UCSD p-System™ Internal Architecture

Software Library Manual

Texas Instruments Professional Computer

UCSD p-System Internal Architecture
TI Part No. 2232400-0001
Original Issue: 15 April 1983

Preface

This publication is a reference manual for the UCSD p-System™* on the Texas Instruments Professional Computer. It covers the internal details of the p-System. The p-machine architecture and instruction set are covered. Codefile 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 UCSD p-System (2232418-0001)

UCSD p-System Program Development (2232399-0001)

UCSD p-System Assembler (2232402-0001)

*UCSD Pascal™** (2232401-0001)

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.

* UCSD p-System and UCSD Pascal are trademarks of the Regents of the University of California.

Contents

Introduction

PURPOSE OF THIS GUIDE

This guide describes the internal design of the UCSD p-System as implemented on the Texas Instruments Professional Computer. 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 are covered.

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

Perhaps the best way to use this guide 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 PMEs for additional processors. For this group of programmers, the *UCSD p-System Internal Architecture* 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 would not suffice. We take it for granted that only a compelling need would lead you to take such steps.

All of these audiences, 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 on the Texas Instruments Professional Computer that is now called the UCSD 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 diskettes and VT-50 terminals. Students were expected to buy their own diskette, 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 practice of displaying a menu or prompt at every level of the p-System and organizing the levels in a tree structure—was intended to be easy to learn for the complete novice, yet not cumbersome for the experienced user. This proved very successful, and has been retained.

PDP-11/10 is a trademark of Digital Equipment Corporation.

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™[1] machines. Because of the emulative approach, it was a relatively straightforward matter to rewrite the p-machine emulator for the 8080 and Z80™[2], and subsequently, for many other processors. The 8086 PME now runs on the Texas Instruments Professional Computer.

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 is not a new observation that hardware costs continue to plummet, while software, being hand-made, continues to be very expensive. It is through modularity and portability that p-System addresses the problem as thoroughly as it does.

[1] LSI-11 is a trademark of Digital Equipment Corporation.

[2] Z80 is a trademark of Zilog, Incorporated.

1

The p-Machine

OVERVIEW

The p-machine is an idealized machine. The operating system, programs such as the filer, and compiled user programs all 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 the native 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 compared to native codes, it is 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 essential 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 some 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. On the Texas Instruments Professional Computer, the PME is written in 8086 assembly language.

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 memory-resident 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.

The Stack can be considered part of the p-machine. 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 also contains the Codepool (see Chapter 3).

The Heap is more fully described in Chapter 3, 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—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 contained SEGMENT routines or EXTERNAL native code 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 and with the Stack and the Heap. The principal constraint, as far as code segments are concerned, is that the calling and called segment must both be present in main memory for an intersegment call to succeed.

Segments in main memory are all stored contiguously in an area called the Codepool. The Codepool resides between the Stack and the Heap, and may be moved about to create more room.

Code segments are described in this chapter. Codepool handling is described in Chapter 3, 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 Basic I/O Subsystem (BIOS), 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 Simplified BIOS (SBIOS). 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 2, Low-Level I/O.

PROGRAM CODE

Code Segments

A code segment is a collection of routines, together with descriptive information. The code and information in a segment is contiguous, since the code segment is the unit of movement for code; code is loaded into memory a segment at a time.

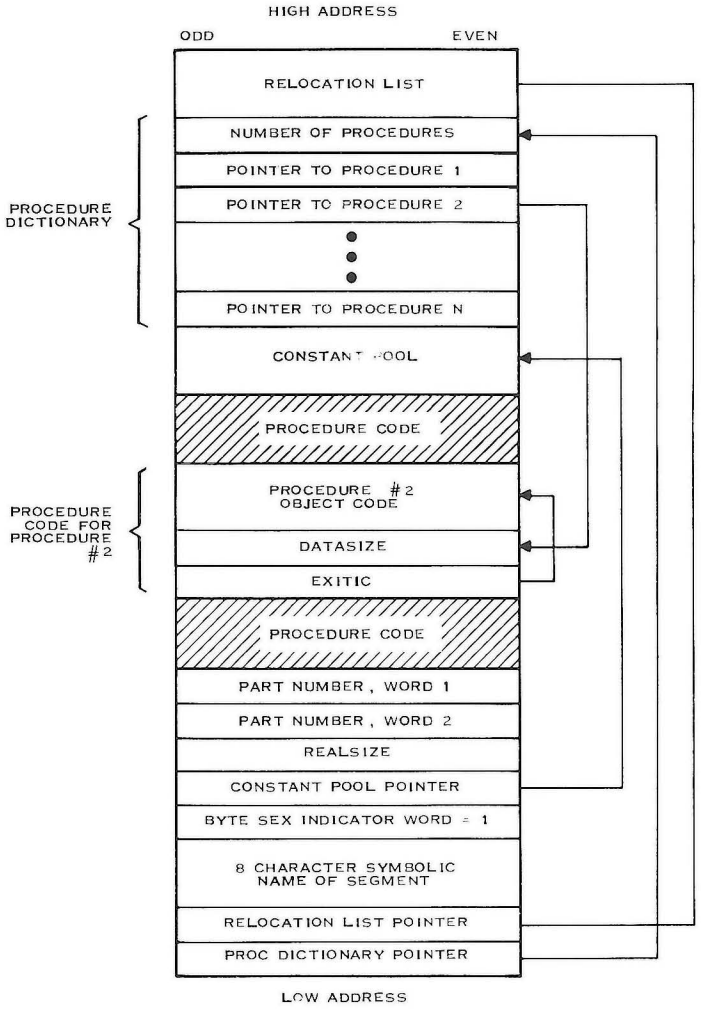
There are up to 255 routines within a segment, numbered 1 through 255.

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. 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 routine dictionary
- Pointer to the relocation list
- The 8-character name of the segment (4 words)
- Byte sex indicator word
- Pointer to the constant pool
- Real size word
- Space reserved for future use (2 words)

The figure following illustrates a code segment as it would be loaded into memory; the various substructures of a code segment are described.


HIGH ADDRESS
ODD
EVEN
RELOCATION LIST
NUMBER OF PROCEDURES
POINTER TO PROCEDURE 1
POINTER TO PROCEDURE 2
POINTER TO PROCEDURE N
PROCEDURE DICTIONARY
CONSTANT POOL
PROCEDURE CODE
PROCEDURE #2 OBJECT CODE
DATASIZE
EXITIC
PROCEDURE CODE FOR PROCEDURE #2
PROCEDURE CODE
PART NUMBER, WORD 1
PART NUMBER, WORD 2
REALSIZE
CONSTANT POOL POINTER
BYTE SEX INDICATOR WORD = 1
8 CHARACTER SYMBOLIC NAME OF SEGMENT
RELOCATION LIST POINTER
PROC DICTIONARY POINTER
LOW ADDRESS

2284131

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 XJP tables or constants accessed by LDC. 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 was run. Immediately following the segment's eight 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 a 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, routine dictionaries are 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 Dictionaries

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 seg-relative word pointer.

Routines within a segment are numbered 1 through 255. A routine's number is an index into the routine dictionary; the nth 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: DATASIZE and EXITIC, followed by the executable object code. The object code may be entirely p-code, entirely native code, or a mixture of the two.

DATASIZE is the number of words of local data space that must be allocated when the procedure is called. DATASIZE does not include parameters; the routine's parameters are assumed to already be on the stack. The first executable instruction starts at the byte or word immediately following the DATASIZE word. If the first executable instruction is native code, DATASIZE is one's-complemented.

If this first instruction is a p-code instruction, then EXITIC is a seg-relative byte pointer to the code that must be executed when the procedure is exited. If this first instruction is a native code instruction, then EXITIC 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.

The Constant Pool

Multiword 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.

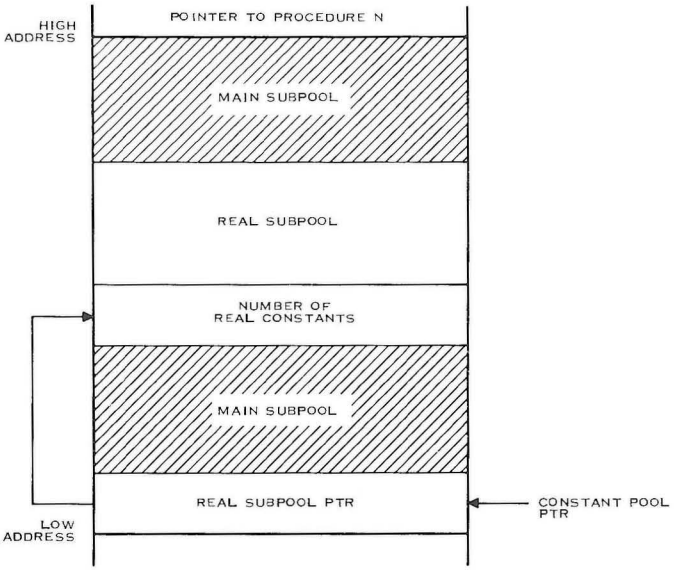
The location of the constant pool is contained in the constant pool pointer, a seg-relative word pointer that immediately follows the byte sex indicator word at the beginning of the segment. It points to the low address of the constant pool. If the constant pool pointer is equal to zero, the segment does not 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 must be 0. The first word of the real pool contains the number of real constants in the real pool.

