terminals (or keyboards) that use full 8-bit character codes to return them unmasked, and to continue to allow terminals that needed to have the parity bit cleared to work.

Every character read from the console should be ANDed with the CHAR_MASK byte found in the SYSCOM data area. This will be set with the SETUP utility to be either 7F or FF (hexadecimal) as needed. The masking should be done before checking for BREAK, START/STOP or FLUSH.

Initialization and Control

The initialization and control part of the console BIOS is responsible for the following tasks (and whatever else the BIOS implementor finds expedient):

SYSCOM Data Area: The system stores soft characters: START/STOP, FLUSH, BREAK and other special variables in the SYSCOM data area. These are variables that must be accessible from both the operating system and the low-level routines (PME, RSP, and BIOS). One parameter to the console initialization and control routine is a pointer to the start of the SYSCOM area. The SYSCOM is a packed record declared in the interface section of the unit KERNEL. Byte offsets within SYSCOM depend on the processor sex (low-byte or high-byte first). The offsets to variables used in the BIOS and RSP are (expressed as positive byte offsets):

	LSB first (decimal)			MSB first (decimal)			
	decimal	hex	octal	decimal	hex	octal	Usage
FLUSH -	83	53	123	82	52	122	BIOS
BREAK -	84	54	124	85	55	125	BIOS
STOP/START -	85	55	125	84	54	124	BIOS
CHARMASK -	92	5C	134	93	5D	135	BIOS
NOBREAK -	58	3A	72	59	3B	73	BIOS
EOF -	82	53	122	83	53	123	RSP
ALPHALOCK -	93	5D	135	92	5C	134	RSP

BREAK Vector: Another initialization and control parameter is the address of the PME routine which handles BREAK. The console initialization code is responsible for setting up a vector to this address via its private data area and calling this routine when the BREAK character is received.

Flags: Initialization should cause the START/STOP and FLUSH flags to be cleared (or whatever else may be required to return to normal).

Type-Ahead Queue: Initialization should cause any characters currently waiting in the type-ahead queue to be discarded.

Console Status

As described in "Control Parameters," at the beginning of this chapter, bit 0 (value 1) of the CONTROL word defines the direction of the status request. The request should return, in the first word of the status record, the number of characters currently queued for the direction specified. If some form of buffering is being used, this will simply be the number of characters in the buffer. If no buffering is implemented, the output status will always return 0, but the input status will return 1 if a character is waiting to be read, or 0 if none is waiting.

Printer

The printer is expected to be a line printer or other hard copy device. In actual practice, any ASCII display device may be used.

Printer Output Requirements

In order to serve the widest variety of hard copy devices, the RSP/IO doesn't buffer a line of text and send it all at once. Rather, it sends the printer BIOS a single character at a time. Many line printers must buffer a line and then print it all at once; if this is the case, it is the BIOS that must do so, in which case, the BIOS must recognize the end of a line (EOLN). EOLN is signalled by a certain character; the possibilities are listed below:

CR <carriage return> (hexadecimal 0D) — Print the line and return the carriage to the first column. An automatic line feed should not be done.

LF <line feed> (hexadecimal 0A) — In normal operation, the RSP/IO will only send an LF to the BIOS immediately after a CR. If the hardware allows a simple line feed to be performed (without a return), then this should be done. If a complete "new line" operation (that is, return and line feed) is the only way your printer can print a line, then do so at an LF—don't do anything about a CR.

FF <form feed> (hexadecimal 0C) — The printer should advance the paper to top-of-form, if possible, and perform a carriage return. If no such feature is available, the printer may execute a "new line" operation; that is, a return followed by a line feed.

Printer Input Requirements

There are no strict requirements for input from the printer device. If the printer device has the capability to transmit data, then the printer input BIOS should return all eight data bits unchanged. If not, then input shouldn't be allowed and should return completion code 3 ("illegal operation on device").

Printer Initialization and Control

Initialization of the printer device should make it ready to print at the beginning of a blank line. A "new line" (carriage return and line feed) operation may be in order here. Any characters that have been buffered but not printed are lost. The printer doesn't need to do a form feed each time it is initialized.

Printer Status

As described above, the number of characters buffered for the direction specified in the CONTROL word should be returned in the first status word. If the printer has no form of self-checking, return 0.

When returning output channel status the number of characters buffered has a special meaning. A zero returned (for number of characters buffered) means the printer is ready to receive a character, a non-zero value in interpreted as meaning the printer isn't ready to receive another character. The print spooler uses this to determine if it can send a character to the printer without hanging the system (in the background task) on a write to the printer.

Disk

Mapping Blocks on Physical Sectors

The disk device may be any type of disk drive (for example, floppy or hard disk). The actual sectoring arrangements of the disk are immaterial. The system addresses the disk in terms of consecutive logical blocks of 512 bytes each. A primary function of the disk BIOS, therefore, is to provide an appropriate mapping scheme into the actual (physical) sectors used on the disk. The sector interleaving algorithm should be optimal for the hardware.

The system makes no assumptions about the interleaving method used by the BIOS (except that it works!).

Bootstrap Location

While bootstrap schemes vary, typical implementations make use of a hardware (usually ROM) bootstrap to load and execute a primary software bootstrap which, in turn, loads and executes a secondary software bootstrap. The secondary bootstrap then loads the PME and operating system, performs required initializations, and starts the system.

To be accessible to the hardware bootstrap, the primary software bootstrap must reside at a location on the disk which is predetermined by the hardware vendor. Since these locations can vary widely, it is necessary that the system's requirements for a physical disk format be flexible in this regard.

The primary bootstrap area must not overlap disk data structures maintained by the system (chiefly the directory and the bootstrap itself).

Logical blocks 0 and 1 of each disk are usually reserved for bootstrap code (a total of 1024 bytes). This is the most convenient alternative.

If 1024 bytes aren't enough room, or if the interleaving format is unacceptable to the hardware bootstrap, the primary bootstrap area must be outside of the "Pascal disk." The Pascal logical blocks must be mapped onto the disk in such a way that the hardware-defined bootstrap area is inaccessible to the p-System as a logical block. (It will still be accessible in Physical Sector Mode, see above.)

For adaptable systems, full details about bootstrap locations and the mechanisms of booting may be found in the Adaptable System Installation Manual.

Physical Sector Mode

When bit 1 (value 2) of the CONTROL word is set, disk access should be performed in Physical Sector Mode, as described in the section, "Physical Sector Addressing Mode," above.

Disk Output Requirements

The disk device BIOS must transfer as many actual sectors as are needed to accommodate the data. To simplify a disk-write in which (BYTESTOTRANSFER) MOD 512 isn't equal to zero (for example, a block is partially written to), the remaining contents of the last block are undefined. This makes it possible to write as much of whatever garbage remains in the buffer; if that is most convenient, to fill up a whole sector. Figure 4-10 illustrates this situation. The language level is responsible for keeping track (in logical block numbers and byte counts) of where the good data is.

EXAMPLE:

```
Write to disk.

  Number of bytes to transfer = 1174
  Starting logical block number = 72
  Data area address = DATAAREA

     |                  |                  |     :            |
     |    Block 72      |    Block 73      |    Block 74      |
     |   (512 bytes)    |   (512 bytes)    | 150 :(362 bytes) |
     |                  |                  |bytes:            |
     |<-----------------data----------------->:<undefined>|
     |                  |                  |     :            |
     ^                                     ^     ^            ^
     |                                     |     |            |
     start of data area                    end of data area   |
                                                              |
                                           end of last block
```

Figure 4-10. State of Blocks on Disk After Being Written

Disk Input Requirements

On input from a disk device, it's not permissible to over-write the end of the assigned data area. Therefore, the BIOS is responsible for transferring no more than the number of bytes requested. One way to accomplish this is to buffer the last sector and then transfer only the requested part.

Disk Initialization and Control

Initialization of a disk device should bring it to a state in which it is ready to read or write from any given track or sector. For some drives with simple controllers, the head may need to be stepped to track 0 to facilitate the BIOS disk driver's remembering the current track. Any buffered data is lost.

Disk Status

Status requests from the disk will return the following words in the status record:

Word 1 — The number of bytes currently buffered for the direction specified in the CONTROL word, as described in the section, "Console Status," above. If no capability for checking is available, it should be set to 0.

Word 2 — The number of bytes per sector.

Word 3 — The number of sectors per track.

Word 4 — The number of tracks per disk.

Remote

This unit is intended to be an RS-232 serial line for supporting various types of communication. It is important that it transfer raw data without changing it in any way. All eight bits of the transferred byte should be considered significant. The transfer rate is usually set to 9600 baud.

Remote Output Requirements

As noted above, all 8-bits of the data byte should be transmitted. The remote BIOS driver is sent one byte at a time.

Remote Input Requirements

Input from a remote device should be buffered, if possible, by the scheme suggested in "Type-Ahead" section. As noted above, all eight data bits must be returned.

Remote Initialization and Control

Initialization of the remote device should bring it to a state in which it is ready to read or write.

Remote Status

The number of bytes buffered for the direction specified in the CONTROL word should be returned in the first status word, as described in the "Console Status" section, above. If no capability for checking is available, it should return 0.

User-Defined Devices

These devices are intended to allow you the freedom to implement devices not specifically defined in this document. The actual implementation is left entirely to you. The only requirement is that they return a completion code when finished and, if the UNITNUMBER isn't defined, that it return code 2 ("illegal unit number"). You should use device numbers starting from 128 (see "User-Defined Devices," above).

Special BIOS Calls

These functions are provided by the BIOS to make configuration-specific functions accessible to the PME. Although these functions aren't related to I/O, they are put into the BIOS as the repository for configuration-specific code.

As with all other routines in the BIOS, each should return a completion code.

System Output

System Output is reserved for future expansion and, at this time, should cause the system to HALT. (Note that HALT may actually cause a reboot on some (few) implementations.)

System Input

System Input is also reserved for future use, and like System Output, should cause a HALT.

System Initialization and Control

The System Initialization and Control BIOS routine should initialize such things as the clock (reset it to 0) and the interrupt system, if either is to be used.

System Status

The System Status BIOS routine should return the following information in the status record:

Word 1 — The address of the last word in accessible contiguous RAM memory; for example, on an 8080 system with 64K bytes of RAM, the last byte address may be 'FFFF', but the last word address is 'FFFE'.

Word 2 — The least significant part of the 32-bit word used by the system clock. If a clock isn't present, then this must be set to 0.

Word 3 — The most significant part of the 32-bit word used by the system clock. If a clock isn't present, then this must be set to 0.

NOTE: If a clock is used, the system assumes that the two words returned are representative of the time in 60ths of a second. It is the clock driver's responsibility to maintain the closest approximation to this time. The time is defined to be 0 at clock initialization. Currently, the CONTROL word is ignored.

BIOS CALLING CONVENTIONS

The following is a summary of the calling conventions described earlier. The processor-specific protocols for certain machines are shown in the following section. All calls to the BIOS return a completion code.

Entry Point	Parameters
CONSOLEREAD	single data byte
CONSOLEWRITE	single data byte
CONSOLECTRL	BREAK vector SYSCOM pointer
CONSOLESTAT	STATREC pointer CONTROL word
PRINTERREAD	single data byte
PRINTERWRITE	single data byte
PRINTERCTRL	(none)
PRINTERSTAT	STATREC pointer CONTROL word
DISKREAD	block number byte count data area address drive number CONTROL word
DISKWRITE	(same as DISKREAD)
DISKCTRL	drive number
DISKSTAT	drive number STATREC pointer CONTROL word
REMOTEREAD	single data byte
REMOTEWRITE	single data byte
REMOTECTRL	(none)

REMOTESTAT	STATREC pointer CONTROL word
USERREAD	block number byte count data area address device number CONTROL word
USERWRITE	(same as USERREAD)
USERCTRL	device number
USERSTAT	device number STATREC pointer CONTROL word
SYSREAD	block number byte count data area address device number CONTROL word
SYSWRITE	(same as SYSREAD)
SYSCTRL	device number EVENT vector
SYSSTAT	STATREC pointer CONTROL word
QUIET	(none)
ENABLE	(none)
SEREAD	device number single data byte
SERWRITE	device number single data byte
SERCTRL	device number
SERSTAT	device number STATREC pointer CONTROL word

PROCESSOR-SPECIFIC BIOS CALLS

8086/8088

Entry Points: All BIOS entry points are given as positive offsets from the BIOS vector table. The location of this vector table is given by the label BIOSVC, which is defined with a .DEF in the BIOS. Each entry in the vector table should be a pointer to the routine that implements that BIOS function. The pointer is relative to be beginning of the PME.

Parameters: When parameters aren't being passed in a specified register, they are pushed onto the stack. Offsets from the address pointed to by SP (indicated as (SP)) are given. (Remember that the stack grows down and that SP normally points at the last word pushed on the stack.)

Completion Code: Return in register AH.

Calling Sequence: The RSP will use the CALL BIOSVC(BX) (intrasegment, indirect) to call the routine within the BIOS. The BIOS routines may make free use of registers AX,BX,CX,DX,BP,SI,DI, with the exception of QUIET, ENABLE, and SYSTEMSTAT which may only use AX,BX,CX,DI. Registers CS,DS,SS,ES must be returned unchanged.

Entry Point	Offset (hex)	Parameters
CONSOLEREAD	00	return data byte in AL
CONSOLEWRITE	02	write data byte in AL
CONSOLECTRL	04	BREAK vector at (SP)+2, (SP)+3 SYSCOM pointer at (SP)+4, (SP)+5
CONSOLESTAT	06	STATREC pointer at (SP)+2, (SP)+3 CONTROL word at (SP)+4, (SP)+5
PRINTERREAD	08	return data byte in AL
PRINTERWRITE	0A	write data byte in AL
PRINTERCTRL	0C	(none)
PRINTERSTAT	0E	STATREC pointer at (SP)+2, (SP)+3 CONTROL word at (SP)+4, (SP)+5
DISKREAD	10	block number at (SP)+2, (SP)+3 byte count at (SP)+4, (SP)+5 data area address at (SP)+6, (SP)+7 drive number at (SP)+8, (SP)+9 CONTROL word at (SP)+10, (SP)+11 data area segment in ES
DISKWRITE	12	(same as DISKREAD)
DISKCTRL	14	drive number in CL
DISKSTAT	16	drive number in CL STATREC pointer at (SP)+2, (SP)+3 CONTROL word at (SP)+4, (SP)+5
REMOTEREAD	18	return data byte in AL
REMOTEWRITE	1A	write data byte in AL
REMOTECTRL	1C	(none)
REMOTESTAT	1E	STATREC pointer at (SP)+2, (SP)+3 CONTROL word at (SP)+4, (SP)+5
USERREAD	20	block number at (SP)+2, (SP)+3 byte count at (SP)+4, (SP)+5 data area address at (SP)+6, (SP)+7 device number at (SP)+8, (SP)+9 CONTROL word at (SP)+10, (SP)+11 data area segment in ES
USERWRITE	22	(same as USERREAD)
USERCTRL	24	device number in CL
USERSTAT	26	device number in CL STATREC pointer in (SP)+2, (SP)+3 CONTROL word in (SP)+4, (SP)+5
SYSREAD	28	block number at (SP)+2, (SP)+3 byte count at (SP)+4, (SP)+5 data area address at (SP)+6, (SP)+7 drive number at (SP)+8, (SP)+9 CONTROL word at (SP)+10, (SP)+11 data area segment in ES
SYSWRITE	2A	(same as SYSREAD)
SYSCTRL	2C	EVENT vector at (SP)+2, (SP)+3 device number in CL
SYSSTAT	2E	device number in CL STATREC pointer in (SP)+2, (SP)+3 CONTROL word in (SP)+4, (SP)+5
QUIET	30	(none)
ENABLE	32	(none)
SERIALREAD	34	return data byte in AL device number in CL

SERIALWRITE	36	write data byte in AL device number in CL
SERIALCTRL	38	device number in CL
SERIALSTAT	3A	device number in CL STATREC pointer in (SP)+2, (SP)+3 CONTROL word in (SP)+4, (SP)+5

8080/Z80

Entry Points: All BIOS entry points are given as positive offsets from the beginning of the BIOS code space. These locations should contain a JMP instruction to the appropriate address in the BIOS.