The following figure illustrates a constant pool with an embedded real subpool.



2284132

Real constants are generated for either 32-bit or 64-bit floating point Binary Coded Decimal (BCD) data formats—real values (and operations upon them) can be transported across all processors with the same-sized representation of floating point numbers, but cannot be transported to machines with floating point formats of a different size.

NOTE

Two-word (32-bit) floating point arithmetic is *not* necessarily available on the Texas Instruments Professional Computer.

Only one size is likely to be available for a particular processor, since real constant handling is done by machine-dependent software (that is, within the PME). Within a single program, all compilation units *must* share the same size for real constants and variables.

The Pascal compiler is configured, when compiled, to default either to 32-bit or 64-bit reals. A directive is available to override the default:

```
{$R2} - sets realsize to 2 words (32 bits)
{$R4} - sets realsize to 4 words (64 bits)
```

This directive must occur before the first symbol in a compilation that is not a comment. The active realsize for a particular compilation is displayed after the compiler's version number at the beginning of the console output during a compilation and in a compiled listing.

The realsize at compilation time is also embedded in every code segment even though it may not reference any reals. The word REALSIZE 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 does not 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 (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 4 low-order digits. The first word contains a signed value between 1 and 9999, it represents the 4 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 4 and 6 words, depending upon the number of significant digits in the constant. The first 2 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.
Mantissa 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 4 digits (left-justified).
Mantissa word 3 contains a negative terminator.

9 .. 12 significant mantissa digits:
Mantissa word 2 contains 4 digits.
Mantissa word 3 contains 1 to 4 digits (left-justified).
Mantissa word 4 contains a negative terminator.

13 .. 16 significant mantissa digits:
Mantissa words 2-3 contain 4 digits.
Mantissa word 4 contains 1 to 4 digits.
Mantissa word 5 contains a negative terminator.

17 .. 20 significant mantissa digits:
Mantissa words 2-4 contain 4 digits.
Mantissa word 5 contains 1 to 4 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 the Real Convert utility to convert their real subpools into native machine format. 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 do not require updating.

The Relocation List

The last (high address) body of information in a memory-resident 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 seg-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 *only* needed by native code. Segments containing p-code exclusively 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:

```
LocTypes = (RelocEnd, {signals end of entire relocation list}
        SegRel, {relative to address of base of this segment}
        BaseRel, {relative to data segment given in DATASEGNUM}
        InterpRel,{relative to PME's interp-relative table}
        ProcRel); {relative to address of 1st instruction in proc}

ListHeader = PACKED RECORD
          ListSize: integer; {number of pointers in sublist}
          DataSegNum: 0..255; {local segment number for BaseRel}
          RelocType: LocTypes; {relocation type of sublist entries}
        END;
```

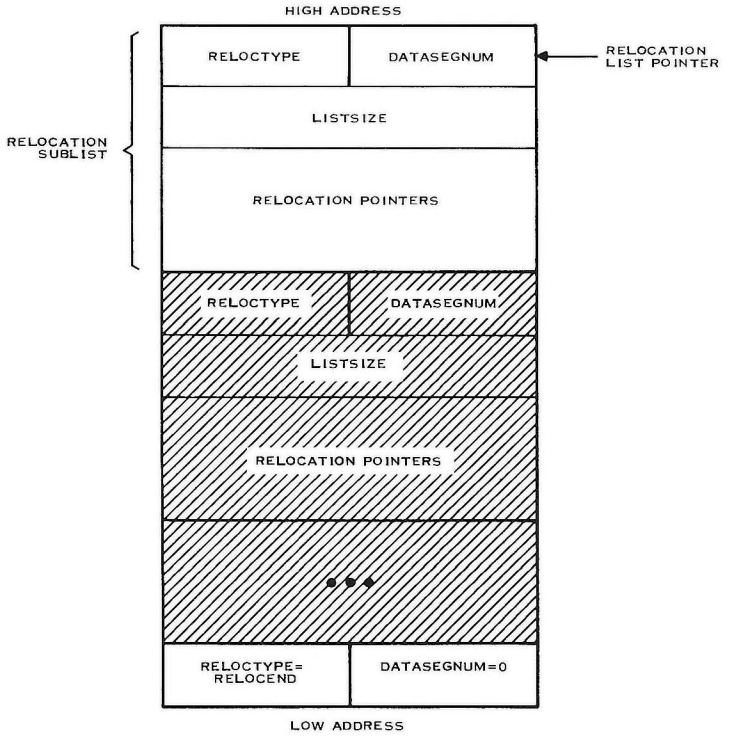
Each sublist contains a ListHeader and zero or more seg-relative byte pointers to the objects which must be relocated. The RelocType field in the ListHeader defines what kind of relocation will be applied to all objects designated by the sublist.

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

The DataSegNum field in the ListHeader is only used in sublists with a RelocType of BaseRel, and in all other cases should be zeroed. It specifies the local segment number of the data segment that all of the sublist's pointers are relative to. Since the assembler cannot 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 ListSize field in the ListHeader contains the number of pointers in the sublist.

The following figure illustrates a relocation list with multiple sublists.



2284133

The relocation list is intended to be used from high address down to low address. Each sublist in turn from high to low is processed until a sublist with a relocation type of RelocEnd is encountered. The DataSegNum and ListSize 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, 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 (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. (For more information about environment vectors, see the section, Code Segment Environments later in this 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, which are created by the compiler and appear in the object code, 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 does not occupy any space in memory during the program's execution.

The segment reference list associates the name of each compilation unit with the number by which that compilation unit is referenced. The compilation unit names do not change.

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

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

The Seg_Refs entry in the segment dictionary contains the number of words in the segment reference list. The Code_Leng field in the segment dictionary can be used as a seg-relative word pointer to the start of the segment reference list. The segment reference list consists of one or more SegRec's, starting directly above the relocation lists and continuing towards higher memory addresses. A SegRec consists of SegName, which contains the name of the segment, SegNum, 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 SegRec with a blank-filled SegName and SegNum of zero.

SegRec's with a SegName of *** are generated so 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 used units that contain a reference to ***, and uses this list to execute the initialization/termination sections of all used units before/after the invocation of the host program.

When the initialization/termination section of a unit (which is procedure 1) is compiled, a <CXG\ <***'s seg num>, 1> instruction is emitted between the initialization and termination parts. 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, itself. When the main program terminates, the calling chain is popped, and termination sections are executed in the reverse order.

Linker Information

Linker information (linker info) 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 eight-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 EOFMark. Linker information records are *always* 8 words long—unused records and unused fields are zero-filled.

If a code segment has linker information, the HasLinkerInfo 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 ahead in this chapter).

Two fields are common to all linker information records. The Name field contains an eight-character segment name. The LIType 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:

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

LITypes   = (EOFMark, GlobRef, PublRef, PrivRef, ConstRef,
             GlobDef, PublDef, ConstDef, ExtProc, ExtFunc,
             SepProc, SepFunc);

LIEntry   = RECORD
              Name: PACKED ARRAY [0..7] OF CHAR;
              CASE LIType: LITypes OF

                GlobRef, PublRef, ConstRef
                        : (Format: (Word, Byte, Big);
                           NRefs: integer);

                PrivRef: (Format: (Word, Byte, Big);
                          NRefs: integer;
                          NWords: integer);

                ExtProc, ExtFunc
                        : (SrcProc: integer;
                           NParams: integer);

                SepProc, SepFunc
                        : (SrcProc: integer;
                           NParams: integer;
                           KoolBit: Boolean);

                GlobDef: (HomeProc: integer;
                          ICOffSet: integer);

                PublDef: (BaseOffset: integer;
                          PubDataSeg: integer);

                ConstDef: (ConstVal: integer);

                EOFMark:
              END {CASE};

            PtrList: ARRAY[0..PtrRecNum] OF
                       ARRAY [0..7] OF integer

          END {LIEntry};
```

GlobRef, PublRef, ConstRef, and PrivRef are all linker information types generated by an assembler. They all consist of two fields that precede a list (PtrList) of seg-relative byte pointers into the associated segment. Format contains the size of the fields pointed to by the accompanying list. NRefs contains the number of pointers in the list. PtrList contains multiples of eight words; all unused words should be zero.

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

GlobRef 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 to by PtrList. This offset must be in the correct addressing mode; that is, bytes or words, depending on the processor being used.

PublRef 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 to by PtrList.

ConstRef 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 to by PtrList.

PrivRef is used to allocate space in the global data segment. Format should always be Word. NWords 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 to by PtrList.

ExtProc and ExtFunc are generated by a compiler to reference EXTERNAL routines. There is no PtrList. SrcProc is the number assigned to the routine. NParams is the number of words allocated for parameter passing.

SepProc and SepFunc are generated by an assembler for routine declarations. There is no PtrList. SrcProc is the number assigned to the routine. NParams is the number of words allocated for parameter passing. KoolBit is TRUE if the routine is relocatable, FALSE otherwise. Thus, .PROC and .FUNC generate SepProc or SepFunc records with KoolBit = FALSE, and .RELPROC and .RELFUNC generate SepProc or SepFunc records with KoolBit = TRUE.

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

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

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

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

The following example 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 cannot have linker information at all:

	Prog_Seg	Unit_Seg	Seprt_Seg
GlobRef			yes
PublRef			yes
PrivRef			yes
ConstRef			yes
ExtProc	yes	yes	
ExtFunc	yes	yes	
SepProc			yes
SepFunc			yes
GlobDef			yes
PublDef	yes	yes	
ConstDef	yes	yes	
EOFMark	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 6 different arrays. This information can be 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}
      Part_Num: PACKED ARRAY [1..8] of 0..15;
      Filler: ARRAY [0..4] OF integer; {reserved for future use}
      Copy_Note: string[77]; {copyright notice}
      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 does not 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; 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; 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, or 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 assembly routines is dynamically relocatable only if all of its assembly routines were originally specified as dynamically relocatable. Note that the use of these assembler directives is an assertion by the programmer that the routines they declare behave properly; the system does not enforce this, so caution must be used. If a routine is to be dynamically relocatable, it cannot store information into the segment body, be self-modifying, or store any pointers to the code segment in data variables, and then assume that things will behave correctly the next time it is called.

The Boolean Relocatable is unaffected by the presence or absence of relocation lists, and is not 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 1 or 2 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 is not known at code file generation time (as is the case with Seprt_Seg's), the field should be blank-filled.

If 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.

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 code file. 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 code file, 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 ProcRel 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 to by the seg-relative word pointer contained in the ExitIC 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 it is being bound to. 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 DataSize 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 cannot 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 DataSegNum field in the ListHeader record of all BaseRel relocation sublists should be zero-filled.

It is the linker's responsibility, when linking assembler-generated segments, to convert all ProcRel relocation sublists into SegRel relocation lists, to correctly set the DataSegNum field in the ListHeader of all BaseRel 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.

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 can 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.

NOTE

A code segment need not be in memory to be active; an active code segment may be on disk or in the Codepool, but its SIB will always be on the Heap.

The following fragment of Pascal code describes an SIB:

```
SIB = RECORD
        Seg_Pool: Pool_Ptr;
        Seg_Base: Mem_Ptr; {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_Base contains the current memory address of the code segment. If the code segment is not 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 a CXP to a routine within the segment. It is decremented whenever a RET from a routine within the segment returns to a routine outside the segment.

Time_Stamp contains a value based on the number of times a segment is used; it increases over time. It is incremented by six whenever a call is made to a routine outside the segment. It is also incremented by six whenever a routine within the segment returns to a routine outside the segment. Finally, it is incremented by six whenever a task switch suspends the segment that is currently executing.

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 and 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 TRUE. A zero indicates that the segment is Swappable, 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 Mem_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 (VIP) 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 is not 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 the Link_Count becomes zero, the SIB is removed from the Heap.

Environment Records (E_RECs)

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

For each segment, there is an Environment Record (E_Rec). This record designates an Environment Vector (E_Vec) 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 segment is allocated on the Heap when the program is called.

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, which is also placed on the Heap when the program is called.

Link_Count indicates the number of active compilation units that are currently USE'ing the segment. This only applies to the principal E_Rec of a compilation unit. Link_Count is maintained in the same way an 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 one. Thus, it may be indexed by local segment numbers. These must be one or greater. The Vec_Length field of the record may be considered to occupy the zero position of the map.

The operating system uses a recursive routine to construct the environments of a program's USEd units, and then its subsidiary segments and principal segment—the 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 USEd 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. Task is implemented by three data structures: the body, the Task Information Block (TIB), and the task stack. 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 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 is no different from the body of a procedure or function.

The amount of space allocated to the task stack depends on the STACKSIZE 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
- Several tasks waiting for semaphores

The tasks that are ready to run are organized into a queue. There is also a queue of waiting tasks for each semaphore, though it 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;
                  ProcNum: byte;
                  TIBIOResult: byte;
                  Hang_Ptr: Sem_Ptr;
                  M_Depend: integer;
              END {of Regs}
        MainTask: Boolean;
        Start_MSCW: MSCW_Ptr;
      END {of TIB}
```

SP is the p-machine Stack Pointer. SP_Low and SP_Upr are the stack pointers for this task.

MP designates the local activation records for this task.

IPC is the p-code Instruction Counter (a seg-relative byte pointer), and ProcNum 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 more urgent the priority.

Wait_Q is used when the task is waiting to run, or waiting on 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 is not 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 E_Rec. The task's SIB may be found through the E_Rec.

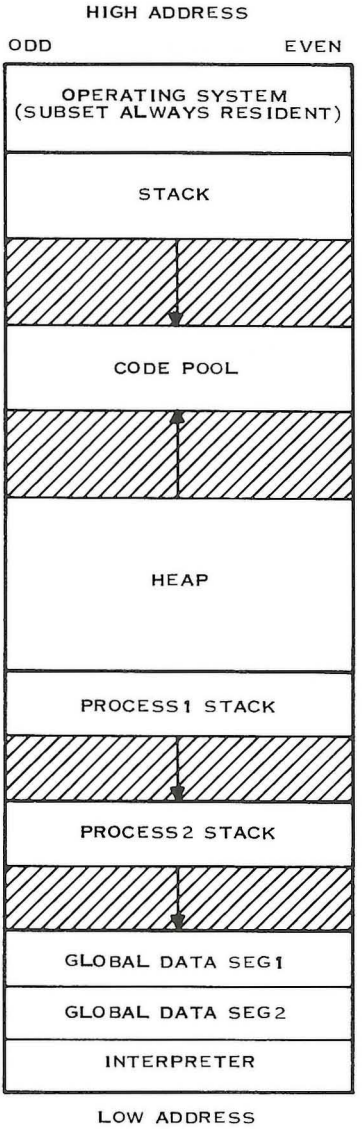
In the future, TIBIOResult will be used to save an IORESULT that is local to the task.

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

MainTask, if TRUE, indicates that this is the TIB of a root or parent task.

Start_MSCW points to the Mark Stack Control Word (MSCW) of the routine that START'ed this task.

Further information about tasks appears in Chapter 3, The Operating System. The following figure shows the layout of main memory while the system is running, including the location of task stacks as discussed in this section.


HIGH ADDRESS
ODD
EVEN
OPERATING SYSTEM
(SUBSET ALWAYS RESIDENT)
STACK
CODE POOL
HEAP
PROCESS1 STACK
PROCESS2 STACK
GLOBAL DATA SEG1
GLOBAL DATA SEG2
INTERPRETER
LOW ADDRESS

2284134

p-MACHINE INSTRUCTIONS

The Intrinsic p_MACHINE

A Pascal compilation unit may directly generate online p-code. This is done by calling the intrinsic procedure 'P_MACHINE'. Producing online 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>:

- 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.

- 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.

- 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
                 FirstField, SecondField: integer
               END;
     PRecords = Records;

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

... the following call to P_MACHINE ...

```
PMACHINE (^Vector[5]^.FirstField, (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-Code Instruction Set

Operands and Notation

Instruction Parameters — The parameters to a p-code instruction contain information about the location and size of that instruction's operands. They are generated at compile time and are, therefore, static. Each p-code uses some fixed combination of these parameters.

These are the five possible parameter formats; there are no others:

UB — Unsigned Byte

Represents a positive integer in the range 0 through 255. When converted to a 16-bit two's complement value, the most significant byte is zeroed.

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)).

DB — Don't Care Byte

Represents a positive integer in the range 0 through 127. It may thus be treated as either an SB or UB. Bit 7 is always 0.

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 should be 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.

W — Word

This is a two-byte parameter. It is a 16-bit two's complement value that represents an integer in the range −32768 through 32767. The word is always least-significant-byte-first.

Dynamic Operands — In the p-machine instruction descriptions that follow, stack-oriented dynamic operands of the p-code will be discussed. This section describes those operands.