Parameters: When parameters aren't being passed in a specified register, they are pushed onto the stack. Offsets from top-of-stack are given (the stack grows down).

Completion Code: Return in register A.

Calling Sequence: The RSP will use the CALL instruction to call the BIOS. Thus the return address is at (SP),(SP)+1. All registers are available for use by the BIOS. The BIOS should clean off the stack before returning to the RSP.

Entry Point	Offset(hex)	Parameters
CONSOLEREAD	00	return data byte in Reg C
CONSOLEWRITE	03	write data byte in Reg C
CONSOLECTRL	06	BREAK vector at (SP)+2,(SP)+3 SYSCOM pointer at (SP)+4,(SP)+5
CONSOLESTAT	09	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
PRINTERREAD	0C	return data byte in Reg C
PRINTERWRITE	0F	write data byte in Reg C
PRINTERCTRL	12	(none)
PRINTERSTAT	15	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
DISKREAD	18	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 drive number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
DISKWRITE	1B	(same as DISKREAD)
DISKCTRL	1E	drive number in Reg C

DISKSTAT	21	drive number in Reg C STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
REMOTEREAD	24	return data byte in Reg C
REMOTEWRITE	27	write data byte in Reg C
REMOTECTRL	2A	(none)
REMOTESTAT	2D	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
USERREAD	30	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 device number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
USERWRITE	33	(same as USERREAD)
USERCTRL	36	device number in Reg C
USERSTAT	39	device number in Reg C STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
SYSREAD	3C	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 device number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
SYSWRITE	3F	(same as SYSREAD)
SYSCTRL	42	EVENT vector at (SP)+2, (SP)+3
SYSSTAT	45	STATREC pointer at (SP)+2, (SP)+3 CONTROL word at (SP)+4, (SP)+5
QUIET	48	(none)
ENABLE	4B	(none)
SERIALREAD	4E	return data byte in C device number at (SP)+2, (SP)+3
SERIALWRITE	51	write data byte in C device number at (SP)+2, (SP)+3
SERIALCTRL	54	device number at (SP)+2, (SP)+3
SERIALSTAT	57	device number at (SP)+2, (SP)+3 STATREC pointer in (SP)+4, (SP)+5 CONTROL word in (SP)+6, (SP)+7

6502

Entry Points: All BIOS entry points are given as positive offsets from the beginning of the BIOS code space. These locations should contain a JMP instruction to the appropriate address in BIOS.

Parameters: When parameters aren't being passed in a specified register, they are pushed onto the stack. Offsets from the address pointed to by S (indicated by (S)) are given (the stack grows down; and that S normally points to the first available address below valid data).

Completion Code: Return in register X.

Calling Sequence: The RSP will use the JSR instruction to call the BIOS. Thus the return address is at (S)+1, (S)+2. All registers are available for use. The stack should be cleaned off by the BIOS before returning to the RSP.

Entry Point	Offset(hex)	Parameters
CONSOLEREAD	00	return data byte in Reg A
CONSOLEWRITE	03	write data byte in Reg A
CONSOLECTRL	06	BREAK vector at (S)+3,(S)+4 SYSCOM pointer at (S)+5,(S)+6
CONSOLESTAT	09	STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
PRINTERREAD	0C	return data byte in Reg A
PRINTERWRITE	0F	write data byte in Reg A
PRINTERCTRL	12	(none)
PRINTERSTAT	15	STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
DISKREAD	18	block number at (S)+3,(S)+4 byte count at (S)+5,(S)+6 data area address at (S)+7,(S)+8 drive number at (S)+9,(S)+A CONTROL word at (S)+B,(S)+C

DISKWRITE	1B	(same as DISKREAD)
DISKCTRL	1E	drive number in Reg A
DISKSTAT	21	drive number in Reg A STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
REMOTEREAD	24	return data byte in Reg A
REMOTEWRITE	27	write data byte in Reg A
REMOTECTRL	2A	(none)
REMOTESTAT	2D	STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
USERREAD	30	block number at (S)+3,(S)+4 byte count at (S)+5,(S)+6 data area address at (S)+7,(S)+8 device number at (S)+9,(S)+A CONTROL word at (S)+B,(S)+C
USERWRITE	33	(same as USERREAD)
USERCTRL	36	device number in Reg A
USERSTAT	39	device number in Reg A STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
SYSREAD	3C	block number at (S)+3,(S)+4 byte count at (S)+5,(S)+6 data area address at (S)+7,(S)+8 device number at (S)+9,(S)+A CONTROL word at (S)+B,(S)+C
SYSWRITE	3F	(same as SYSREAD)
SYSCTRL	42	device number in A EVENT vector at (S)+3, (S)+4
SYSSTAT	45	device number in A STATREC pointer in (S)+3, (S)+4 CONTROL word in (S)+5, (S)+6
QUIET	48	(none)
ENABLE	4B	(none)
SERIALREAD	4E	return data byte in A device number in Y
SERIALWRITE	51	write data byte in A device number in Y
SERIALCTRL	54	device number in A
SERIALSTAT	57	device number in A STATREC pointer in (S)+3, (S)+4 CONTROL word in (S)+5, (S)+6

6809

Entry Points: All BIOS entry points are given as positive offsets from the beginning of the BIOS code space. These locations should contain a vector to the appropriate address in the BIOS.

Parameters: When parameters aren't being passed in a specified register, they are pushed onto the stack. Offsets from the address pointed to by SP (indicated by (SP)) are given (the stack grows down; and that SP normally points at the last item pushed on the stack).

Completion Code: Return in register B.

Calling Sequence: The RSP will use the JSR instruction to call the BIOS. Thus, the return address will be at (SP)+0, (SP)+1. The U and Y registers contain PME information which must be preserved/restored by the BIOS prior to returning to the RSP. All other registers are available for use. The stack should be cleaned off by the BIOS before returning to the RSP.

Entry Point	Offset (hex)	Parameters
CONSOLEREAD	00	return data byte in Reg A
CONSOLEWRITE	03	write data byte in Reg A
CONSOLECTRL	06	BREAK vector at (SP)+2,(SP)+3 SYSCOM pointer at (SP)+4,(SP)+5
CONSOLESTAT	09	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5

PRINTERREAD	0C	return data byte in Reg A
PRINTERWRITE	0F	write data byte in Reg A
PRINTERCTRL	12	(none)
PRINTERSTAT	15	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
DISKREAD	18	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 drive number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
DISKWRITE	1B	(same as DISKREAD)
DISKCTRL	1E	drive number in Reg A
DISKSTAT	21	drive number in Reg A STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
REMOTEREAD	24	return data byte in Reg A
REMOTEWRITE	27	write data byte in Reg A
REMOTECTRL	2A	(none)
REMOTESTAT	2D	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
USERREAD	30	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 device number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
USERWRITE	33	(same as USERREAD)
USERCTRL	36	device number in Reg A
USERSTAT	39	device number in Reg A STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
SYSREAD	3C	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 device number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
SYSWRITE	3F	(same as SYSREAD)
SYSCTRL	42	device number in A EVENT vector at (SP)+2, (SP)+3
SYSSTAT	45	device number in A STATREC pointer at (SP)+2, (SP)+3 CONTROL word at (SP)+4, (SP)+5
QUIET	48	(none)
ENABLE	4B	(none)
SERIALREAD	4E	return data byte in A device number at (SP)+2, (SP)+3
SERIALWRITE	51	write data byte in A device number at (SP)+2, (SP)+3
SERIALCTRL	54	device number at (SP)+2, (SP)+3
SERIALSTAT	57	device number at A STATREC pointer in (SP)+2, (SP)+3 CONTROL word in (SP)+4, (SP)+5

68000

Entry Points: All BIOS entry points are given as positive offsets from the BIOS jump table. The location of this jump table is given by the label BIOSVC which is defined with a .DEF in the BIOS. Each entry in the jump table should be a long BRAnch to the routine that implements that BIOS function.

Parameters: In general, parameters are passed to the BIOS using the following register scheme:

```
D0.B - character          A0 - free
D1.B - result code        A1 - free
D2.W - control            A2 - buffer address
D3.W - block number
D4.W - bytes
D5.B - unit number
```

Completion Code: Returned in register D1.B.

Calling Sequence: The RSP will use the JSR instruction to call into the jump table within the BIOS. The BIOS routines may make free use of registers D0,D1,A0,A1. No other registers (including the parameter registers) may be destroyed.

Entry Point	Offset (hex)	Parameters
CONSOLEREAD	00	return data byte in D0.B
CONSOLEWRITE	04	write data byte in D0.B
CONSOLECTRL	08	SYSCOM pointer in A0 BREAK vector in A1
CONSOLESTAT	0C	STATREC pointer in A2 CONTROL word in D2.W
PRINTERREAD	10	return data byte in D0.B
PRINTERWRITE	14	write data byte in D0.B
PRINTERCTRL	18	(none)
PRINTERSTAT	1C	STATREC pointer in A CONTROL word in D2.W
DISKREAD	20	block number in D3.W byte count in D4.W data area address in A2 drive number in D5.B CONTROL word in D2.W
DISKWRITE	24	(same as DISKREAD)
DISKCTRL	28	drive number in D5.B
DISKSTAT	2C	drive number in D5.B STATREC pointer in A2 CONTROL word in D2.W
REMOTEREAD	30	return data byte in D0.B
REMOTEWRITE	34	write data byte in D0.B
REMOTECTRL	38	(none)
REMOTESTAT	3C	STATREC pointer in A2 CONTROL word in D2.W
USERREAD	40	block number in D3.W byte count in D4.W data area address in A2 device number in D5.B CONTROL word in D2.W
USERWRITE	44	(same as USERREAD)
USERCTRL	48	device number in D5.B
USERSTAT	4C	device number in D5.B STATREC pointer in A2 CONTROL word in D2.W
SYSREAD	50	block number in D3.W byte count in D4.W data area address in A2 device number in D5.B CONTROL word in D2.W
SYSWRITE	54	(same as SYSREAD)
SYSCTRL	58	device number in D5.B EVENT vector in A0
SYSSTAT	5C	device number in D5.B STATREC pointer in A2 CONTROL word in D2.W
QUIET	60	(none)
ENABLE	64	(none)
SERIALREAD	68	return data byte in D0.B device number in D5.B
SERIALWRITE	6C	write data byte in D0.B device number in D5.B
SERIALCTRL	70	device number in D5.B

SERIALSTAT	74	device number in D5.B STATREC pointer in A2 CONTROL word in D2.W

Operating System

CHAPTER 5

THE OPERATING SYSTEM

OVERVIEW OF THE OS

The operating system is a collection of Pascal UNITs. The organization of UNITs in the operating system was determined by three considerations: functional grouping, space and language restrictions, and necessary code-sharing with other portions of the system. Some UNITs (such as SCREENOPS) are intended to be accessible to your programs as well. The name of a UNIT in the operating system generally reflects its function. This is a full list of operating system UNITs:

Unit Name	Function
HEAPOPS EXTRAHEAP PERMHEAP	Heap operators
SCREENOPS	Screen control
FILEOPS	File and directory operations
PASCALIO EXTRAIO SOFTOPS	File-level I/O
SMALLCOMMAND COMMANDIO	I/O redirection and chaining
STRINGOPS	String intrinsics
OSUTIL	Conversion utilities
CONCURRENCY	Concurrency
REALOPS	Floating point functions and real number I/O
LONGOPS	Long integer operations
GOTOXY	Screen cursor control (may be user-supplied)
KERNEL	Nonswappable central facilities of op. system (always resident in main memory)
GETCMD USERPROG INITIALIZE PRINTERROR	Subsidiary segments of KERNEL (swappable)

KERNEL contains the resident code necessary to maintain the code pool, handle faults, and read segments. The Kernel also contains four subsidiary segments, which are swappable:

GETCMD processes your input at the main command level, and builds your program's run-time environment;

USERPROG is the reserved segment slot for your program (at bootstrap time it contains the Pascal-level code which builds the initial run-time environment for the operating system);

INITIALIZE is called when the system is booted or reinitialized. It reads SYSTEM.MISCINFO, locates the system code files, and sets up the table of devices;

PRINTERROR prints run-time error messages.

The operating system UNITs are compiled separately. They are bound together in a single code file, SYSTEM.PASCAL, by using the utility LIBRARY.

Because of certain bootstrap restrictions, KERNEL must always reside in segment-slot 0 and USERPROG must always reside in slot 15. There are no other restrictions on the location of units within SYSTEM.PASCAL.

P-MACHINE SUPPORT

The Heap: An Overview

The heap is an area in low memory used for the allocation of dynamically stored variables. The upper bound of the heap depends upon the size of the stack (and the code pool if it is internal). The area between the heap and its upper bound is provisionally available to the heap: stack faults and segment faults may change the size of this area. Heap faults are used by the heap operators to request that more space be allocated to the heap.

The heap is manipulated by a number of intrinsic routines that allocate or deallocate heap space in a particular way. These routines are described below.

MARK and RELEASE

MARK saves the location of the current top of the heap. RELEASE cuts the heap back to the location of the corresponding mark. Variables which were allocated between the time of the MARK and the time of the RELEASE are removed from the heap, except for variables allocated by PERMNEW. MARK and RELEASE may be nested; the integrity of the heap requires that they be correctly paired.

NEW and VARNEW

NEW and VARNEW cause variables to be allocated on the heap above the topmost mark. NEW(P), where variable P is a pointer to type T, causes the number of words in type T to be allocated. P is assigned the address of the first location allocated to P on the heap. If T is a record with variants, space for the largest variant is allocated. In Pascal, a call to NEW may designate a particular variant, so that space for this particular variant is allocated (which may be less than the largest variant in that record).

VARNEW(P,NWords), where P is a pointer to type T, causes NWords to be allocated on the heap. T would most commonly be an array. NWords (indirectly) determines how many elements of the array are actually available in this instance. P returns the address of the first location allocated on the heap.

VARNEW is a function, and returns the number of words that actually were allocated. This should equal NWords; if it is 0, then there was less than NWords of available space, and if it is some other number, something went wrong.

DISPOSE and VARDISPOSE

DISPOSE and VARDISPOSE de-allocate space reserved by NEW and VARNEW, respectively. DISPOSE(P) frees the number of words pointed to by P. VARDISPOSE(P,NWords) frees NWords words. In both cases, P is assigned the value NIL.

To avoid destroying important information that is on the heap, extreme caution should be used with these intrinsics, which do little error-checking of their own. Heap space allocated by a VARNEW should be freed only by a VARDISPOSE with the same NWords parameter, and MARK/RELEASE pairs should always match. Furthermore, if the NEW is called for a specific variant, the same variant should be used to DISPOSE that area.

If these intrinsics are misused, the system is likely to crash. This is the least mysterious of the symptoms that may occur.

PERMNEW and PERMDISPOSE

A variable can be allocated on the heap by PERMNEW(P), where P is a pointer to the variable's type. A variable allocated by PERMNEW can only be deallocated by PERMDISPOSE(P). Even a RELEASE can't remove it. These routines are meant for the system's use, not yours.

The operating system uses these routines to allow variables to remain defined across MARK/RELEASE pairs. Program CHAIN commands are saved on the heap with PERMNEW, so that even after the chaining program terminates, and its heap space is released, these commands are still available to determine the further actions of the system.

Heap Implementation

Unit Organization

Code for the heap operators is contained in three units: HEAPOPS, EXTRAHEAP, and PERMHEAP. HEAPOPS contains MARK, RELEASE, and NEW. EXTRAHEAP contains DISPOSE, VARNEW, VARAVAIL, MEMLOCK, and MEMSWAP. PERMHEAP contains PERMNEW, PERMDISPOSE, and PERMRELEASE. (VARAVAIL, MEMLOCK, and MEMSWAP are for segment management and are discussed elsewhere.)

Heap Globals

The operating system uses several variables to manage the heap. The heap is maintained by a linked list of MARKs. The topmost MARK is indicated by HeapInfo.TopMark. A MARK (also called an HMR, for heap mark record) has the following structure:

```
TYPE
 MemLink = RECORD
             Avail_list: MemPtr;
             NWords: integer;
             CASE Boolean OF
               true: (Last_Avail,
                      Prev_Mark: MemPtr);
           END;
```

In a MARK, NWords is always 0, and the variant is always TRUE. NWords is 0 because the MARK merely marks a location on the heap, and doesn't reserve any space.

Each MARK points to an Avail_List, which is a list of records of type MemLink. These records are FALSE variants of MemLink, and NWords contains the number of words of available space (including the two words of the record itself). The Avail_List chain is ended by an Avail_List of NIL.

The first MARK on the heap contains a Prev_Mark of NIL. All successive MARKs point back to their predecessor, so that the MARK chain can be traversed.

For each MARK, the first Avail_List record is the lowest unallocated space above the MARK. Last_Avail points to the last of the available space. This is typically bounded by allocated heap space or by another MARK; if the MARK is TopMark, Last_Avail is bounded by the code pool if the pool is internal or by the stack if the pool is external.

The heap maintenance variables have the following structure:

```
VAR
 HeapInfo: RECORD
             Lock: semaphore;
             TopMark,
             HeapTop: MemPtr;
           END;
 PermList: MemPtr;
```

The Lock semaphore guarantees that the heap is modified by only one process at a time. TopMark points to the highest MARK. HeapTop points to the highest allocated space on the heap. The Faulthandler uses HeapTop to determine how close the code pool or the stack can be to the heap. A base value is computed which is either the base of the code pool (if internal) or SP_Low (if the pool is external). PermList points to a linked list of PERMNEW'ed variables. The list is identical in structure to an Avail_List, but each NWords indicates the number of words allocated by a PERMNEW. If PermList is NIL, then no variables have been PERMNEW'ed.

Tactics

In general, a request for heap space through a MARK, NEW, VARNEW, or PERMNEW causes HeapTop to be set to the new top of the heap. The fault handler always places the code pool (located at PoolBase) above HeapTop if the pool is internal. If the pool is external, the stack and heap are allowed to grow toward each other. If they meet, a stack overflow condition exists. Thus, HeapTop reserves space for the heap as soon as a heap operator requests it. This is necessary because of possible interactions between stack fault handling and heap space allocation.

The operating system uses the global variable SysCom^.GDirP (global directory pointer) to allocate a disk directory on the heap. The operating system's use of this heap space is meant to be invisible to you. Therefore, before any heap operation (except DISPOSE), SYSCOM^.GDirP is DISPOSEed to make the space occupied by the directory available again.

Run-Time Environment

Since both you and the operating system use the heap, the operating system MARK's the heap immediately before the execution of your program by the call:

```
MARK (EMPTYHEAP);
```

After your program terminates, the operating system calls:

```
RELEASE (EMPTYHEAP);
```

Thus, all your space is freed after the program terminates, unless space has been allocated by one or more calls to PERMNEW.

MARK (EMPTYHEAP) occurs after the run-time environment for your program has been built. The program's run-time environment structures such as SIBs, E_Rec's, and E_Vect's, are for the use of the operating system, and are allocated space before EMPTYHEAP. Data that is global to your program and any units it USES also appears before EMPTYHEAP. Heap space that follows EMPTYHEAP is intended only for the local use of your program.

The heap is shared by all tasks in the system.

THE CODE POOL

The code pool resides in main memory between the stack and the heap. It contains executable code segments that may possibly be discarded, or swapped in from disk again. Thus, the contents, size, and position of the code pool may change during a program's execution. The flexibility of code pool handling can provide a running program with more free memory space than in previous versions.

A segment in the code pool must be either p-code or relocatable native code. Nonrelocatable native code segments reside on the heap; they are placed there at associate time.

The code pool is a contiguous block of code segments—whenever a segment is discarded, the surrounding segments are moved together. Segments being swapped in are given space at either end of the code pool.

Segments in the code pool are organized into a doubly-linked list by pointers in each segment's SIB (described in the previous chapter).

The routines that manage the code pool are in the operating system's KERNEL unit. They make use of the following global values:

SegPool: ^PoolDes; — This field is within the SIB. It points to a description of the code pool, which is declared as follows:

```
Pooldes = Record
            PoolBase   : fulladdress;
            PoolSize   : integer;
            MinOffset  : memptr;
            MaxOffset  : memptr;
            Resolution : interger; (in bytes)
            PoolHead   : SIB_P;
            Perm_SIB   : SIB_P;
            Extended   : boolean;
          End;
```

PoolBase: Full_Address; — Points to the memory location at the base of the code pool. (A Full_Address is a 32-bit address.)

PoolSize: Integer; — The size of the code pool in words. Set by the SETUP utility.

MinOffset: Integer; — Lower boundary of code pool.

MaxOffset: Integer; — Upper boundary of code pool.

Resolution: Integer;	A segment must be placed in memory starting at a location which has an address that is a multiple of this number. This is set by the SEGMENT ALIGNMENT field in SYSTEM.MISCINFO (determined by the SETUP utility).
PoolHead: SIB_P;	Points to the SIB of the segment at the base of the code pool (next to the heap).
PermSIB: SIB_P;	Points to the SIB of the segment that is always resident in the code pool (currently, GOTOXY).
Extended: Boolean;	True if extended memory is used; false, otherwise. Set from the HAS EXTENDED MEMORY field in SYSTEM.MISCINFO (determined by the SETUP utility).
SP_Low: Mem_Ptr;	The lowest possible bound of the stack; this points to the address which is one word above the top of the code pool (if it is internal). SP_Low is in the TIB.

HeapTop: Mem_Ptr;	Points to the top of the heap. HeapTop is part of the HeapInfo record.

If the code pool is internal when space is requested either for the heap or the stack, the code pool management routines first attempt to reposition the code pool without swapping out any segments.

The actual bounds of the code pool are in MinOffset, which points to the low end of the code pool, and MaxOffset, which points to one word above the top of the code pool. The code pool operators may move the pool all the way to HeapTop on the heap side, or up to SP minus a 40-word margin on the stack side (if the pool is internal). MaxOffset is the same as SP_Low for an internal pool.

An internal code pool may be modified under any of the following circumstances:

1. A heap fault is detected, and the code pool is moved up in memory toward the stack to free the needed number of words for the heap.

2. A stack fault is detected, and the code pool is moved down in memory toward the heap to free the needed number of words for the stack.

3. A heap fault or stack fault is detected, and the code pool can't be moved to allocate the space; one or more segments are swapped out, the remaining segments are moved together, and the code pool is positioned to allow for the needed heap or stack space.

4. A heap or stack fault is detected, and even after swappping out all of the swappable segments, not enough space is available; a stack overflow error is reported, and the system is reinitialized.

An internal or external code pool may be modified when a segment fault is detected. The code pool management routines first try to read the segment in at either end of the code pool without moving it. If this is impossible, and the pool is internal, they attempt to create more room by moving the code pool toward either the stack or the heap, and then read the segment. If this too is impossible, or the pool is external, segments are swapped out to make room, and the new segment is then read in. If this last effort also fails, a stack overflow error is reported if the pool is internal or a pool overflow error is reported if the pool is external. The system is reinitialized.

The code pool management routines are only called by the Faulthandler. Since the Faulthandler is a subsidiary task, its own stack is statically allocated. Thus, the Faulthandler can manipulate the code pool freely, without fear of causing a stack fault.

Fault Handling

When memory space is required by the stack or heap, or entry into a nonresident segment is attempted, a fault is issued. The Faulthandler process is activated, and uses the code pool management routines to rearrange main memory (as described in the previous section).

The Faulthandler is a process that is STARTed at bootstrap time. Most of the time it is idle, it WAITs for a semaphore. When the semaphore is SIGNALed, the Faulthandler is activated and performs its memory management functions.

Faults can be SIGNALed by the PME (stack and segment faults), or by the EXECERROR procedure in the operating system (heap faults and one segment fault).

The semaphore record used by the Faulthandler resides in SYSCOM. It is declared as follows:

```
Fault_Message = RECORD
                    Fault_TIB: TIB_Ptr;
                    Fault_E_Rec: E_Rec_Ptr;
                    Fault_Words: integer;
                    Fault_Type: Seg_Fault .. Pool_Fault;
                  END;

Fault_Sem: RECORD
              Real_Sem, Message_Sem: semaphore;
              Message: Fault_Message;
            END;
```

The PME detects only stack and segment faults. When the PME detects a fault, it places the appropriate information in Fault_Sem.Message and SIGNALs Fault_Sem.Message_Sem. The SIGNAL causes a task switch to the Faulthandler, and the fault is processed. After it has dealt with the code pool, Faulthandler WAITs—this causes a task switch back to the previously running process. The instruction that caused the fault is reexecuted.

The operating system issues heap faults, and in one instance, a segment fault. Heap faults are detected by the heap operators when requests are made for heap space by MARK, NEW, VARNEW, and PERMNEW. The one segment fault is issued by MEMLOCK if a segment to be locked in the code pool isn't already resident. To issue a fault, the operating system calls the execution error procedure (EXECERROR), and passes it the needed information. EXECERROR then performs a SIGNAL on Message_Sem.

The Faulthandler first ensures that the currently running segment isn't swapped out, and then uses the code pool management routines to adjust the main memory layout.

If a stack fault is caused by a call to a routine in a different segment, Faulthandler must lock both calling and called segments into memory.

Concurrency

Operating system routines support concurrency only by the activation and deactivation of processes; actual task switching is accomplished by the p-machine operations SIGNAL and WAIT.

Concurrency support is intended for low-level tasks. Most system-level facilities, particularly I/O, are synchronous. For instance, a READ or UNITREAD from the console doesn't return to the caller until a character is available. No task switch can occur during the waiting period.

The operating system global variable Task_Info is used to keep track of some of the data for subsidiary processes. Its structure is as follows:

```
Task_Info: RECORD
             Lock,
             Task_Done: semaphore;
             N_Tasks: integer;
           END {of Task_Info};
```

Task_Info.Lock is used to ensure mutual exclusion while changing the values of other Task_Info fields. Task_Done is used to WAIT for the termination of any subsidiary processes. N_Tasks is the number of subsidiary tasks that have been STARTed.

The unit CONCURRENCY has three routines: START, STOP, and BLK_EXIT. For each process initiation, the compiler emits initialization code that signals the semaphore passed to START. The compiler also emits a call to STOP in the exit code of each process; a call to BLK_EXIT is part of the exit code of a main process.

START builds the data structures for a new task and sets it in execution. The task's TIB, activation record, and stack space are allocated on the heap, and the operating system forces a task switch by issuing a WAIT. Presumably, the new process starts executing, and switches back to START by doing a SIGNAL after its parameters have been copied. Actually, when START performs the WAIT, it is the process with the highest priority that begins executing.

STOP records the termination of a process. It decrements Task_Info.N_Tasks, SIGNALs Task_Info.Task_Done, and then initializes and waits for a dummy semaphore in order to force a permanent task switch from the terminating process.

BLK_EXIT is called by a main task, and waits for the termination of all subsidiary tasks. It waits on Task_Done, and terminates the main task when N_Tasks equals zero.

I/O SUPPORT

FIBs

File I/O is controlled with a structure called a FIB (File Information Block). When you declare a file, the compiler emits code to initialize a FIB for that file. A FIB is declared as follows:

```
FIB = RECORD
       FWindow: Window_P;
       FEOF, FEOLN: Boolean;
       FState: (FJandW, FNeedChar, FGotChar);
       FRecSize: integer;
       FLock: semaphore;
       CASE FIsOpen: Boolean OF
         true: (FIsBlkd: Boolean;
                FUNIT:UNITNUM;
                FVID:VID:
                FReptCnt,
                FNxtBlk,
                FMaxBlk: integer;
                FModified: Boolean;
                FHeader: DirEntry;
                CASE FSoftBuf: Boolean OF
                  true: (FNxtByte, FMaxByte: integer;
                         FBufChngd: Boolean;
                         FBuffer: PACKED ARRAY [0..FBlkSize]
                                  OF CHAR))
       END {of FIB}
```

FWindow points to the current character in the file's buffer. FEOF and FEOLN are the EOF and EOLN flags. FState indicates that the file is either a standard (Jensen and Wirth) file, an INTERACTIVE file awaiting a character, or an INTERACTIVE file with a character. FRecSize is 0 for unentered files, 1 for INTERACTIVE files and text files; if it is larger than zero, it indicates the size (in bytes) of a record. FLock is used to ensure that only one process at a time may modify the file. FIsOpen is TRUE only when the file is open.

If FIsOpen is TRUE, then several other fields become relevant. FIsBlkd is TRUE if the file resides on a storage device. FDev is the number of that device, and FVolID is the name of the volume. FReptCnt contains a count of the number of times the window value is valid before another GET is needed. FNxtBlk is the next (relative) block to access. FMaxBlk is the maximum (relative) block that can be accessed. FModified becomes TRUE if the file is modified; a new date is then set in the directory. FHeader is a copy of the file's directory entry. FSoftBuf is TRUE if soft-buffered I/O is used. This is the case for all files on storage device, except unentered files.

If FSoftBuf is TRUE, then the last set of FIB fields are used. FNxtByte and FMaxByte are used for buffer handling, FBufChngd indicates that the buffer contents have been modified, and FBuffer is the buffer itself.

Directories

Figure 5-1 illustrates the structure of a directory (as on a disk or other storage device).