Activation Record

See the following section.

Addr (address)

A 16-bit hardware word address. On byte-addressable processors, this is typically an even quantity.

Bool (Boolean)

A 16-bit quantity treated as a logical value.

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 even 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.

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.

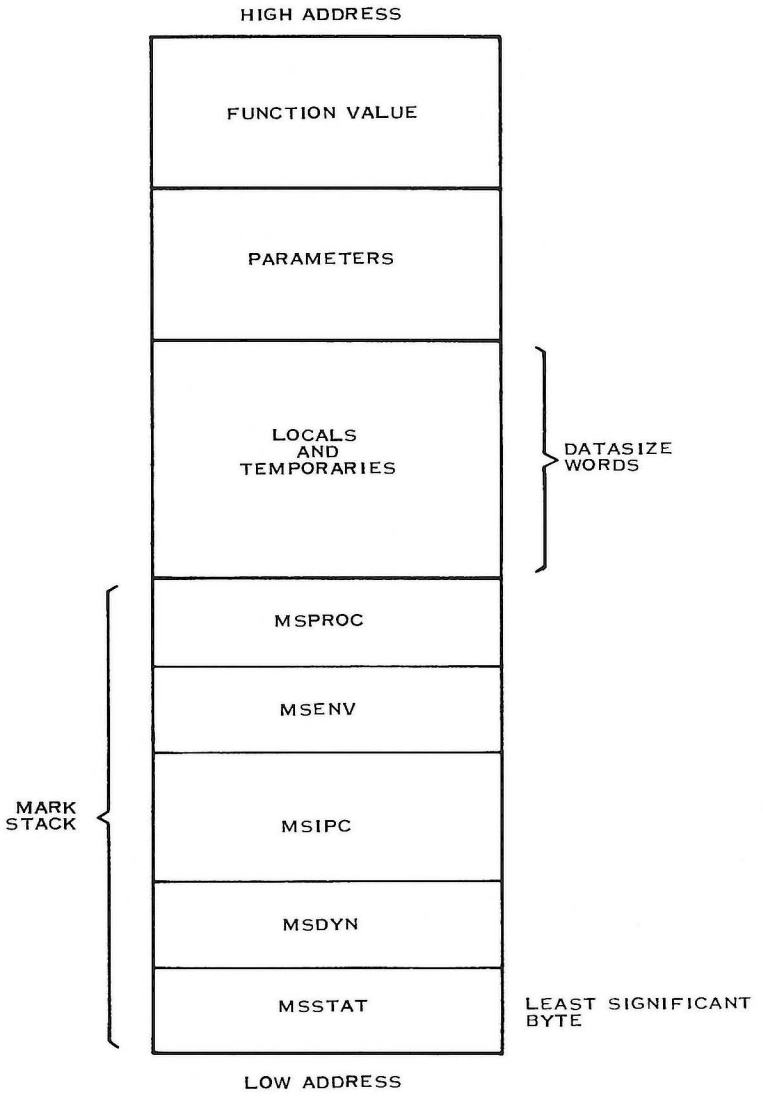
Word

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

Word-block

A group of zero or more words.

Activation Records — An activation record is created for each invocation of an active routine. The following figure illustrates an activation record.



2284135

The parts of an activation record are:

- Mark Stack.
- Five (full) words of housekeeping information:
  - MSSTAT — pointer to the activation record of the lexical parent.
  - MSDYN — pointer to the activation record of the caller.
  - MSIPC — seg-relative byte pointer to point of call in the caller.
  - MSENV — E_Rec pointer of the caller
  - MSPROC — procedure number of the caller
- Local and temporary variables. This area is DataSize words long.
- Parameters.
- This area (which may be empty) contains:
  - Addresses — for VAR parameters, and record and array value parameters.
  - Values — for other value parameters.
- Function value. This area is present only for functions, and is either one or two words (or four words, if reals are that size).

Conventions — The next section describes individual p-machine instructions, grouped by the nature of their operation.

On the left is the mnemonic for the instruction, followed by its value. All p-code instructions are represented by a single byte. This is followed by the format for the parameters, if any.

If the instruction has more than one parameter of the same format, then they are distinguished by an underscore followed by a number. Parameters of a given kind are numbered left-to-right, starting from one.

On the right is a verbal description of the instruction.

Below the opcode value is a notational description of the p-machine Stack before and after the p-code's execution. Only the expression-evaluation portion (the top words of the Stack) is shown.

On the left is a depiction of the Stack before the opcode is executed, followed by a colon (:), followed by a depiction of the Stack after the opcode is executed. Each depiction of the Stack is enclosed in angle brackets (< >). Within the brackets, the Stack grows from left to right. Individual operands are separated by commas, and vertical bars represent exclusive alternatives (one or the other value, but not both). Thus, the operand closest to the right bracket (>) is the top-of-stack (TOS). Brackets that do not enclose any operands represent an empty evaluation stack.

Constant One-Word Loads —

SLDC 0..31
<>:<word>

Short Load Word Constant. Push the op-code, with the high byte zero.

LDCN 152
<>:<NIL>

Load Constant NIL. Push NIL. The value may vary across processors.

LDCB 128 UB
<>:<word>

Load Constant Byte. Push UB, with high byte zero.

LDCI 129 W
<>:<word>

Load Constant Word. Push W.

LCO 130 B
<>:<offset>

Load Constant Offset. B is a word offset into the constant pool of the current segment. Convert B to a seg-relative word offset. If operating on a byte addressed machine, convert B to a byte offset. Push the offset on the Stack.

Local One-Word Accessing —

SLDL1	32
. . .	. . .
SLDL16	47
	<>:<word>

Short Load Local Word. SLDLx: fetch the word with offset x in the local activation record and push it.

LDL	135 B
	<>:<word>

Load Local Word. Fetch the word with offset B in the local activation record and push it.

SLLA1	96
. . .	. . .
SLLA8	103
	<>:<addr>

Short Load Local Address. Push the address of the indicated offset in the local activation record.

LLA	132 B
	<>:<addr>

Load Local Address. Calculate address of the word with offset B in the local activation record and push it.

SSTL1	104
. . .	. . .
SSTL8	111
	<word>:<>

Short Store Local Word. Store TOS in the indicated offset in the local activation record.

STL 64 B
<word>:< >

Store Local Word. Store TOS into word with offset B in the local activation record.

Global One-Word Accessing —

SLDO1 48
. . . . . .
SLDO16 63
< >:<word>

Short Load Global Word. SLDOx: fetch the word with offset x in the global data area of the current segment and push it.

LDO 133 B
< >:<word>

Load Global Word. Fetch the word with offset B in the global data area of the current segment and push it.

LAO 134 B
< >:<addr>

Load Global Address. Push the word address of the word with offset B in the global data area of the current segment.

SRO 165 B
<word>:< >

Store Global Word. Store TOS into the word with offset B in global data area of the current segment.

Intermediate One-Word Accessing —

SLOD1	173 B
. . .	. . .
SLOD2	174 B
	<>:<word>

Short Load Intermediate Word. Push the word at offset B in the lexical parent (LOD1) or grandparent (LOD2) activation record of the local activation record.

LOD	137 DB,B
	<>:<word>

Load Intermediate Word. DB indicates the number of static links to traverse to find the activation record to use. Push the word at offset B in that activation record.

LDA	136 DB, B
	<>:<addr>

Load Intermediate Address. DB indicates the activation record as for LOD. Push the address of offset B in that record.

STR	166 DB, B
	<word>:<>

Store intermediate word. Store TOS at offset B in the activation record indicated by DB.

Extended One-Word Accessing —

LDE	154 UB, B
	<>:<word>

Load Extended Word. Push the word at offset B in the global data area of local segment UB.

LAE 155 UB, B
<>:<addr>

Load extended address. Push the address of the word at offset B in the global data area of local segment UB.

STE 217 UB, B
<word>:<>

Store extended word. Store TOS at offset B in the global data area of local segment UB.

Indirect One-Word Accessing —

SIND0 120
. . . . . .
SIND7 127
<addr>:<word>

Short Index and Load Word. TOS is the address of a record. SINDx: replace it with word x of the record.

IND 230 B
<addr>:<word>

Index and Load Word. TOS is the address of a record. Replace it with the Bth word in the record.

STO 196
<addr,word>:<>

Store Indirect. Store TOS into the word pointed to by TOS-1.

Multiple-Word Accessing —

LDC 131 UB_1, B, UB_2
<>:<word-block>

Load Multiple Word Constant. B is a word offset into the constant pool of the current segment. Push the UB_2 words starting at that offset onto the evaluation stack. If UB_1, the mode, is 2, and the current segment is of opposite byte sex from the host, swap the bytes of each word as it is pushed.