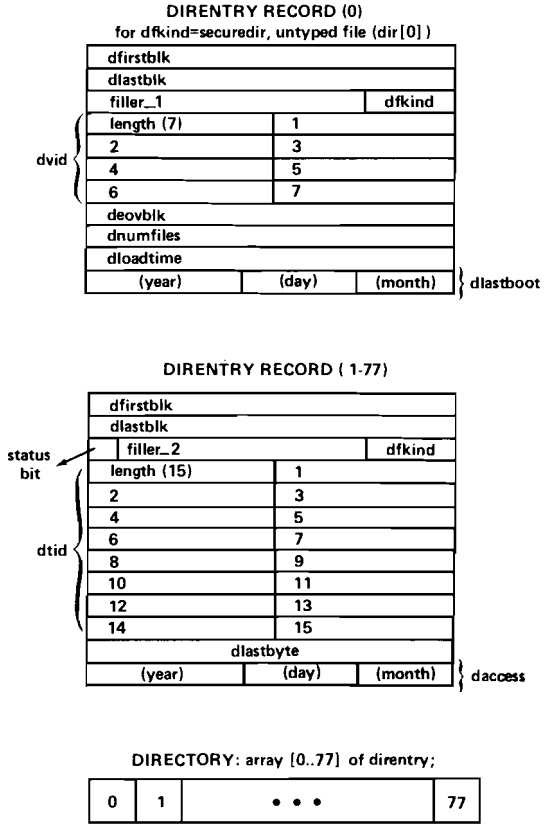


Figure 5-1. Directory Format

VARIETIES OF I/O

Record I/O

Record I/O applies to entered Pascal files, using the intrinsics GET and PUT.

Screen I/O

Screen I/O may be handled by the unit SCREENOPS, whose routines are described in the following section.

Input from the screen is accomplished by the procedure CHAR_DEV_GET, which uses SC_CHECK_CHAR (in SCREENOPS) and SYSCOM^.MISCINFO to determine whether any special handling needs to be done.

Output to the screen is accomplished by a simple UNITWRITE.

Block I/O

Block I/O applies to unentered files. The routines BLOCKREAD and BLOCKWRITE are used. These are part of the system routine FBLOCKIO in the EXTRAIO unit.

When a file is accessed as an unentered file, all other file formatting is disabled.

Text I/O

A text file is a file of ASCII characters. It has a 2-block header that contains formatting information used by the Screen Oriented Editor. When a text file is used by a system program other than the editor, the operating system ignores this header. When a new text file is created, the operating system writes a 2-block header filled with NULs. When SofTech's internal part number is added to a text file, it is stored in the last two words of the header (end of block 1).

Text files always have an even number of blocks. Thus, the smallest possible text file is four blocks long. Each pair of blocks after the header is considered as a "page." Each page contains lines of text terminated by <return>. The last line of text in a page must not be continued on the next page in the text file. Extra space after the last line in each page must be filled with NULs (decimal 0).

Each line in a text file may optionally start with a DLE (decimal 16), which is interpreted as a blank compression code. The byte following a blank compression code is ASCII code 32+n, where n is the number of leading blanks. This blank compression code is generated by the editor (chiefly for the purpose of saving space in indented program source).

Your programs typically handle text files with READ, READLN, WRITE, and WRITELN. GET and PUT may be used, and follow the Jensen and Wirth standard for files of type TEXT.

Program Execution

CHAPTER 6

PROGRAM EXECUTION

BUILDING A RUN-TIME ENVIRONMENT

The run-time environment for your program is created by the operating system's GETCMD unit. GETCMD starts the execution of system programs such as the compiler, linker, filer, and so on, and your programs named in the X(ecute command. In all such cases, GETCMD calls the procedure ASSOCIATE, which finds the appropriate code file, and then calls BUILDENV. BUILDENV constructs a program's run-time environment, as outlined in Chapter 3, "The P-Machine."

BUILDENV recursively traverses the segments used by a program. For each segment, it initializes an E_Vect, E_Rec, and SIB. As each E_Rec is created, it is linked to a chain of segments that are already active. In this way, the operating system can keep track of all active segments. Before BUILDENV initializes segment information, it checks to see if that segment is already active, and if it is, it does nothing but initialize the proper pointers. Otherwise, the E_Vect, E_Rec, and SIB must be created from information present in the code file.

SEGREFs are segment reference assignments emitted by the compiler. Segment numbers are local to a code segment. The main program is segment 2, and subsidiary segments, if any, are numbered starting from 3. Segment 1 is always the operating system's KERNEL unit. SEGREFs are emitted for any principal segments used by the compilation (such as a used unit). At associate time, BUILDENV uses the SEGREF list to find the segments that the program uses.

All run-time errors detected by the system cause the current program to halt. The system displays an error message, and when you press <space>, the system is reinitialized. The program's run-time environment is lost.

When a program terminates, control returns to GETCMD, which waits for further instructions. When a program terminates normally, its environment is not lost, and the program can be restarted with the U(ser restart command. The system may or may not need to call BUILDENV again.

QUICKSTARTING PROGRAMS

The QUICKSTART utility (described in the Operating System Reference Manual) constructs a description of the execution environment for a program and generates a code file for the program which contains this execution environment description. The GETCMD unit detects the presence of execution environment descriptions within code files and attempts to reconstruct the required execution environment from such descriptions when the programs are invoked. In this section, an execution environment description built by the QUICKSTART utility is called a "Program Environment Descriptor" or "PED" for short.

Program Invocation Overview

When the execution of a program is requested, the system first inspects the code file to determine if an execution environment description is present within it. If such a description is present, the system attempts to reconstruct the execution environment required by the program from the description in the code file. If the code file doesn't contain an environment description, or the environment description contained within the code file is determined to be obsolete, the system attempts to build the environment for the program in the normal manner.

The program invocation process begins with an attempt at opening the code file. If the code file is sucessfully located, and the file is judged to be a code file (based on directory information), then the segment dictionary is read from block zero. Within the segment dictionary are two fields which indicate the presence, size, and location of an imbedded PED. (See description of the segment dictionary structure in the next section.)

If the PED_BLK field of the segment dictionary isn't zero, a PED exists within the code file, and an attempt is made to reconstruct the execution environment for the program using the information stored in the PED. If this reconstruction of the execution environment fails, the system automatically attempts to construct the program's execution environment using the normal execution environment construction process. When the PED_BLK word contains zero, the normal execution environment construction process is used.

A PED completely describes the execution environment required by a program. A detailed description of the structure of a PED is give in the "PED Structure" section, below.

A PED contains a list of p-System operating system units which are referenced by the program. Each of these referenced operating system units must be present within the system environment in which the program is being invoked. The execution environments for these operating system units are created by the p-System at p-System bootstrap time. The execution environments for referenced units which are resident in other library code files (such as SYSTEM.LIBRARY or a user library code file) are described completely so that the execution environment can be reconstructed without performing detailed examinations of the libraries.

Also included within the PED is a list of referenced library code files. The description of each library code file includes the name of the library code file and the name of the volume on which the library code file resided at the time the PED was constructed. Part of the environment reconstruction process involves establishing the location of each referenced library code file. A particular library code file is sought first on the volume indicated in the PED information, then on the prefix volume, and then on the root volume.

While specific volume block offsets for referenced library code files aren't recorded within a PED, the relative block locations of segments within the various referenced library code files are recorded. A mechanism is required to assure that this internal code file configuration is the same at program invocation time as it was when the PED was constructed.

In order to perform this check, the QUICKSTART utility installs into block zero of each referenced library code file a number in the form of a 16-bit checksum calculated over the entire contents of the library file. The QUICKSTART utility program only installs a new checksum into a referenced library code file when it lacks a valid checksum. The checksum value zero is reserved to indicate the absence of a valid checksum. The p-System compilers and assemblers create code files with the checksum field set to zero. The p-System LIBRARY utility program clears the checksum field when it creates a new output code file.

A copy of the checksum for each referenced library code file is also stored within the PED. During the environment reconstruction process, the checksum in each library code file is compared with the corresponding checksum in the PED. If the checksums aren't the same, the configuration which existed at QUICKSTART time has changed, and the reconstruction of the execution environment using the information stored in the PED is aborted. Thus, whenever a referenced library code file is modified, any PEDs which reference that library code file become obsolete.

The following is a rough sketch of the steps taken by the system when reconstructing the execution environment for a program from a PED:

- Using the information stored in block zero of the code file, read the contents of the PED into a temporary buffer.

- Extract the list of system unit names from the PED and locate the E_Rec for each of the referenced system units.

- Extract the referenced library code file descriptors from the PED. As each descriptor is extracted, establish the location of the specified library code file by searching the following volumes: the volume specified in the descriptor, the prefix volume, the root volume. Report an environment reconstruction failure if a particular library code file can't be located, or if the checksum stored in a located library code file doesn't match the checksum in the corresponding descriptor in the PED.

- Allocate enough memory to contain the E_Vect's, E_Rec's, and SIBs which are necessary to represent the execution environment for the program.

- Extract the set of E_Vect templates from the PED and move them into position.

- Examine each E_Vect, converting each of its entries from a "global segment number" to a pointer to the appropriate E_Rec structure.

- Extract the set of SIB templates from the PED and construct the SIBs for all of the units and subsidiary segments within the program's execution environment.

- Link the E_Rec's for the principal segments within the environment into the system's list of active E_Rec's.

Segment Dictionary Structure

Figure 6-1 shows the revised structure of block zero of a p-System code file after new fields related to QUICKSTART have been assigned. The five words which are labeled in the figure correspond to the five unused words in the original block zero structure.

From this figure, it can be seen that the first previously unused word has been allocated to hold a checksum for the code file. (The usage of this checksum was described in the previous section.)

The next new field is called PED_BLK and is the relative block number within the code file where the PED is located. If the PED_BLK word contains zero, the code file doesn't have a PED.

The third new word is called PED_BLK_COUNT and contains the size of the PED in blocks.

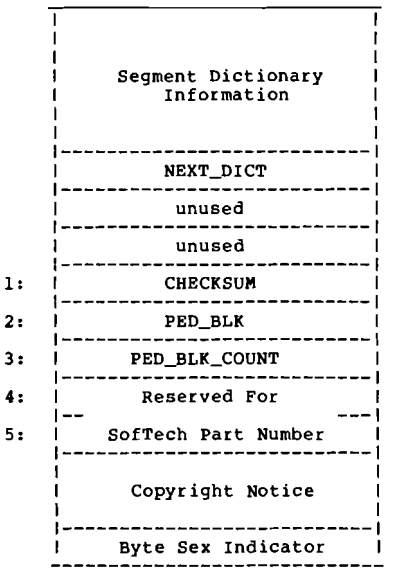


Figure 6-1. Code File Block Zero Structure

PED Structure

The general structure of a PED is illustrated in Figure 6-2. From this figure, it can be seen that a PED begins with a header record. This record contains global information about the program and about the remaining structures contained within the PED. The structure of this header record is defined by the PED_HEADER type declaration shown in Figure 6-2.

The PED_BYTE_SEX field of the header record indicates the byte sex of the PED. A value of 1 is placed into this field at the time the PED is constructed. If at program invocation time this field contains the value 256, then the PED has the opposite byte sex from the byte sex of the processor on which the quickstarted program is being invoked.

The PED_LAST_SYSTEM_SEGMENT field of the PED_HEADER record contains the number of operating system segments which are referenced by the execution environment for the program described by the PED. This field of the header record is set to the value zero if the PED describes the execution environment of the operating system itself.

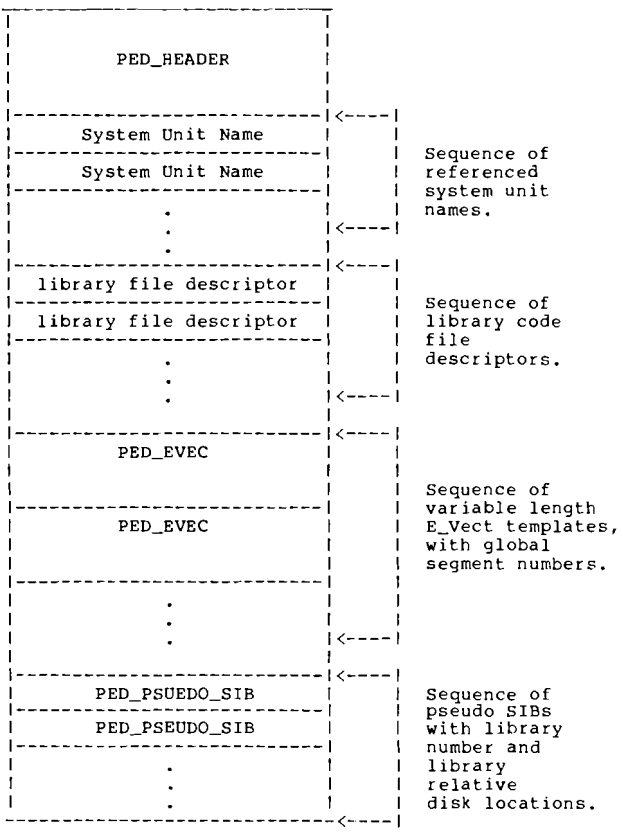


Figure 6-2. General PED Structure

```
type  {PED header record.}

   ped_header =
     record
       ped_byte_sex: integer;
                          {PED Byte sex indicator.}

       ped_format_level: integer;
                          {PED structures
                           version indicator.}

       ped_library_count: integer;
                          {Number of library
                           file descriptors.}

       ped_principal_segment_count: integer;
                          {Number of principal
                           segments described.}

       ped_subsidiary_segment_count: integer;
                          {Number of
                           subsidiary segments
                           described.}

       ped_total_evec_words: integer;
                          {Size of EVEC
                           templates.}

       ped_last_system_segment: integer;
                          {Number of system
                           segments referenced
                           by environment.}

       ped_start_unit: integer;
                          {Global segment
                           number of
                           principal segment
                           where execution
                           should begin.}

       ped_uses_realops_unit: boolean;
                          {TRUE if REALOPS
                           unit required.}

       ped_expansion_area:
           array[1..5] of 0..0;
                          {Reserved for
                           future use.}
     end;
```

Figure 6-3. PED Header Structure

Following the PED_HEADER record is a sequence of 8-character system unit names. The number of names in the sequence is given by the PED_LAST_SYSTEM_SEGMENT field of the PED_HEADER record. Each name in the sequence is the name of a system unit which is referenced by the execution environment described by the PED.

The list of system unit names is followed by a sequence of library file descriptors. The number of library file descriptors is given by the PED_LIBRARY_COUNT field of the PED_HEADER record. Each library file descriptor contains a word with the checksum for the referenced library file, followed by a string identifying the volume the library file should be located on and a string containing the title of the library file. These strings occupy the minimum number of words required for the string value. For each of these string values, if the length of the string plus the length byte is odd, a padding byte of zero will follow the string to cause the string to occupy an integral number of words in the PED.

The library file descriptors are in turn followed by a sequence of E_Vect templates. There are a total of PED_PRINCIPAL_SEGMENT_COUNT E_Vect templates which occupy PED_TOTAL_EVEC_WORDS in the PED. These E_Vect templates are exactly the same size as the E_Vect's which are needed to represent the execution environment of the program.

Each of these E_Vect templates begins with a word containing the number of entries in the E_Vect. This count word is in turn followed by the words for the E_Vect entries. The E_Vect entries are followed in the template by an extra entry (one word in length) which is required by the operating system.

Each of the E_Vect entries stored in the E_Vect templates of the PED references a particular segment of the program's environment via a "global segment number."

The global segment number zero is reserved to denote an empty E_Vect entry which should be initialized to the value NIL.

An E_Vect entry containing a negative global segment number in the range—PED_LAST_SYSTEM_SEGMENT..—1 denotes a reference to a system segment. The identity of the referenced system segment is established by using the absolute value of the global segment number as an index into the sequence of system segment names.

An E_Vect entry containing a positive global segment number in the range 1..(PED_PRINCIPAL_SEGMENT_COUNT + PED_SUBSIDIARY_SEGMENT_COUNT) indicates a reference to the E_Rec for one of the segments of the environment described in the PED. The E_Rec structure for a given nonsystem segment is located by using the global segment number as an index into an array of E_Rec structures which is allocated at environment reconstruction time. An index value in the range 1..PED_PRINCIPAL_SEGMENT_COUNT selects the E_Rec for one of the principal segments within the execution environment. An index value which is greater than PED_PRINCIPAL_SEGMENT_COUNT specifies a reference to one of the subsidiary segments within the program execution environment.

Following the E_Vect templates in the PED is a sequence of "pseudo-SIB" records. The structure of a pseudo SIB is defined in Figure 6-4. Each pseudo SIB is a template for a fragment of an actual SIB, which must be allocated when the program's execution environment is reconstructed. Thus, there exists one pseudo SIB for each nonsystem segment in the environment. The pseudo SIBs for the principal segments appear first in the sequence followed by the pseudo SIBs for the subsidiary segments in the environment.

```
type

ped_pseudo_sib =
  record
    ps_seg_name: alpha;
                 {Name of segment.}

    ps_seg_leng: integer;
                 {Length of segment.}

    ps_seg_addr: integer;
                 {Relative block
                  address of segment
                  in library file.}

    ps_seg_data_size: integer;
                 {Size of segment
                  data area.}

    ps_seg_lib_num: integer;
                 {Index into sequence
                  of library code
                  file descriptors.}

    ps_seg_attributes:
         packed
         record
           ps_relocatable: boolean;
                  {Relocatable
                   indicator from
                   segment
                   dictionary.}

           ps_mach_type: m_types;
                  {Type of code in
                   segment.}

           ps_filler: 0..2047;
                  {11 bits of filler
                   to round out to
                   one word.}
       end;
  end;
```

Figure 6-4. Pseudo SIB Structure

The information stored in each pseudo SIB structure consists of the information from the segment dictionary entry for the segment which is needed to construct the actual SIB when the program's environment is reconstructed. The PS_SEG_LIB_NUM field of a pseudo SIB is used to establish the identity of the library code file which contains the segment described by the pseudo SIB. This identity is established by using the PS_SEG_LIB_NUM field as an index into the sequence of library file descriptors within the PED. The index value one selects the first library file descriptor in the sequence.

Appendices

APPENDICES

APPENDIX A
P-MACHINE OPCODES
(Alphabetic Order)

Opcode	Dec	Hex	Description
ABI	224	E0	Absolute Value Integer
ABR	227	E3	Absolute Value Real
ADI	162	A2	Add Integer
ADJ	199	C7	Adjust Set
ADR	192	C0	Add Real
ASTR	235	EB	Assign String
BNOT	159	9F	Boolean Not
BPT	158	9E	Break point
CAP	171	AB	Copy Array Parameter
CFP	151	97	Call Formal Procedure
CGP	145	91	Call Global Procedure
CHK	203	CB	Check Subrange Bounds
CIP	146	92	Call Intermediate Procedure
CLP	144	90	Call Local Procedure
CSP	172	AC	Copy String Parameter
CSTR	236	EC	Check String Index
CXG	148	94	Call External Global
CXI	149	95	Call External Intermediate
CXL	147	93	Call External Local
DECI	238	EE	Decrement Integer
DIF	221	DD	Set Difference
DUP1	226	E2	Duplicate One Word
DUPR	198	C6	Duplicate Real
DVI	141	8D	Divide Integer
DVR	195	C3	Divide Real
EFJ	210	D2	Equal False Jump
EQBYT	185	B9	Equal Byte Array
EQPWR	182	B6	Equal Set
EQREAL	205	CD	Equal Real
EQSTR	232	E8	Equal String
EQUI	176	B0	Equal Integer
FJP	212	D4	False Jump
FJPL	213	D5	False Jump Long
FLT	204	CC	Float
GEBYT	187	BB	Greater Than or Equal Byte Array
GEPWR	184	B8	Greater Than or Equal Set
GEQI	179	B3	Greater Than or Equal Integer
GEREAL	207	CF	Greater Than or Equal Real
GESTR	234	EA	Greater Than or Equal String
GEUSW	181	B5	Greater Than or Equal Unsigned
INC	231	E7	Increment
INCI	237	ED	Increment Integer
IND	230	E6	Index
INN	218	DA	Set Membership
INT	220	DC	Set Intersection
IXA	215	D7	Index Array
IXP	216	D8	Index Packed Array
LAE	155	9B	Load Extended Address
LAND	161	A1	Logical And
LAO	134	86	Load Global Address
LCO	130	82	Load Contant Offset
LDA	136	88	Load Intermediate Address
LDB	167	A7	Load Byte
LDC	131	83	Load Constant
LDCB	128	80	Load Constant Byte

LDCI	129	81	Load Constant Integer
LDCN	152	98	Load Constant NIL
LDCRL	242	F2	Load Constant Real
LDE	154	9A	Load Extended
LDL	135	87	Load Local
LDM	208	D0	Load Multiple
LDO	133	85	Load Global
LDP	201	C9	Load Packed
LDRL	243	F3	Load Real
LEBYT	186	BA	Less Than or Equal Byte Array
LEPWR	183	B7	Less Than or Equal Set
LEQI	178	B2	Less Than or Equal Integer
LEREAL	206	CE	Less Than or Equal Real
LESTR	233	E9	Less Than or Equal String
LEUSW	180	B4	Unsigned Less Than or Equal
LLA	132	84	Load Local Address
LNOT	229	E5	Logical Not
LOD	137	89	Load Intermediate
LOR	160	A0	Logical Or
LPR	157	9D	Load Processor Register
LSL	153	99	Load Static Link
MODI	143	8F	Modulo Integers
MOV	197	C5	Move
MPI	140	8C	Multiply Integer
MPR	194	C2	Multiply Real
NAT	168	A8	Enter Native Code
NAT-INFO	169	A9	Native Code Information
NEQI	177	B1	Not Equal Integer
NFJ	211	D3	Not Equal False Jump
NGI	225	E1	Negate Integer
NGR	228	E4	Negate Real
NOP	156	9C	No Operation
RESERVE1	250	FA	reserved
RESERVE2	251	FB	"
RESERVE3	252	FC	"
RESERVE4	253	FD	"
RESERVE5	254	FE	"
RESERVE6	255	FF	"
RND	191	BF	Round Real
RPU	150	96	Return from Procedure
SBI	163	A3	Subtract Integer
SBR	193	C1	Subtract Real
SCPI1	239	EF	Short Call Intermediate Procedure
SCPI2	240	F0	" " " "
SCXG1	112	70	Short Call External Global
SCXG2	113	71	" " " "
SCXG3	114	72	" " " "
SCXG4	115	73	" " " "
SCXG5	116	74	" " " "
SCXG6	117	75	" " " "
SCXG7	118	76	" " " "
SCXG8	119	77	" " " "
SIGNAL	222	DE	Signal
SIND0	120	78	Short Index
SIND1	121	79	" "
SIND2	122	7A	" "
SIND3	123	7B	" "
SIND4	124	7C	" "
SIND5	125	7D	" "
SIND6	126	7E	" "
SIND7	127	7F	" "
SLDC0	0	00	Short Load Constant
SLDC1	1	01	" " "
SLDC2	2	02	" " "
SLDC3	3	03	" " "

SLDC4	4	04	" " "
SLDC5	5	05	" " "
SLDC6	6	06	" " "
SLDC7	7	07	" " "
SLDC8	8	08	" " "
SLDC9	9	09	" " "
SLDC10	10	0A	" " "
SLDC11	11	0B	" " "
SLDC12	12	0C	" " "
SLDC13	13	0D	" " "
SLDC14	14	0E	" " "
SLDC15	15	0F	" " "
SLDC16	16	10	" " "
SLDC17	17	11	" " "
SLDC18	18	12	" " "
SLDC19	19	13	" " "
SLDC20	20	14	" " "
SLDC21	21	15	" " "
SLDC22	22	16	" " "
SLDC23	23	17	" " "
SLDC24	24	18	" " "
SLDC25	25	19	" " "
SLDC26	26	1A	" " "
SLDC27	27	1B	" " "
SLDC28	28	1C	" " "
SLDC29	29	1D	" " "
SLDC30	30	1E	" " "
SLDC31	31	1F	" " "
SLDL1	32	20	Short Load Local
SLDL2	33	21	" " "
SLDL3	34	22	" " "
SLDL4	35	23	" " "
SLDL5	36	24	" " "
SLDL6	37	25	" " "
SLDL7	38	26	" " "
SLDL8	39	27	" " "
SLDL9	40	28	" " "
SLDL10	41	29	" " "
SLDL11	42	2A	" " "
SLDL12	43	2B	" " "
SLDL13	44	2C	" " "
SLDL14	45	2D	" " "
SLDL15	46	2E	" " "
SLDL16	47	2F	" " "
SLDO1	48	30	Short Load Global
SLDO2	49	31	" " "
SLDO3	50	32	" " "
SLDO4	51	33	" " "
SLDO5	52	34	" " "
SLDO6	53	35	" " "
SLDO7	54	36	" " "
SLDO8	55	37	" " "
SLDO9	56	38	" " "
SLDO10	57	39	" " "
SLDO11	58	3A	" " "
SLDO12	59	3B	" " "
SLDO13	60	3C	" " "
SLDO14	61	3D	" " "
SLDO15	62	3E	" " "
SLDO16	63	3F	" " "
SLLA1	96	60	Short Load Local Address
SLLA2	97	61	" " " "
SLLA3	98	62	" " " "
SLLA4	99	63	" " " "

SLLA5	100	64	" " " "
SLLA6	101	65	" " " "
SLLA7	102	66	" " " "
SLLA8	103	67	" " " "
SLOD1	173	AD	Short Load Intermediate
SLOD2	174	AE	" " "
SPR	209	D1	Store Processor Register
SRO	165	A5	Store Global
SRS	188	BC	Subrange Set
SSTL1	104	68	Short Store Local
SSTL2	105	69	" " "
SSTL3	106	6A	" " "
SSTL4	107	6B	" " "
SSTL5	108	6C	" " "
SSTL6	109	6D	" " "
SSTL7	110	6E	" " "
SSTL8	111	6F	" " "
STB	200	C8	Store Byte
STE	217	D9	Store Extended
STL	164	A4	Store Local
STM	142	8E	Store Multiple
STO	196	C4	Store
STP	202	CA	Store Packed
STR	166	A6	Store Intermediate
STRL	244	F4	Store Real
SWAP	189	BD	Swap
TJP	241	F1	True Jump
TNC	190	BE	Truncate Real
UJP	138	8A	Unconditional Jump
USPL	139	8B	Unconditional Jump Long
UNI	219	DB	Set Union
WAIT	223	DF	Wait
XJP	214	D6	Case Jump

APPENDIX B
P-MACHINE OPCODES
(Numeric Order)

Dec	Hex	Opcode	Description
0	00	SLDC0	Short Load Constant
1	01	SLDC1	" " "
2	02	SLDC2	" " "
3	03	SLDC3	" " "
4	04	SLDC4	" " "
5	05	SLDC5	" " "
6	06	SLDC6	" " "
7	07	SLDC7	" " "
8	08	SLDC8	" " "
9	09	SLDC9	" " "
10	0A	SLDC10	" " "
11	0B	SLDC11	" " "
12	0C	SLDC12	" " "
13	0D	SLDC13	" " "
14	0E	SLDC14	" " "
15	0F	SLDC15	" " "
16	10	SLDC16	" " "
17	11	SLDC17	" " "
18	12	SLDC18	" " "
19	13	SLDC19	" " "
20	14	SLDC20	" " "
21	15	SLDC21	" " "
22	16	SLDC22	" " "
23	17	SLDC23	" " "
24	18	SLDC24	" " "
25	19	SLDC25	" " "
26	1A	SLDC26	" " "
27	1B	SLDC27	" " "
28	1C	SLDC28	" " "
29	1D	SLDC29	" " "
30	1E	SLDC30	" " "
31	1F	SLDC31	" " "
32	20	SLDL1	Short Load Local
33	21	SLDL2	" " "
34	22	SLDL3	" " "
35	23	SLDL4	" " "
36	24	SLDL5	" " "
37	25	SLDL6	" " "
38	26	SLDL7	" " "
39	27	SLDL8	" " "
40	28	SLDL9	" " "
41	29	SLDL10	" " "
42	2A	SLDL11	" " "
43	2B	SLDL12	" " "
44	2C	SLDL13	" " "
45	2D	SLDL14	" " "
46	2E	SLDL15	" " "
47	2F	SLDL16	" " "
48	30	SLDO1	Short Load Global
49	31	SLDO2	" " "
50	32	SLDO3	" " "
51	33	SLDO4	" " "
52	34	SLDO5	" " "
53	35	SLDO6	" " "
54	36	SLDO7	" " "

55	37	SLDO8	" " "
56	38	SLDO9	" " "
57	39	SLDO10	" " "
58	3A	SLDO11	" " "
59	3B	SLDO12	" " "
60	3C	SLDO13	" " "
61	3D	SLDO14	" " "
62	3E	SLDO15	" " "
63	3F	SLDO16	" " "
64	40		unused
..			"
95	5F		"
96	60	SLLA1	Short Load Local Address
97	61	SLLA2	" " " "
98	62	SLLA3	" " " "
99	63	SLLA4	" " " "
100	64	SLLA5	" " " "
101	65	SLLA6	" " " "
102	66	SLLA7	" " " "
103	67	SLLA8	" " " "
104	68	SSTL1	Short Store Local
105	69	SSTL2	" " "
106	6A	SSTL3	" " "
107	6B	SSTL4	" " "
108	6C	SSTL5	" " "
109	6D	SSTL6	" " "
110	6E	SSTL7	" " "
111	6F	SSTL8	" " "
112	70	SCXG1	Short Call External Global
113	71	SCXG2	" " " "
114	72	SCXG3	" " " "
115	73	SCXG4	" " " "
116	74	SCXG5	" " " "
117	75	SCXG6	" " " "
118	76	SCXG7	" " " "
119	77	SCXG8	" " " "
120	78	SIND0	Short Index
121	79	SIND1	" "
122	7A	SIND2	" "
123	7B	SIND3	" "
124	7C	SIND4	" "
125	7D	SIND5	" "
126	7E	SIND6	" "
127	7F	SIND7	" "
128	80	LDCB	Load Constant Byte
129	81	LDCI	Load Constant Integer
130	82	LCO	Load Contant Offset
131	83	LDC	Load Word Constant
132	84	LLA	Load Local Address
133	85	LDO	Load Global
134	86	LAO	Load Global Address
135	87	LDL	Load Local
136	88	LDA	Load Intermediate Address
137	89	LOD	Load Intermediate
138	8A	UJP	Unconditional Jump
139	8B	UJPL	Unconditional Jump Long
140	8C	MPI	Multiply Integer
141	8D	DVI	Divide Integer
142	8E	STM	Store Multiple
143	8F	MODI	Modulo Integers
144	90	CLP	Call Local Procedure
145	91	CGP	Call Global Procedure
146	92	CIP	Call Intermediate Procedure
147	93	CXL	Call External Local