LDM 208 UB
<addr>:<word-block>

Load Multiple Words. TOS is a pointer to the beginning of a block of UB words. Push the block onto the stack, preserving the order of words in the block.

STM 142 UB
<addr,word-block>:<>

Store Multiple Words. TOS is a block of UB words. Transfer the block from the stack to the destination block starting at the address TOS-1, and preserving the order of words in the block.

LDCRL 242 B
<>:<real>

Load Real Constant. Push the real constant designated by the constant pool index B in the current segment. The constant is guaranteed to be in the native byte sex of the host, so no byte flipping is necessary during the load.

LDRL 243
<addr>:<real>

Load Real. TOS is the address of a real variable. Replace the address by the value of the variable.

STRL 244
<addr,real>:< >

Store Real. TOS is the value of a real variable. TOS-1 is an address. Store TOS at the address in TOS-1.

String and Array Parameter Copying — To copy value parameters of type string or packed array of character into the activation record of a called routine, the calling routine generates a parameter descriptor. This descriptor is a 2-word record. The first (low address) 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 parameter. If the first word points to an E_Rec, the second word is an offset relative to the designated segment. The offset is generated by an LCO instruction.

The called routine uses a CAP or CSP instruction to copy the parameter into its activation record. CAP and CSP use the parameter descriptor to do this.

CAP 171 B
<addr,addr>:< >

Copy Array Parameter. TOS is the address of the parameter descriptor for a packed array of characters. Cause a segment fault if the parameter descriptor designates a non-resident segment. Otherwise, copy the source (which is B words big) into the destination address at TOS-1.

CSP 172 UB
<addr,addr>:< >

Copy String Parameter. TOS is the address of the parameter descriptor for a string. Cause a segment fault if the descriptor designates a non-resident segment. Otherwise, compare the dynamic length of the designated string against UB, the declared size (in bytes) of the destination formal parameter. Cause a string overflow fault if the length of the source is greater than the capacity of the destination. Otherwise copy for the length of the source into the destination whose address is TOS-1.

Byte Accessing —

LDB 167
<byte-ptr>:<word>

Load Byte. TOS is a byte pointer. Pop it and push a word with the byte it designated in the least significant bits and a most signifant byte of zero.

STB 200
<byte-ptr,word>:< >

Store Byte. Store byte TOS into the location specified by byte pointer TOS-1.

Packed Field Accessing —

LDP 201
<pack-ptr>:<word>

Load a Packed Field. Replace the packed field pointer TOS with the field it designates. Before being pushed on the Stack, the field is right-justified and zero-filled.

STP 202
<pack-ptr,word>:< >

Store into a Packed Field. TOS is the right-justified data, TOS-1 a packed field pointer. Store TOS into the field described by TOS-1.

Record and Array Accessing —

MOV 197 UB, B
<addr,addr>:< >

Move. Move B words from the source designated by TOS to the destination designated by TOS-1. TOS is either the address of a word block (if UB is zero) or the offset of a constant word block in the current segment. If UB is 2, and the current segment has opposite byte sex from the host, swap the bytes of each word as it is moved.

INC 231 B
<addr>:<addr>

Increment Field Pointer. The word pointer TOS is indexed by B words and the resultant pointer is pushed.

IXA 215 B
<addr,word>:<addr>

Index Array. TOS is an integer index, TOS-1 is the array base word pointer, and B is the size (in words) of an array element. Push a word pointer to the indexed element.

IXP 216 UB_1, UB_2
<addr,word>:<pack-ptr>

Index Packed Array. TOS is an integer index, TOS-1 is the array base word pointer. UB_1 is the number of elements per word, and UB_2 is the field width (in bits). Compute and push a packed field pointer.

Logical Operators —

LAND 161
<word,word>:<word>

Logical And. AND TOS into TOS-1.

LOR 160
<word,word>:<word>

Logical Or. OR TOS into TOS-1.

LNOT 229
<word>:<word>

Logical Not. Take one's complement of TOS.

BNOT 159
<Bool>:<Bool>

Boolean Not. Complement the low order bit and clear the remainder of TOS.

LEUSW 180
<word,word>:<Bool>

Less Than or Equal Unsigned. Push Boolean result of unsigned comparison TOS-1 < = TOS.

GEUSW 181
<word,word>:<Bool>

Greater Than or Equal Unsigned. Push Boolean result of unsigned comparison TOS-1 > = TOS.

Integer Arithmetic —

ABI 224
<int>:<int>

Absolute Value Integer. Take absolute value of integer TOS. Result is undefined if TOS is initially −32768.

NGI 225
<int>:<int>

Negate Integer. Take the two's complement of TOS.

INCI 237
<int>:<int>

Increment Integer. Add 1 to TOS.

DECI 238
<int>:<int>

Decrement Integer. Subtract 1 from TOS.

ADI 162
<int,int>:<int>

Add Integers. Add TOS into TOS-1.

SBI 163
<int,int>:<int>

Subtract Integers. Subtract TOS from TOS-1.

MPI 140
<int,int>:<int>

Multiply Integers. Multiply TOS by TOS-1. This instruction may cause overflow if result is larger than 16 bits.

DVI 141
<int,int>:<int>

Divide Integers. Divide TOS-1 by TOS and push quotient.

MODI 143
<int,int>:<int>

Modulo Integers. Divide TOS-1 by TOS and push the remainder.

CHK 203
<int,int,int>:<int>

Check Subrange Bounds. Ensure that TOS-1 < = TOS-2 < = TOS, leaving TOS-2 on the Stack. If conditions are not satisfied, cause a run-time error.

EQUI 176
<int,int>:<Bool>

Equal Integer. Push Boolean result of integer comparison TOS-1 = TOS.

NEQI 177
<int,int>:<Bool>

Not Equal Integer. Push Boolean result of integer comparison TOS-1 < > TOS.

LEQI 178
<int,int>:<Bool>

Less than or Equal Integer. Push Boolean result of integer comparison TOS-1 < = TOS.

GEQI 179
<int,int>:<Bool>

Greater than or Equal Integer. Push Boolean result of integer comparison TOS-1 > = TOS.

Real Arithmetic — All overflows and underflows cause a run-time error.

FLT 204
<int>:<real>

Float Top-of-Stack. Convert the integer TOS to a floating point number.

TNC 190
<real>:<int>

Truncate Real. Convert the real TOS to an integer by truncating.

RND 191
<real>:<int>

Round Real. Convert the real TOS to an integer by rounding.

ABR 227
<real>:<real>

Absolute Value of Real. Take the absolute value of the real TOS.

NGR 228
<real>:<real>

Negate Real. Negate the real TOS.

ADR 192
<real,real>:<real>

Add Reals. Add TOS into TOS-1.

SBR 193
<real,real>:<real>

Subtract Reals. Subtract TOS from TOS-1.

MPR 194
<real,real>:<real>

Multiply Reals. Multiply TOS by TOS-1.

DVR 195
<real,real>:<real>

Divide Reals. Divide TOS into TOS-1.

EQREAL 205
<real,real>:<Bool>

Equal Real. Push Boolean result of real comparison TOS-1 = TOS.

LEREAL 206
<real,real>:<Bool>

Less than or Equal Real. Push Boolean result of real comparison TOS-1 < = TOS.

GEREAL 207
<real,real>:<Bool>

Greater than or Equal Real. Push Boolean result of real comparison TOS-1 < = TOS.

Set Operations —

ADJ 199 UB
<set>:<word-block>

Adjust Set. Force the set TOS to occupy UB words, either by expansion (putting zeroes between TOS and TOS-1) or compression (chopping of high words of set), and discard its length word.

SRS 188
<int,int>:<set>

Build a Subrange Set. The integers TOS and TOS-1 must be in [0 through 4079]. If not, cause a run-time error, else push the set [TOS-1 through TOS]. If TOS-1 > TOS, push the empty set.

INN 218
<int,set>:<Bool>

Set Membership. Push Boolean result of TOS-1 IN TOS.

UNI 219
<set,set>:<set>

Set Union. Push the union of sets TOS and TOS-1. (TOS OR TOS-1)

INT 220

<set,set>:<set>

Set Intersection. Push the intersection of sets TOS and TOS-1. (TOS AND TOS-1)

DIF 221

<set,set>:<set>

Set Difference. Push the difference of sets TOS and TOS-1. (TOS-1 AND NOT TOS)

EQPWR 182

<set,set>:<Bool>

Equal Set. Push the Boolean result of set comparison TOS-1 = TOS.

LEPWR 183

<set,set>:<Bool>

Less than or Equal Set. Push true if TOS-1 is a subset of TOS, else push false.

GEPWR 184

<set,set>:<Bool>

Greater than or Equal Set. Push true if TOS is a superset of TOS or else push false.

Byte Array Comparisons —

EQBYT 185 UB__1, UB__2, B
<addr|offset,addr|offset>:
<Bool>

Equal Byte Array. TOS and TOS-1 are each a pointer to a byte array (if the corresponding UB is zero) or the offset of the constant byte array in the current segment. B is the size (in bytes) of that array. UB__1 and UB__2 are mode flags. They refer to TOS and TOS-1, respectively. If the byte sex of the segment is different from the host and the corresponding mode is 2, swap the bytes of each word of that operand before doing the comparison. Push the Boolean result of the byte array comparison TOS-1 = TOS.

LEBYT 186 UB__1, UB__2, B
<addr|offset,addr|offset>:
<Bool>

Less than or Equal Byte Array. TOS and TOS-1 each point to a byte array (if the corresponding UB is zero) or the offset of the constant byte array in the current segment. B is the size (in bytes) of that array. UB__1 and UB__2 are mode flags. They refer to TOS and TOS-1, respectively. If the byte sex of the segment is opposite that of the host and the corresponding mode is 2, swap the bytes of each word of that operand before doing the comparison. Push the Boolean result of the byte array comparison TOS-1 < = TOS.

GEBYT 187 UB_1, UB_2, B
<addr|offset,addr|offset>:
<Bool>

Greater than or Equal Byte Array. TOS and TOS-1 each point to a byte array (if the corresponding UB is zero) or the offset of a constant byte array in the current segment. B is the size (in bytes) of that array. UB_1 and UB_2 are mode flags. They refer to TOS and TOS-1, respectively. If the byte sex of the segment is opposite that of the host and the corresponding mode is 2, swap the bytes of each word of that operand before doing the comparison. Push the Boolean result of the byte array comparison TOS-1 < = TOS.

Jumps —

UJP 138 SB
<>:<>

Unconditional Jump. Jump by offset SB.

FJP 212 SB
<Bool>:<>

False Jump. Jump by offset SB if TOS is false.

TJP 241 SB
<Bool>:<>

True Jump. Jump by offset SB if TOS is true.

EFJ 210 SB
<int,int>:< >

Equal False Jump. Jump by offset SB if TOS < > TOS-1.

NFJ 211 SB
<int,int>:< >

Not Equal False Jump. Jump by offset SB if TOS = TOS-1.

JPL 139 W
< >:< >

Unconditional Long Jump. Jump W words from current location.

FJPL 213 W
<Bool>:< >

False Long Jump. Jump W words from current location if TOS is false.

XJP 214 B
<int>:< >

Case Jump. The first word, W1, with word offset B in the constant pool of the current segment is word-aligned and is the minimum index of the table. The next word up, W2, is the maximum index. The case table is the next (W2−W1)+1 words. If the byte sex of the segment is opposite to the host, any of these words must be byte-swapped before they are used.

If TOS, the actual index, is in the range W1 through W2, then jump W3 words from the current location, where W3 is the contents of the word pointed to by TOS. Otherwise, do nothing.

Routine Calls and Returns —

CPL 144 UB
<param>:<activation>

Call Local Procedure. Call procedure UB, which is an immediate child of the currently executing procedure and in the same segment. Static link of the new MSCW is set to old MP.

CPG 145 UB
<param>:<activation>

Call Global Procedure. Call procedure UB, which is at lex level 1 and in the same segment. The static link of the MSCW is set to BASE.

SCPI1 239 UB
. . . . . .
SCPI2 240 UB
<param>:<activation>

Short Call Intermediate Procedure. Set the static chain to point to the lexical parent (CPI1) or grandparent (CPI2) of the calling environment. Call procedure UB.

CPI 146 DB, UB
<param>:<activation>

Call Intermediate Procedure. Call procedure UB, which is at lex level DB less than the currently executing procedure and in the same segment. Use that activation record's static link as the static link of the new MSCW.

CXL 147 UB_1, UB_2
<param>:<activation>

Call Local External Procedure. Call procedure UB_2, which is an immediate child of the currently executing procedure and in the segment UB_1.

SCXG1 112 UB
. . . . . .
SCXG8 119 UB
<param>:<activation>

Short Call External Global Procedure. The segment number is indicated by the opcode (1-8) and UB is the procedure number.

CXG 148 UB_1, UB_2
<param>:<activation>

Call Global External Procedure. Call procedure UB_2 which is at lex level 1 and in the segment UB_1.

CXI 149 UB_1, DB, UB_2
<param>:<activation>

Call Intermediate External Procedure. Call procedure UB_2 which is at lex level DB less than the currently executing procedure, and in the segment UB_1.

CPF 151
<param,proc-ptr>:<activation>

Call Formal Procedure. TOS contains a procedure number. TOS-1 contains an E_Rec pointer. TOS-2 contains a static link. Call the indicated procedure.

RPU 150 B
<activation>:<func>

Return from User Procedure. Restore state of calling procedure from MSCW and discard. Pop MSCW from Stack. Cut back an additional B words from Stack, leaving function value, if appropriate.

LSL 153 DB
<>:<addr>

Load Static Link onto Stack. DB indicates the number of static links to traverse. Push the indicated static link.

BPT 158
<>:<activation>

Breakpoint. Unconditionally call execution error procedure.

Concurrency Support —

SIGNAL 222
<addr>:<>

Signal. TOS is a semaphore address. Signal this semaphore.

WAIT 223
<addr>:<>

Wait. TOS is a semaphore address. Wait on this semaphore.

String Instructions —

EQSTR 232 UB__1, UB__2
<addr|offset,addr|offset>:
<Bool>

Equal String. TOS and TOS-1 each point to a string variable (if the corresponding UB is zero) or the offset of a constant string in the current segment. UB__1 and UB__2 refer to TOS and TOS-1, respectively. Push the Boolean result of the string comparison TOS-1 = TOS.

LESTR 233 UB__1, UB__2
<addr|offset,addr|offset>:
<Bool>

Less or Equal String. TOS and TOS-1 each point to a string variable (if the corresponding UB is zero) or the offset of a constant string in the current segment. UB__1 and UB__2 refer to TOS and TOS-1, respectively. Push the Boolean result of the string comparison TOS-1 < = TOS.

GESTR 234 UB__1, UB__2
<addr|offset,addr|offset>:
<Bool>

Greater or Equal String. TOS and TOS-1 each point to a string variable (if the corresponding UB is zero) or the offset of a constant string in the current segment. UB__1 and UB__2 refer to TOS and TOS-1, respectively. Push the Boolean result of the string comparison TOS-1 > = TOS.

ASTR 235 UB__1, UB__2
<addr,addr|offset>:< >

Assign String. TOS-1 is the address of the destination string variable. UB__2 is the declared size of that string. TOS represents the source for the assignment. It is either the address of a string variable (if the mode, UB__1, or the offset of a string constant in the current segment). Cause a string overflow fault if the dynamic size of the source string is greater than the declared size of the destination. Otherwise, copy the source into the destination.

CSTR 236
<addr,int>:< >

Check String Index. TOS-1 is the address of a string variable. TOS is an index into that variable. Check that the index is between 1 and the current dynamic length of the variable. If not, cause a range-check execution error.

Miscellaneous Instructions —

LPR 157
<int>:<word>

Load Processor Register. TOS is a register number. Push the contents of the register indicated in this fashion: (for SPR, also):

a. Register number is positive; it is a word index into the TIB.

b. Register number is negative:

−1 Indicates the pointer to the TIB of the currently running task

−2 Indicates the current E__Vec__P

−3 Indicates the pointer to the TIB at the head of the ready queue

SPR 209
<int,word>:< >

Store Processor Register. TOS-1 is a register number (defined as for LPR). Store TOS in indicated register.

DUP1 226
<word>:<word,word>

Duplicate One Word. Duplicate one word on TOS.

DUPR 198
<word-block>:<word-block>

Duplicate Real. Duplicate the real on TOS.

SWAP 189
<word,word>:<word,word>

Swap. Swap TOS with TOS-1.

NOP 156
<>:<>

No Operation. Continue execution.

NAT 168
<>:<>

Native Code. Transfer control to native code that begins directly after this instruction. Details are machine-dependent.

NAT- INFO 169 B
<>:<>

Native Code Information. Ignore the next B bytes in the p-code stream. This information is used in the generation of native code. Treat instruction as a long form of NOP.

RESERVE1 250
. . . . . .
RESERVE6 255

These codes are reserved for use by the compiler to identify embedded compiler directives. They must not be explicitly generated by programs.