148	94	CXG	Call External Global
149	95	CXI	Call External Intermediate
150	96	RPU	Return from Procedure
151	97	CFP	Call Formal Procedure
152	98	LDCN	Load Constant NIL
153	99	LSL	Load Static Link
154	9A	LDE	Load Extended
155	9B	LAE	Load Extended Address
156	9C	NOP	No Operation
157	9D	LPR	Load Processor Register
158	9E	BPT	Break point
159	9F	BNOT	Boolean Not
160	A0	LOR	Logical Or
161	A1	LAND	Logical And
162	A2	ADI	Add Integer
163	A3	SBI	Subtract Integer
164	A4	STL	Store Local
165	A5	SRO	Store Global
166	A6	STR	Store Intermediate
167	A7	LDB	Load Byte
168	A8	NAT	Enter Native Code
169	A9	NAT-INFO	Native Code Information
170	AA		reserved
171	AB	CAP	Copy Array Parameter
172	AC	CSP	Copy String Parameter
173	AD	SLOD1	Short Load Intermediate
174	AE	SLOD2	" " "
175	AF		unused
176	B0	EQUI	Equal Integer
177	B1	NEQI	Not Equal Integer
178	B2	LEQI	Less Than or Equal Integer
179	B3	GEQI	Greater Than or Equal Integer
180	B4	LEUSW	Less Than or Equal Unsigned
181	B5	GEUSW	Greater Than or Equal Unsigned
182	B6	EQPWR	Equal Set
183	B7	LEPWR	Less Than or Equal Set
184	B8	GEPWR	Greater Than or Equal Set
185	B9	EQBYT	Equal Byte Array
186	BA	LEBYT	Less Than or Equal Byte Array
187	BB	GEBYT	Greater Than or Equal Byte Array
188	BC	SRS	Subrange Set
189	BD	SWAP	Swap
190	BE	TNC	Truncate Real
191	BF	RND	Round Real
192	C0	ADR	Add Real
193	C1	SBR	Subtract Real
194	C2	MPR	Multiply Real
195	C3	DVR	Divide Real
196	C4	STO	Store
197	C5	MOV	Move
198	C6	DUPR	Duplicate Real
199	C7	ADJ	Adjust Set
200	C8	STB	Store Byte
201	C9	LDP	Load Packed
202	CA	STP	Store Packed
203	CB	CHK	Check Subrange Bounds
204	CC	FLT	Float
205	CD	EQREAL	Equal Real
206	CE	LEREAL	Less Than or Equal Real
207	CF	GEREAL	Greater Than or Equal Real
208	D0	LDM	Load Multiple
209	D1	SPR	Store Processor Register
210	D2	EFJ	Equal False Jump
211	D3	NFJ	Not Equal False Jump

212	D4	FJP	False Jump
213	D5	FJPL	False Long Jump
214	D6	XJP	Case Jump
215	D7	IXA	Index Array
216	D8	IXP	Index Packed Array
217	D9	STE	Store Extended
218	DA	INN	Set Membership
219	DB	UNI	Set Union
220	DC	INT	Set Intersection
221	DD	DIF	Set Difference
222	DE	SIGNAL	Signal
223	DF	WAIT	Wait
224	E0	ABI	Absolute Value Integer
225	E1	NGI	Negate Integer
226	E2	DUP1	Duplicate One Word
227	E3	ABR	Absolute Value Real
228	E4	NGR	Negate Real
229	E5	LNOT	Logical Not
230	E6	IND	Index
231	E7	INC	Increment
232	E8	EQSTR	Equal String
233	E9	LESTR	Less Than or Equal String
234	EA	GESTR	Greater Than or Equal String
235	EB	ASTR	Assign String
236	EC	CSTR	Check String Index
237	ED	INCI	Increment Integer
238	EE	DECI	Decrement Integer
239	EF	SCPI1	Short Call Intermediate Procedure
240	F0	SCPI2	" " " "
241	F1	TJP	True Jump
242	F2	LDCRL	Load Constant Real
243	F3	LDRL	Load Real
244	F4	STRL	Store Real
245	F5		unused
...			"
249	F9		"
250	FA	RESERVE1	reserved
251	FB	RESERVE2	"
252	FC	RESERVE3	"
253	FD	RESERVE4	"
254	FE	RESERVE5	"
255	FF	RESERVE6	"

APPENDIX C
P-MACHINE INTRINSICS

Kernel procedures provided in the p-machine emulator.

```
 1 - reserved
 2 -    "
 3 -    "
 4 - Procedure RELOCSEG( seg_erec:erec_p );
 5 - reserved
 6 -    "
 7 -    "
 8 -    "
 9 -    "
10 -    "
11 -    "
12 -    "
13 -    "
14 - Procedure MOVESEG( seg_sib:sib_p; src_pool:poolptr; src_offset:memptr );
15 - Procedure MOVELEFT( var source:bytearray; var dest:bytearray;
                         n_bytes:integer );
16 - Procedure MOVERIGHT( var source:bytearray; var dest:bytearray;
                          n_bytes:integer );
17 - reserved
18 - Procedure UNITREAD( unit:unitnum; var buff:bytearray;
                         n_bytes, block, control:integer;
19 - Procedure UNITWRITE( unit:unitnum; var buff:bytearray;
                          n_bytes, block, control:integer;
20 - Procedure TIME( var hi_word, lo_word:integer );
21 - Procedure FILLCHAR( var dest:bytearray; n_bytes, value:integer );

(The SCAN parameters shown here do not match those in an actual SCAN call:
 the Pascal compiler generates the appropriate parameters.)

22 - Function SCAN( disp:integer; not_equal:boolean; target:byte;
                    var a:bytearray; start_index, mask:integer ):
23 - Procedure IOCHECK;
24 - Procedure GETPOOLBYTES( var dest:bytearray; pooldesc:poolptr;
                            pooloffset:memptr; nbytes:integer );
25 - Procedure PUTPOOLBYTES( source:bytearray; pooldesc:poolptr;
                            pooloffset:memptr; nbytes:integer );
26 - Procedure FLIPSEGBYTES( seg_erec:erec_p; seg_offset, n_words:integer );
27 - Procedure QUIET;
28 - Procedure ENABLE;
29 - Procedure ATTACH( sem:sem_p; vector:integer );
30 - Function  IORESULT:integer;
31 - Function  UNITBUSY( unit:unitnum ):boolean;
32 - Function  POWEROFTEN( arg:integer ):real;
33 - Procedure UNITWAIT( unit:unitnum );
34 - Procedure UNITCLEAR( unit:unitnum );
35 - reserved
36 - Procedure UNITSTATUS( unit:unitnum; var stat_rec:status_rec;
                          control:integer );
37 - Procedure IDSEARCH( var symcursor:symrec; var buffer:bytearray );
38 - Function  TREESEARCH( root:treerec_p; var node:treerec;
                          name:alpha ):integer;
39 - Function  READSEG( seg_erec:erec_p ):integer;
```

APPENDIX D
PASCAL DEFINITIONAL RSP

```
{$U-}
Unit Globals;

Interface

Uses    {$u opsys:kernel.code} kernel( {types} syscomrec, bytearray );

Type    syscom_ptr=^syscomrec;
        statrec=array[0..29] of integer;
        statctrl=packed record
                   io_direction:(output_status,input_status);
                   reserved:array[1..12] of boolean;
                   user_defined:array[13..15] of boolean;
                 end;
        rwctrl=packed record
                  async:boolean;
                  physsect:boolean;
                  no_spec:boolean;
                  no_crlf:boolean;
                  reserved:array[4..12] of boolean;
                  user_defined:array[13..15] of boolean;
                end;
        word_array=array[0..0] of integer;
        char_array=packed array[0..0] of char;
        p_memory=record case integer of
                   0:( i:integer );
                   1:( c:char );
                   2:( pb:^byte_array );
                   3:( pw:^word_array );
                   4:( pc:^char_array );
                 end;
        disk_inforec=record
                       tracks_per_disk:integer;
                       sectors_per_track:integer;
                       bytes_per_sector:integer;
                       interleave:integer;
                       first_pascal_track:integer;
                       track_skew:integer;
                     end {disk_inforec};

Procedure PME_Signal_Event( event_num:integer );
Procedure PME_Got_Break;

Implementation

End {Globals}.

{ST PASCAL DEFINITIONAL RUN-TIME SUPPORT PACKAGE }
{$U-}
{$D ASYNCHRONOUS-}

Unit RSP;

Interface

Uses    {$U opsys:kernel.code} kernel( { const } DLE,
                                       { types } iorsltwd,
                                       bytearray, full_address,
                                       unitnum, utable, syscomrec,
                                       { vars } syscom ),
        {$U globals.code} globals;

Procedure UNITCLEAR( unit_no:unitnum );
Procedure UNITSTATUS( unit_no:unitnum; var status:statrec; control:statctrl );
```

```
(Buffer parameter declarations for UNITREAD and UNITWRITE, below, do not
 reflect the way Pascal programs use these procedures.  In an actual
 call, the Pascall compiler generates the appropriate parameters.  The
 byte-pointer has been separated into two parameters so that the RSP
 can handle them individually.)

Procedure UNITREAD( unit_no:unitnum; buffer:p_memory; index:integer;
                    length:integer; block:integer; control:rwctrl );
Procedure UNITWRITE( unit_no:unitnum; buffer:p_memory; index:integer;
                     length:integer; block:integer; control:rwctrl );
Procedure SYSREAD( unit_no:unitnum; buffer:p_memory; index:integer;
                   length:integer; block:integer; control:rwctrl;
                   codepool:full_address );
Function  UNITBUSY( unit_no:unitnum ):boolean;
Procedure UNITWAIT( unit_no:unitnum );
Function  IORESULT:iorsltwd;
Procedure IOCHECK;
Procedure QUIET;
Procedure ENABLE;
Procedure TIME( var hiword, loword:integer );

{$P}
Implementation

{Note that actual calls to the bios are between two assembly language routines
 and parameters may not be passed as in pascal procedures, in particular
 parameters and results are often in registers. }

Uses    ($u bios.code) BIOS;

Const   max_char_vols=29;       { console,printer,remote & 26 serial vols }
        lf=10;                  {ASCII linefeed}
        cr=13;                  {ASCII carriage return}

Type    unit_types=(bad,sys,con,dsk,prn,rem,ser,usr);
        op_types=(read_op,write_op,init_op,status_op);

Var     event_code:p_memory;    {filled in by interpreter}
        break_code:p_memory;    {filled in by interpreter}
        alpha_locked:packed array[1..max_char_vols] of boolean;
        dle_rcvd:packed array[1..max_char_vols] of boolean;
{$P}
Function Translate( unit_no:unitnum; operation:op_types;
                    var block:integer; var blocked:boolean; var device:integer;
                    var flags:integer; var u_type:unit_types ):iorsltwd;

  procedure trans_err( err:iorsltwd );
  begin
    translate:=err;
    u_type:=bad;
    exit(translate);
  end;

begin
  { if an error occurs in translate then u_type will be "bad" and
    the function result will indicate what error occurred. }
  u_type:=bad;
  translate:=i_no_error;
  if unit_no<=8
    then begin
           device:=0;
           case unit_no of
             0:   u_type:=sys;
             1,2: u_type:=con;
             3:   trans_err( i_bad_unit );
             4,5: begin
```

```
                    u_type:=dsk;
                    device:=unit_no-4;
                  end;
            6:    u_type:=prn;
            7: if operation=write_op   {remin}
                  then trans_err( i_bad_mode )
                  else u_type:=rem;
            8: if operation=read_op    {remout}
                  then trans_err( i_bad_mode )
                  else u_type:=rem;
          end {case};
        end
    else if unit_no<(syscom^.subsidstart)
          then begin
                 u_type:=dsk;
                 device:=unit_no-9+2;
               end
          else if unit_no<(syscom^.subsidstart +
                           syscom^.unitdivision.subsidmax)
                 then with syscom^.unitable^[unit_no] do
                        begin
                          if ueovblk=0 then trans_err( i_no_unit );
                          if block>=ueovblk then trans_err( i_ill_block );
                          u_type:=dsk;
                          unit_no:=uphysvol;
                          if unit_no>=9 then device:=unit_no-9+2
                                        else device:=unit_no-4;
                          block:=block+ublkoff;
                          translate:=i_no_error;
                        end
                 else if unit_no<(syscom^.subsidstart+
                                  syscom^.unitdivision.subsidmax+
                                  syscom^.unitdivision.serialmax)
                        then begin
                               u_type:=ser;
                               device:=unit_no-(syscom^.subsidstart+
                                        syscom^.unitdivision.subsidmax);
                             end
                        else if unit_no>=128
                               then begin
                                      u_type:=usr;
                                      device:=unit_no-128;
                                    end
                               else trans_err( i_bad_unit );
  blocked:=u_type in [sys,dsk,usr];
  case u_type of
    con: flags:=1;
    rem: flags:=2;
    prn: flags:=3;
    ser: flags:=4+device;
  end {case};
end {translate};

{$P}
Procedure UNITREAD( unit_no:unitnum; buffer:p_memory; index:integer;
                    length:integer; block:integer; control:rwctrl );
var     stack_addr:fulladdress;
begin
  stack_addr[0]:=0;
  stack_addr[1]:=0;
  SYSREAD( unit_no, buffer, index, length, block, control, stack_addr );
end {UNITREAD};

Procedure SYSREAD( unit_no:unitnum; buffer:p_memory; index:integer;
                   length:integer; block:integer; control:rwctrl;
                   codepool:fulladdress );
```

```
var     device,flags:integer;
        blocked:boolean;
        u_type:unit_types;
        iorslt:iorsltwd;

  Function Char_In:iorsltwd;
  label 100;
  var   ch:char;
        iorslt:iorsltwd;

    Procedure check_result;
    begin
      if iorslt<>i_no_error
        then begin
               char_in:=iorslt;
               exit(char_in);
             end;
    end {check_result};

    procedure echo_char( ch:char );
    begin
          iorslt:=bios_con_write( ch );
          check_result;
          if (ch=chr(cr)) and (not control.no_crlf)
            then echo_char( chr(lf) );
        end {echo_char};

      begin
        while length>0 do
          begin
            case u_type of
              con: iorslt:=bios_con_read( ch );
              prn: iorslt:=bios_prn_read( ch );
              rem: iorslt:=bios_rem_read( ch );
              ser: iorslt:=bios_ser_read( device, ch );
            end;
            check_result;
            if not control.no_spec then
              begin
                if ch=syscom^.crtinfo.eof
                  then begin
                         if unit_no=1
                           then fillchar( buffer.pc^[index], length, chr(0) )
                           else buffer.pc^[index]:=ch;
                         exit(char_in);
                       end;
                if ch=syscom^.crtinfo.alphalok
                  then begin
                         alpha_locked[flags]:=not alpha_locked[flags];
                         goto 100;
                       end;
              end {not no_spec};
            if alpha_locked[flags]
              then if ch in [
                     then ch:=chr( ord(ch)-32 );
            if unit_no=1 then echo_char( ch );
            buffer.pc^[index]:=ch;
            index:=index+1;
            length:=length-1;
            100:
          end {while length>0};
        char_in:=iorslt;
      end {char_in};
```

```
begin
  syscom^.iorslt:=translate( unit_no, read_op, block, blocked,
                             device, flags, u_type );
  if syscom^.iorslt<>i_no_error then exit(SYSREAD);
  if blocked
    then case u_type of
           sys: syscom^.iorslt:=bios_sys_read( block, length, buffer, index,
                                               device, control );
           dsk: syscom^.iorslt:=bios_dsk_read( block, length, buffer, index,
                                               device, control, codepool );
           usr: syscom^.iorslt:=bios_usr_read( block, length, buffer, index,
                                               device, control );
         end {case}
    else syscom^.iorslt:=char_in;
end {SYSREAD};

{$P}
Procedure UNITWRITE( unit_no:unitnum; buffer:p_memory; index:integer;
                     length:integer; block:integer; control:rwctrl );
var     device,flags:integer;
        blocked:boolean;
        u_type:unit_types;
        iorslt:iorsltwd;

  Function Char_Out:iorsltwd;
  var   ch:char;
        iorslt:iorsltwd;
        i:integer;

    Procedure check_result;
    begin
      if iorslt<>i_no_error
        then begin
               char_out:=iorslt;
               exit(char_out);
             end;
    end {check_result};

    Procedure Send_Char( ch:char );
    begin
      case u_type of
        con: iorslt:=bios_con_write( ch );
        prn: iorslt:=bios_prn_write( ch );
        rem: iorslt:=bios_rem_write( ch );
        ser: iorslt:=bios_ser_write( device, ch );
      end;
      check_result;
      if (ch=chr(cr)) and (not control.no_crlf)
        then send_char( chr(lf) );
    end {send_char};

  begin
    while length>0 do
      begin
        ch:=buffer.pc^[index];
        if control.no_spec
          then send_char( ch )
          else if dle_rcvd[flags]
                 then begin
                        for i:=1 to ord(ch)-32 do send_char(
                        dle_rcvd[flags]:=false;
                      end
                 else if ch=chr(dle)
                        then dle_rcvd[flags]:=true
                        else send_char( ch );
        index:=index+1;
        length:=length-1;
      end {while length>0};
    char_out:=iorslt;
```

```
  end {char_out};

begin
  syscom^.iorslt:=translate( unit_no, write_op, block, blocked,
                             device, flags, u_type );
  if syscom^.iorslt<>i_no_error then exit(UNITWRITE);
  if blocked
    then case u_type of
           sys: syscom^.iorslt:=bios_sys_write( block, length, buffer, index,
                                                device, control );
           dsk: syscom^.iorslt:=bios_dsk_write( block, length, buffer, index,
                                                device, control );
           usr: syscom^.iorslt:=bios_usr_write( block, length, buffer, index,
                                                device, control );
         end {case}
    else syscom^.iorslt:=char_out;
end {UNITWRITE};

{$P}
Procedure UNITCLEAR( unit_no:unitnum );
var     block,device,flags:integer;
        blocked:boolean;
        u_type:unit_types;
begin
  syscom^.iorslt:=translate( unit_no, init_op, block, blocked,
                             device, flags, u_type );
  if syscom^.iorslt<>i_no_error then exit(UNITCLEAR);
  case u_type of
    sys: syscom^.iorslt:=bios_sys_init( device, PME_signal_event );
    con: syscom^.iorslt:=bios_con_init( PME_got_break, syscom );
    dsk: syscom^.iorslt:=bios_dsk_init( device );
    prn: syscom^.iorslt:=bios_prn_init;
    rem: syscom^.iorslt:=bios_rem_init;
    ser: syscom^.iorslt:=bios_ser_init( device );
    usr: syscom^.iorslt:=bios_usr_init( device );
  end {case};
  if not blocked
    then begin
           alpha_locked[flags]:=false;
           dle_rcvd[flags]:=false;
         end;
end {UNITCLEAR};

{$P}
Procedure UNITSTATUS( unit_no:unitnum; var status:statrec; control:statctrl );
var     block,device,flags:integer;
        blocked:boolean;
        u_type:unit_types;
begin
  syscom^.iorslt:=translate( unit_no, status_op, block, blocked,
                             device, flags, u_type );
  if syscom^.iorslt<>i_no_error then exit(UNITSTATUS);
  case u_type of
    sys: syscom^.iorslt:=bios_sys_stat( status, control );
    con: syscom^.iorslt:=bios_con_stat( status, control );
    dsk: syscom^.iorslt:=bios_dsk_stat( device, status, control );
    prn: syscom^.iorslt:=bios_prn_stat( status, control );
    rem: syscom^.iorslt:=bios_rem_stat( status, control );
    ser: syscom^.iorslt:=bios_ser_stat( device, status, control );
    usr: syscom^.iorslt:=bios_usr_stat( device, status, control );
  end {case};
end {UNITSTATUS};
```

```
{$P}
Function  UNITBUSY( unit_no:unitnum ):boolean};
var     in_status,
        out_status: statrec;
        control_word: statctrl;
begin
  {$B ASYNCHRONOUS+}
  control_word.io_direction:=input_status;
  unitstatus( unit_no, in_status, control_word );
  control_word.io_direction:=output_status;
  unitstatus( unit_no, out_status, control_word );
  unitbusy:= (in_status[0]<>0) or (out_status[0]<>0);
  {$E ASYNCHRONOUS+}
  {$B ASYNCHRONOUS-}
  unitbusy:=false;
  {$E ASYNCHRONOUS-}
end {UNITBUSY};

Procedure UNITWAIT( unit_no:unitnum );
var     in_status,
        out_status: statrec;
        in_ctrl_word,
        out_ctrl_word: statctrl;
begin
  {$B ASYNCHRONOUS+}
  in_ctrl_word.io_direction:=input_status;
  out_ctrl_word.io_direction:=output_status;
  repeat
    unitstatus( unit_no, in_status, in_ctrl_word );
    unitstatus( unit_no, out_status, out_ctrl_word );
  until (in_status[0]=0) and (out_status[0]=0);
  {$E ASYNCHRONOUT+}
  {$B ASYNCHRONOUS-}
  {unitwait does nothing on synchronous systems}
  {$E ASYNCHRONOUS-}
end {UNITWAIT};

{$P}
Function IORESULT{:iorsltwd};
begin
  ioresult:=syscom^.iorslt;
end {IORESULT};

Procedure IOCHECK;
begin
  if syscom^.iorslt<>i_no_error then { exec_error( 10 ) };
end {IOCHECK};

Procedure QUIET;
begin
  bios_quiet;
end {QUIET};

Procedure ENABLE;
begin
  bios_enable;
end {ENABLE};

Procedure TIME( var hiword, loword:integer );
Var     status: statrec;
        control: statctrl;
begin
  UNITSTATUS(0,status,control);
  loword:=status[1];
  hiword:=status[2];
end {TIME};

End {RSP}.
```