2

Low Level I/O

THE I/O SUBSYSTEM

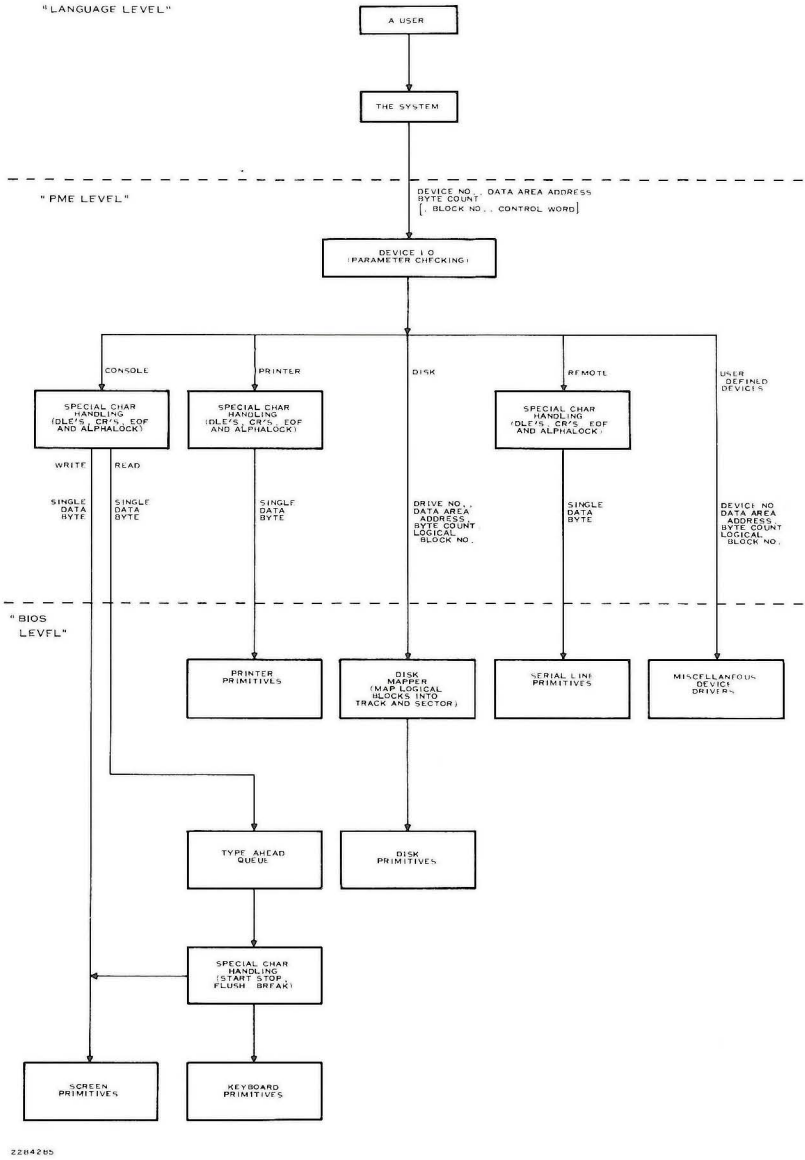
Besides emulating the p-machine, each PME must contain some native code to perform certain time-critical operations, and deal with hardware dependencies such as I/O devices. The body of code that is *not* devoted to emulating p-code is called the Run-time Support Package (RSP). The portion of the RSP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

The user's I/O calls (READLN, WRITELN) are mapped by the compiler and operating system into calls to the RSP (UNITREAD, UNITWRITE). The RSP/IO 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 does not 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 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.


"LANGUAGE LEVEL"
A USER
THE SYSTEM
"PME LEVEL"
DEVICE NO., DATA AREA ADDRESS
BYTE COUNT
[, BLOCK NO., CONTROL WORD]
DEVICE I/O
(PARAMETER CHECKING)
CONSOLE
PRINTER
DISK
REMOTE
USER DEFINED DEVICES
SPECIAL CHAR HANDLING (DLE'S, CR'S, EOF AND ALPHALOCK)
SPECIAL CHAR HANDLING (DLE'S, CR'S, EOF AND ALPHALOCK)
SPECIAL CHAR HANDLING (DLE'S, CR'S, EOF AND ALPHALOCK)
WRITE
READ
SINGLE DATA BYTE
SINGLE DATA BYTE
SINGLE DATA BYTE
DRIVE NO., DATA AREA ADDRESS, BYTE COUNT, LOGICAL BLOCK NO.
SINGLE DATA BYTE
DEVICE NO, DATA AREA ADDRESS, BYTE COUNT, LOGICAL BLOCK NO.
"BIOS LEVEL"
PRINTER PRIMITIVES
DISK MAPPER (MAP LOGICAL BLOCKS INTO TRACK AND SECTOR)
SERIAL LINE PRIMITIVES
MISCELLANEOUS DEVICE DRIVERS
TYPE AHEAD QUEUE
DISK PRIMITIVES
SPECIAL CHAR HANDLING (START, STOP, FLUSH, BREAK)
SCREEN PRIMITIVES
KEYBOARD PRIMITIVES
2284285

When the p-System was adapted to the 8080 and Z80, the widespread availability of CP/M™ was used—p-System I/O called CP/M BIOS routines. In this way, any hardware environment that CP/M already supported could then host the p-System.

As adaptations for additional processors (the 9900 and the 6502) were begun, 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 the BIOS definition did not address the problem of standardizing bootstrap mechanisms, and that 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 Simplified BIOS (SBIOS) is a very simple hardware interface. 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 DMA support in the disk interface.

CP/M is a trademark of Digital Research Incorporated.

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

NOTE

> The p-System is not sold by Texas Instruments as an adaptable system on the Texas Instruments Professional Computer since the SBIOS for the Professional Computer is already written.

DEVICE I/O ROUTINES

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 do not concern us here. The Device I/O Routines are not written in Pascal, but in fact are the native code procedures that comprise the RSP/IO. The way that these procedures are called is described next.

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 in fact 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

When you make direct calls to Device I/O Routines, they look like any other intrinsic routines. 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

As described elsewhere, each device is referred to in the system by a given number. The formal parameter UNITNUMBER in the preceding declarations determines which physical unit 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 unit. The following table is a list of the predefined unit numbers associated with each physical unit. 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	<Reserved for future expansion>

User-Defined Devices — The system reserves all device numbers above 127 for user-defined devices. They have no preassigned names, yet can be accessed through the UNIT intrinsics just as 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 the following two figures.

	MSB					LSB
	15-13 USER DEFINED	12-4	3	2	1	0
		(Reserved)	NOCRLF	NOSPEC	PHYSSECT	ASYNC
Value			8	4	2	1

Bit 0 ASYNC	Set (1) implies asynchronous I/O request. Reset (0) implies synchronous I/O request. (This bit should always be reset.)
Bit 1 SECT	Set implies "Physical Sector Mode" for disk I/O. Reset implies "Logical Block Mode" for disk I/O.
Bit 2 NOSPE	Set implies "no special character handling." Reset implies "special character handling."
Bit 3 OCRLF	Set implies no LFs are to be appended CRs during nondisk I/O. Reset implies LFs are to be appended to CRs during nondisk I/O.
Bits 4-1	Reserved for future expansion.
Bits 13-1	User-defined functions.

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

	MSB				LSB
	15-13 USER DEFINED	12-1	0		
		(Reserved)	IODIR		
Value			1		

Bit 0 IODIR	Set (1) implies the status of the *input* channel is to be returned. Reset (0) 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. The SYStem COMmunication area (SYSCOM) 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 the following figure.

Code	Meaning
0	No error
1	Bad block, CRC error (parity)
2	Bad device number
3	Illegal I/O request
4	Data-com time out
5	Volume is no longer online
6	File is no longer in directory
7	Illegal file name
8	No room; insufficient space on disk
9	No such volume online
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
19-127	Reserved for future expansion

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

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 Physical Sector Number (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 are not 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 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

This section now 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 has already been discussed.

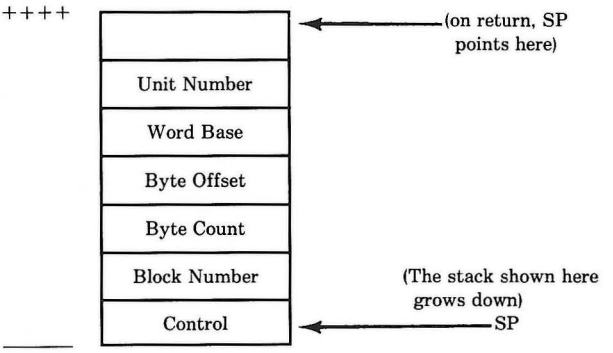
DATAAREA is your buffer to or from which the data will be transferred. Describing it as a VAR parameter signifies that UNITREAD and UNITWRITE are passed a pointer to the start of the data area. This pointer is actually represented as an address couple, consisting of a word base and a byte offset. On 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 indices, 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.