APPENDIX E
PASCAL DEFINITIONAL BIOS

```
{$T PASCAL DEFINITIONAL BASIC INPUT/OUTPUT SYSTEM }
{$U-}

Unit BIOS;

Interface

Uses    {$u opsys:kernel.code} KERNEL( { types } full_address, bytearray,
                                       iorsltwd, syscomrec ),
        {$u globals.code} globals;

Var     curform:disk_inforec;           {initialized by bootstrap}
        sect_per_block:integer;         {initialized by bootstrap}
        hi_addr:p_memory;               {initialized by bootstrap}
        sector_buffer:p_memory;         {initialized by bootstrap}
        sect_trans:p_memory;            {initialized by bootstrap}

Function bios_con_read( var ch:char ):iorsltwd;
Function bios_con_write( ch:char ):iorsltwd;
Function bios_con_init( procedure got_break; syscom:syscom_ptr ):iorsltwd;
Function bios_con_stat( var status:statrec; control:statctrl ):iorsltwd;

Function bios_prn_read( var ch:char ):iorsltwd;
Function bios_prn_write( ch:char ):iorsltwd;
Function bios_prn_init:iorsltwd;
Function bios_prn_stat( var status:statrec; control:statctrl ):iorsltwd;

Function bios_dsk_read( block:integer; length:integer; buffer:p_memory;
                        index, drive:integer; control:rwctrl;
                        codepool:fulladdress ):iorsltwd;
Function bios_dsk_write( block:integer; length:integer; buffer:p_memory;
                         index, drive:integer; control:rwctrl ):iorsltwd;
Function bios_dsk_init( drive:integer ):iorsltwd;
Function bios_dsk_stat( drive:integer; var status:statrec;
                        control:statctrl ):iorsltwd;

Function bios_rem_read( var ch:char ):iorsltwd;
Function bios_rem_write( ch:char ):iorsltwd;
Function bios_rem_init:iorsltwd;
Function bios_rem_stat( var status:statrec; control:statctrl ):iorsltwd;

Function bios_usr_read( block:integer; length:integer; buffer:p_memory;
                        index, device:integer; control:rwctrl ):iorsltwd;
Function bios_usr_write( block:integer; length:integer; buffer:p_memory;
                        index, device:integer; control:rwctrl ):iorsltwd;
Function bios_usr_init( device:integer ):iorsltwd;
Function bios_usr_stat( device:integer; var status:statrec;
                        control:statctrl ):iorsltwd;

Function bios_sys_read( block:integer; length:integer; buffer:p_memory;
                        index, device:integer; control:rwctrl ):iorsltwd;

Function bios_sys_write( block:integer; length:integer; buffer:p_memory;
                         index, device:integer; control:rwctrl ):iorsltwd;
Function bios_sys_init( device:integer;
                        procedure signal_event( event_num:integer ) ):iorsltwd;
Function bios_sys_stat( var status:statrec; control:statctrl ):iorsltwd;

Procedure bios_quiet;
Procedure bios_enable;

Function bios_ser_read( device:integer; var ch:char ):iorsltwd;
Function bios_ser_write( device:integer; ch:char ):iorsltwd;
Function bios_ser_init( device:integer ):iorsltwd;
Function bios_ser_stat( device:integer; var status:statrec;
                        control:statctrl ):iorsltwd;
```

Appendix E

```
Procedure disk_change( var newform:disk_inforec ); {called by bootstrap}

Implementation

Uses    {$u sbios.code} SBIOS;

Const   q_size=64;
        q_empty=0;
        bell=7;

Type    q_rec=record
                bottom,top:integer;
                iorslt:iorsltwd;
                ringbell:boolean;
                data:packed array[1..q_size] of char;
              end;

Var     save_syscom_ptr:syscom_ptr;
        { save_break_proc:procedure; }  {can
        flush_flag,stop_flag:boolean;

        con_queue:q_rec;
        prn_queue:q_rec;
        rem_queue:q_rec;

Procedure pollunits; forward;

Procedure q_init( var q:q_rec );
begin
  {set up empty queue}
  q.bottom:=q_size;
  q.top:=q_empty;
  q.iorslt:=i_noerror;
end {q_init};

Function q_count( q:q_rec ):integer;
begin
  if q.top>q.bottom then q_count:=q.top-q.bottom
                    else q_count:=q_size-q.bottom+q.top;
end {q_count};

Function q_get( var q:q_rec ):char;
begin
  repeat
    pollunits;
  until q_count( q )>0;
  if q.bottom=q_size
    then q.bottom:=1
    else q.bottom:=q.bottom+1;
  q_get:=q.data[q.bottom];
  if q.bottom=q.top then q_init( q );
end {q_get};

Procedure q_put( var q:q_rec; ch:char );
var old_top:integer;
begin
  old_top:=q.top;
  if q.top=q_size
    then q.top:=1
    else q.top:=q.top+1;
  if q.top=q.bottom
    then begin {overflow}
           q.top:=old_top;
           if q.ringbell
             then q.iorslt:=conwrit( chr(bell) );
           exit(q_put);
         end;
  q.data[q.top]:=ch;
end {q_put};

Procedure pollunits;
var char_rdy:boolean;
    ch:char;
```

```
Function special_char( var ch:char ):boolean;
{only used on console}
begin
  with save_syscom_ptr^.crtinfo do
    begin
      special_char:=true;
      ch:=chr(ord( odd(ord(ch)) and
                   odd(ord(char_mask)) ));
      if ch=break
        then begin
               {call save_break_proc}
               stop_flag:=false;
               flush_flag:=false;
             end
        else if ch=stop
               then stop_flag:=not stop_flag
               else if ch=flush
                      then begin
                             flush_flag:=not flush_flag;
                             stop_flag:=false;
                           end
                      else special_char:=false;
    end;

    end {special_char};

  begin
    with con_queue do
      begin
        iorslt:=constat( char_rdy );
        if char_rdy and (iorslt=i_noerror)
          then begin
                 iorslt:=conread( ch );
                 if iorslt=i_noerror
                   then if not special_char( ch )
                          then q_put( con_queue, ch );
               end;
      end;
    with prn_queue do
      begin
        iorslt:=prnstat( char_rdy );
        if char_rdy and (iorslt=i_noerror)
          then begin
                 iorslt:=prnread( ch );
                 if iorslt=i_noerror
                   then q_put( prn_queue, ch );
               end;
      end;
    with rem_queue do
      begin
        iorslt:=remstat( char_rdy );
        if char_rdy and (iorslt=i_noerror)
          then begin
                 iorslt:=remread( ch );
                 if iorslt=i_noerror
                   then q_put( rem_queue, ch );
               end;
      end;
  end {pollunits};

  Function bios_con_read( var ch:char ):iorsltwd;
  begin
    flush_flag:=false;
    stop_flag:=false;
    ch:=q_get( con_queue );
    bios_con_read:=con_queue.iorslt;
  end;
```

```
 Function bios_con_write( ch:char ):iorsltwd);
 begin
   bios_con_write:=i_no_error;
   repeat
     pollunits;
     if flush_flag then exit(bios_con_write);
   until stop_flag=false;
   bios_con_write:=conwrit( ch );
 end;

 Function bios_con_init( procedure got_break; syscom:syscom_ptr ):iorsltwd);
 begin

  { Note:  the address of got_break procedure should be saved at this point,
    this cannot be shown in Pascal; however, the following describes what is
    actually done in a bios:
  save_break_proc:= got_break; }
  save_syscom_ptr:=syscom;       (save syscom pointer)
  flush_flag:=false;
  stop_flag:=false;
  q_init( con_queue );
  con_queue.ringbell:=true;
  bios_con_init:=coninit;
end;

Function bios_con_stat( var status:statrec; control:statctrl ):iorsltwd);
begin
  pollunits;
  if control.io_direction=input_status
    then status[0]:=q_count( con_queue )
    else status[0]:=0;
  bios_con_stat:=con_queue.iorslt;
end;

Function bios_prn_read( var ch:char ):iorsltwd);
begin
  ch:=q_get( prn_queue );
  bios_prn_read:=prn_queue.iorslt;
end;

Function bios_prn_write( ch:char ):iorsltwd);
begin
  pollunits;
  bios_prn_write:=prnwrit( ch );
end;

Function bios_prn_init(:iorsltwd);
begin
  q_init( prn_queue );
  prn_queue.ringbell:=false;
  bios_prn_init:=prninit;
end;

Function bios_prn_stat( var status:statrec; control:statctrl ):iorsltwd);
begin
  pollunits;
  if control.io_direction=input_status
    then status[0]:=q_count( prn_queue )
    else status[0]:=0;
  bios_prn_stat:=prn_queue.iorslt;
end;

Procedure disk_change( var newform:disk_inforec );
const   blksize=512;
var     phys_sector,logic_sector,tryit:integer;
        used:boolean;
begin
  curform:=newform;
  sect_per_block:=blksize div curform.bytes_per_sector;
```

```
  {initialize sector_translation table}
  phys_sector:=0;
  for logic_sector:=0 to curform.sectors_per_track-1 do
    begin
      repeat
        if phys_sector>=curform.sectors_per_track
          then phys_sector:=phys_sector-curform.sectors_per_track;
        used:=false;
        for tryit:=0 to logic_sector-1 do
          if sect_trans.pb^[tryit]=phys_sector then used:=true;
        if used then phys_sector:=phys_sector+1;
      until not used;
      sect_trans.pb^[logic_sector]:=phys_sector;
      phys_sector:=phys_sector+curform.interleave;
    end;
end {disk_change};

Function disk_rw( block:integer; length:integer; buffer:p_memory;
                  index, drive:integer; control:rwctrl;
                  baseaddr:fulladdress; read_disk:boolean ):iorsltwd;
label   10,99;
const   max_retry=5;
var     partial:boolean;
        retry,rel_sector,logic_sector,logic_track,skew,
        sector,track:integer;
        stackbase:fulladdress;
        iorslt:iorsltwd;

  procedure calc_sect;
  {calculate physical sector address from logical sector address}
  begin
    sector:=( (sect_trans.pb^[logic_sector]+skew) mod
              curform.sectors_per_track )+1;
  end;

  procedure convert;
  {convert block parameter to track and sector address}
  begin
    if control.physsect
      then begin
             length:=curform.bytes_per_sector;
             track:=block div curform.sectors_per_track;
             sector:=(block mod curform.sectors_per_track)+1;
           end
      else begin
             rel_sector:=block*sect_per_block;
             logic_track :=rel_sector div curform.sectors_per_track;
             logic_sector:=rel_sector mod curform.sectors_per_track;
             track:=logic_track+curform.first_pascal_track;
             skew:=logic_track*curform.track_skew;
             calc_sect;
           end;
  end {convert};

begin
  setdisk( drive );
partial:=false;
convert;
dskstrt;
settrak( track );
while length>0 do
  begin
    retry:=max_retry;
    if track>curform.tracks_per_disk
      then begin
             iorslt:=i_ill_block;
             goto 99;
           end;
    if (length<curform.bytes_per_sector) and read_disk
      then begin
             partial:=true;
             stackbase[0]:=0;
             stackbase[1]:=0;
             setbufr( sector_buffer, 0, stackbase );
           end
```

```
    else setbufr( buffer, index, baseaddress );
  setsect( sector );
  10:
  pollunits;
  if read_disk then iorslt:=dskread
               else iorslt:=dskwrit;
  if iorslt<>i_no_error
    then begin
           if iorslt<>i_bad_block then goto 99;
           if retry=max_retry
             then begin
                    iorslt:=dskinit;
                    if iorslt=i_no_unit then goto 99;
                  end;
           retry:=retry-1;
           if retry>0 then goto 10;
           goto 99;
         end;
  if partial
    then begin
           {Note:  when copying partial sector buffer, destination
            address also includes "baseaddr" which is not shown below}
           moveleft( sector_buffer.pb^[0], buffer.pb^[index], length );
           goto 99;
         end;
  index:=index+curform.bytes_per_sector;
  length:=length-curform.bytes_per_sector;
  logic_sector:=logic_sector+1;
  if logic_sector>=curform.sectors_per_track
    then begin
           skew:=skew+curform.track_skew;
           track:=track+1;
           settrak( track );
           logic_sector:=0;
         end;
  calc_sect;
end {while length>0};
  99:
  dskstop;
  disk_rw:=iorslt;
end {disk_rw};

Function bios_dsk_read( block:integer; length:integer; buffer:p_memory;
                        index, drive:integer; control:rwctrl;
                        codepool:fulladdress ):iorsltwd};
begin
  bios_dsk_read:=disk_rw( block, length, buffer, index, drive, control,
                          codepool, true );
end;

Function bios_dsk_write( block:integer; length:integer; buffer:p_memory;
                         index, drive:integer; control:rwctrl ):iorsltwd};
var     baseaddr:fulladdress;
begin
  baseaddr[0]:=0;
  baseaddr[1]:=0;
  bios_dsk_write:=disk_rw( block, length, buffer, index, drive, control,
                           baseaddr, false );
end;

Function bios_dsk_init( drive:integer ):iorsltwd};
begin
  setdisk( drive );
  dskstrt;
  bios_dsk_init:=dskinit;
  dskstop;
end;

Function bios_dsk_stat( drive:integer; var status:statrec;
                        control:statctrl ):iorsltwd};
begin
  setdisk( drive );     {setdisk might call disk_change}
  status[0]:=0;
  status[1]:=curform.bytes_per_sector;
  status[2]:=curform.sectors_per_track;
  status[3]:=curform.tracks_per_disk;
  bios_dsk_stat:=i_no_error;
end;
```

```
Function bios_rem_read( var ch:char ):iorsltwd);
begin
  ch:=q_get( rem_queue );
  bios_rem_read:=rem_queue.iorslt;
end;

Function bios_rem_write( ch:char ):iorsltwd);
begin
  pollunits;
  bios_rem_write:=remwrit( ch );
end;

Function bios_rem_init(:iorsltwd);
begin
  q_init( rem_queue );
  rem_queue.ringbell:=false;
  bios_rem_init:=reminit;
end;

Function bios_rem_stat( var status:statrec; control:statctrl ):iorsltwd);
begin
  pollunits;
  if control.io_direction=input_status
    then status[0]:=q_count( rem_queue )
    else status[0]:=0;
  bios_rem_stat:=rem_queue.iorslt;
end;

Function bios_usr_read( block:integer; length:integer; buffer:p_memory;
                        index, device:integer; control:rwctrl ):iorsltwd);
begin
  bios_usr_read:=usrread( block, length, buffer, index, device, control );
end;

Function bios_usr_write( block:integer; length:integer; buffer:p_memory;
                        index, device:integer; control:rwctrl ):iorsltwd);
begin
  bios_usr_write:=usrwrit( block, length, buffer, index, device, control );
end;

Function bios_usr_init( device:integer ):iorsltwd);
begin
  bios_usr_init:=usrinit( device );
end;

Function bios_usr_stat( device:integer; var status:statrec;
                        control:statctrl ):iorsltwd);
begin
  bios_usr_stat:=usrstat( device, status, control );
end;

Function bios_sys_read( block:integer; length:integer; buffer:p_memory;
                        index, device:integer; control:rwctrl ):iorsltwd);
begin
  syshalt;
end;

Function bios_sys_write( block:integer; length:integer; buffer:p_memory;
                         index, device:integer; control:rwctrl ):iorsltwd);
begin
  syshalt;
end;

Function bios_sys_init( device:integer;
                        procedure signal_event( event_num:integer ) ):iorsltwd);
begin
  bios_sys_init:=sysinit( signal_event, pollunits, disk_change );
end;

Function bios_sys_stat( var status:statrec; control:statctrl ):iorsltwd);
```

Appendix E

```
begin
  status[0]:=hi_addr.i;
  bios_sys_stat:=clkread( status[2], status[1] );
end;

Procedure bios_quiet;
begin
  quiet;
end;

Procedure bios_enable;
begin
  enable;
end;

Function bios_ser_read( device:integer; var ch:char ):iorsltwd);
begin
  bios_ser_read:=i_no_unit;
end;

Function bios_ser_write( device:integer; ch:char ):iorsltwd);
begin
  bios_ser_write:=i_no_unit;
end;

Function bios_ser_init( device:integer ):iorsltwd);
begin
  bios_ser_init:=i_no_unit;
end;

Function bios_ser_stat( device:integer; var status:statrec;
                        control:statctrl ):iorsltwd);
begin
  bios_ser_stat:=i_no_unit;
end;

End (BIOS).
```

APPENDIX F
ASCII TABLE

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	78	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	89	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

APPENDIX G
GLOSSARY

This specialized glossary is intended as an aid to readers who are unfamiliar with some of the terms used in this document. It isn't meant to be comprehensive.

ASSOCIATE TIME — That part of a program's lifetime in which the segments and their various references to each other are associated by the operating system. This occurs when the program is prepared for execution.

BLANK-FILLED — All 8-bit bytes within the specified region are filled with blanks (ASCII 32).

BLOCK — An area of memory (usually on a disk) with a fixed size of 512 contiguous 8-bit bytes (256 contiguous 16 bit-words).

BLOCK BOUNDARY — Byte zero of any block.

BYTE POINTER — A byte address (as opposed to a word address).

BYTE SEX — Some processors address 16-bit words with the most-significant-byte first, others with the least-significant-byte first. Byte sex refers to this difference in addressing; two machines with different addressing styles are said to have different (or opposite) byte sex.

COMPILATION UNIT — A program or portion of a program that can be compiled by itself—in other words, a program or a UNIT.

COMPILE TIME — That part of a program's lifetime in which it is being compiled (or assembled).

CONCURRENCY — The execution of two or more tasks or processes in parallel; that is, at the same time. Synonymous with multitasking.

DYNAMIC — Information which changes during program execution (or isn't known before run-time).

FILLER — A field in a data structure that is at present unused. If this area is described as "reserved for future use," then it usually should be zero-filled. This avoids confusion when future versions of the system make use of filler space.

INTERSEGMENT — The data (or program) in question occupies more than one segment, or contains pointers to another segment.

LINK TIME — That part of a program's lifetime in which it is being operated on by the linker.

MULTIPROGRAMMING — An environment that supports more than one user, where each user can perform multitasking. (The p-System doesn't support multiprogramming.)

MULTITASKING — The execution of two or more tasks in parallel; that is, at the same time. A task is a PROCESS from your point of view; from the system's point of view it might be a program. (The p-System does support multitasking.)

MULTIWORD — Some positive integral number of words.

NATIVE CODE — Assembled code for some physical (as opposed to ideal) processor. Also called machine code or (sometimes) hard code.

ONE'S COMPLEMENT — All bits in the designated field are flipped.

P-CODE — Assembled code for an ideal processor. P-code stands for "pseudo-code." The p-System PME implements a "pseudo machine emulator."

POSTPROCESSOR — A program which is executed after the completion of some other program, and uses as input the output of that previous program. A postprocessor that creates output which can be used by still another program is often called a "filter."

PRINCIPAL SEGMENT — A segment that has a segment reference list; for example, a segment with a SEG_TYPE of PROG_SEG or UNIT_SEG. Corresponds to the outer segment of any compilation unit. UNITs, FORTRAN programs, and the outermost block of a Pascal program are all principal segments.

RELOCATABLE — A portion of object code that can be moved to different locations in memory without changing its meaning. P-code is relocatable. Native code may or may not be.

RUN-TIME — That part of a program's lifetime in which it is being executed (or "run").

SELF-MODIFYING — Code which overwrites or modifies itself during execution, thus changing its meaning. This isn't recommended!

SEG-RELATIVE — The address of an object is specified as an offset from the beginning of the code segment in which it resides.

STATIC — Information which doesn't change throughout program execution (it is known before run-time).

SUBSIDIARY SEGMENT — A segment that has no segment reference list; for example, a segment with a SEG_TYPE of PROC_SEG or SEPRT_SEG. Corresponds to the object code of any segment whose source text is not separately compilable. Pascal segment procedures and segments produced by the UCSD adaptable assembler are subsidiary segments.

TOS — Short for "top of stack." Also represents the object that is on the top of the p-machine stack (which is the object that was most recently pushed).

UPWARD COMPATIBILITY — Code that runs on current versions of a system will run on future versions of that system. A more limited and more easily obtained version of upward compatibility requires source code to be recompiled on new versions, but ensures that it will run when recompiled.

WORD — Sixteen bits aligned on an even byte-address boundary. The byte which is most significant is determined by the byte sex of the machine for which it was generated.

WORD POINTER — A word address (as opposed to a byte address). The address of a word must be even.

ZERO-FILLED — A field of data that contains nothing but zeroes (all bits must be 0).

INDEX

- * -

*** 2-21

- A -

ALPHALOCK 4-29
Assembler-Generated Code Files 2-36

- B -

Basic Input/Output Subsystem 4-3
Basic I/O Subsystem 3-7
BIOS 3-7, 4-3
  Console 4-39
  Disk 4-53
  Entry Points 4-62
  Printer 4-50
  Remote 4-58
  Routine Parameters 4-62
  6500 Specifics 4-69
  68000 Specifics 4-73
  6809 Specifics 4-71
  8080 Specifics 4-67
  8086 Specifics 4-64
  Z80 Specifics 4-67
Blank Compression Code 4-26
Block I/O 5-25
BREAK 4-46
Byte Sex 2-6

- C -

code pool 5-13
Code Segments 2-3, 3-5
Completion Codes 4-13
Concurrency 5-20
concurrency 3-17
Constant Pool 2-8
CONTROL Parameters 4-11

- D -

DATAAREA 4-20
Data_Size 2-7
DEF 2-37
Device I/O 3-7
Device Numbers 4-10
Directories 5-24
DISPOSE 5-7
DLE 4-26

- E -

E_nvironment Record 3-13
Environment Records 3-13
EOF 4-28
E_REC 3-13
EVEC 3-13
Exit_IC 2-7

- F -

Fault Handling 5-18
Fault_Message 5-18
Fault_Sem 5-18

FIB 5-22
File Information Block 5-22
FLUSH 4-45

- H -

heap 3-4, 5-5
HeapInfo 5-10

- I -

interpreter 3-4
IORESULT 4-13
IPC 3-19

- L -

Linker Information 2-22
Logical Disk Structure 4-14

- M -

MARK 5-5
MemLink 5-9
MP 3-19

- N -

NEW 5-6
NOCRLF Bit 4-27

- O -

Operating System 5-3

- P -

p-machine 3-3
p-machine emulator 3-4
PED 6-5
PERMDISPOSE 5-7
PERMNEW 5-7
physical sector mode 4-15
P_MACHINE Intrinsic 3-22
PME 3-4
Program Environment Descriptor 6-5

- R -

Record I/O 5-25
REF 2-37
RELEASE 5-5
Relocation List 2-15
Routine Dictionaries 2-7
RSP 4-3, 4-25
RSP/IO 4-3

- S -

SBIOS 4-4
Screen I/O 5-25
Segment Information Blocks 3-8
SIB 3-8
SP 3-19
stack 3-4
START/STOP 4-44

- T -

task 3-17
Task Environments 3-17
Task_Info 5-20
Text_I/O 5-26
TIB 3-18
Type-Ahead 4-47

- U -

UNITBUSY 4-9, 4-22
UNITCLEAR 4-9, 4-13, 4-24
UNITNUMBER 4-19
UNITREAD 4-13, 4-19
UNITSTATUS 4-9, 4-13, 4-24
UNITWAIT 4-9, 4-23
UNITWRITE 4-9, 4-13, 4-19
User-Defined Devices 4-10

- V -

VARDISPOSE 5-7
VARNEW 5-6