BYTESTOTRANSFER, the third item in the parameter list, 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 them both the default value zero. LOGICALBLOCK is only relevant for disk reads or writes; as discussed in the section, Logical Disk Structure, it specifies the Pascal logical block to be accessed. The CONTROL word has been discussed in the section, Control Parameters.

Parameter Stack Format

UNITREAD and UNITWRITE receive their parameters on the evaluation stack in the order shown in the following figure. Each box represents a 16-bit quantity.

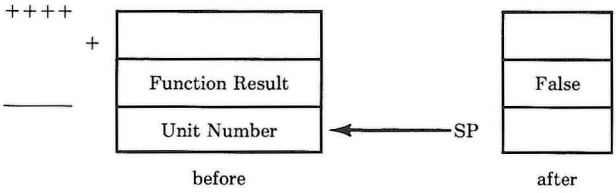


Like ordinary Pascal procedures, these RSP routines pop their parameters from the Stack when they are finished.

UNITBUSY

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

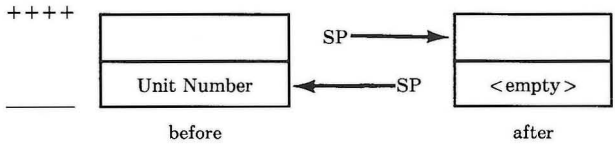
The UNITBUSY function has meaning only in an asynchronous environment and thus will always return FALSE (0) for this synchronous specification. The use of the stack is illustrated in the following figure.



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 an 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 the following figure.



UNITCLEAR

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

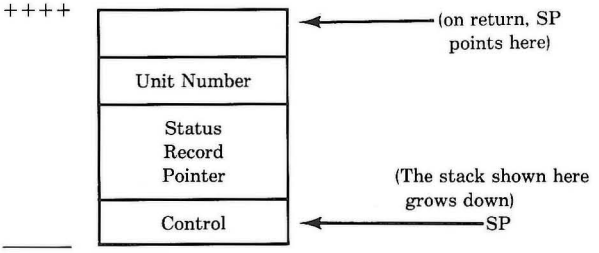
The purpose of UNITCLEAR is to restore the specified device to its initial state. At the RSP level, this would mean clearing any state flags pertaining to the specified device. The initial state for each device at the BIOS level is defined in the following section, BIOS Responsibilities. The stack format is identical to that of UNITWAIT (see the previous figure).

UNITSTATUS

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

The purpose of UNITSTATUS is to acquire 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 a CONTROL word is stored and also 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.

UNITSTATUS receives its parameters on the evaluation stack in the order shown in the following figure. Each box represents a 16-bit quantity:



RSP Responsibilities

This section will detail 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 shall be described here.

Special Character Output Handling

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

Blank Compression Codes (DLEs) — 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, due to 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 are not normally sent to the device (see the paragraph, NOSPEC Bit in Control Parameter, that follows).

Carriage Return (CR) — Line Feed (LF) — 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 CR's is turned off; that is, they are sent out like other characters and an LF is not automatically appended.

There are several characters which should receive special treatment when received from the console, the printer, or 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.

End-of-File (EOF) Character — The End-of-File (EOF) character, when received from the console, printer or remote devices, signals that the *end of file* has been reached on that particular unit. Rather than being a fixed ASCII code, this 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 can be found in the following section, Character Codes. 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 the buffer, starting at the current position, is packed with nulls (decimal 0). For UNIT 2 (SYSTERM:), the printer and the remote devices, the EOF character is put into the 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 following paragraph, NOSPEC Bit).

BIOS Functions — 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 (See the following paragraph, Console Input Options).

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 previously described are turned off. These characters should then be transferred as any other character. The BIOS functions are not affected.

Translation for Subsidiary Volumes

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

The syscom area contains a pointer to the unit-table which contains a record for each p-System unit. Each record for storage devices contains a block offset and physical disk unit number. The RSP must look up calls to subsidiary volumes and give the physical 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 describes the RSP handling of 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 previously, the Basic Input/Output Subsystem 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 sufficient detail for an experienced programmer to write the code for a given processor and set of peripherals.

The general scheme 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 the following section, 8086-Specific BIOS Calls.

Design Goals

The speed of the BIOS code 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 is one less available to the programmer. For these reasons, the major design goals should 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, at the beginning of Chapter 2. Most of the standard completion codes are not 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 online
15	Ring Buffer Overflow
16	Write protect; wrttempt 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 online) 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. For a summary of the BIOS calling requirements, see the following.

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 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 require five parameters:

1. A starting logical block number as previously described.

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 require five parameters:

1. A starting logical block number as previously described.

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 up to the device handler to decode 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 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 is that the control codes for terminals vary widely and users change terminals fairly often and so it was necessary to be able to change a terminal without creating a new BIOS. The basic issue is one of mapping logical control symbols into the 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 online. The details of how this is done are described elsewhere.

Since many devices can make use of eight-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 will be 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. A typewriter device may also be used for the console.

Console Output Requirements — As previously noted, we depend on the action of certain ASCII control codes. These are the minimum requirements for a console device:

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

LF <line feed> (hexadecimal 0A): Move 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 should sound. If one is not available, the terminal should do nothing. The delay time required while doing nothing is immaterial.

SP <space> (hexadecimal 20): Write 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 did not. As above, we would prefer that the cursor remain where it was and that the screen not scroll.

NUL <null> (hexadecimal 00): Delay for the time required to write one character. The state of the console should not change.

Any Printable Character: Same as the discussion for SP, except, of course, write the character.

NOTE

The effect of sending nonprintable characters other than those previously described is not defined, since it is known to vary from terminal to terminal.

Console Output Options — The following set of cursor and screen functions should be provided if possible. However, they are optional in the sense that almost all major functions of the system will still be available even if they are not provided. The control characters or sequences of characters which provide these functions are left unspecified (these 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: Move 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 is not feasible, the cursor and screen should just remain as they were.

Nondestructive Forward and Backward Space: Move the cursor in the direction indicated without changing the contents of the screen, that is, move it nondestructively. 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: Move the cursor to the upper left-hand corner of the screen without changing the contents of the screen.

Cursor X,Y Positioning: Move 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: Erase 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: Erase 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 in the following paragraphs. 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 is scrolling across the screen at 9600 baud, or a printer must be brought online before the program starts sending it characters. 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, does not display them) until: (1) FLUSH is typed 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 are tired of looking at it. Type FLUSH and 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 are not necessarily zero.

Type-Ahead: When nonspecial characters not described in previous sections 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 System Communication (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, expressed as positive byte offsets, are:

	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 the preceding section, Control Parameters, 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 conceived as being 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 does not 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 and the BIOS must recognize the end of a line. EOLN is signaled 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 (carriage return and line feed) is the only way your printer can print a line, then do so at an LF—do not 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 should not 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 does not 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 zero.

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 recieve a character; a nonzero value is interpreted as meaning the printer is not ready to recieve 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.

Mapping Blocks on Physical Sectors — The disk device may be any type of disk drive, diskette, 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 the operating system, performs the 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.

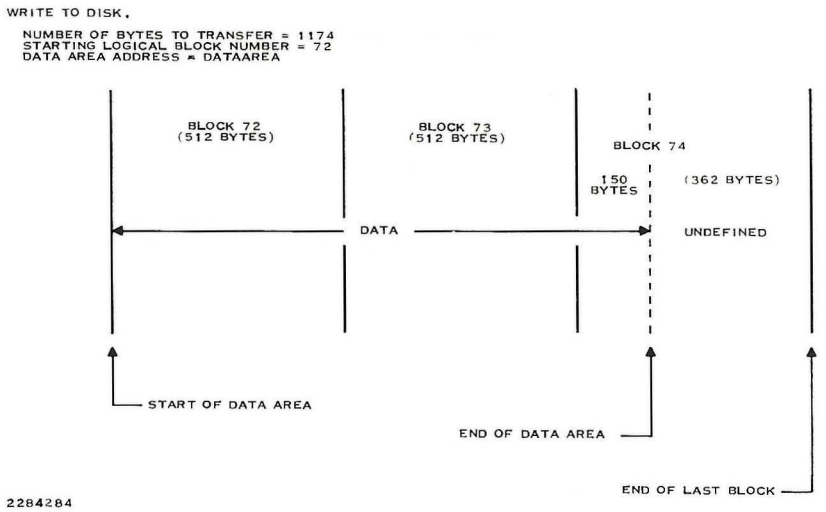
The primary bootstrap area must not overlap disk data structures maintained by the system, chiefly the directory and the bootstrap itself.

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 are not 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.

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 preceding section, Physical Sector Addressing Mode.

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 is not equal to zero (for example, a block that is partially written to), the remaining contents of the last block are undefined. This makes it possible to fill up a whole sector, if desired, with whatever garbage remains in the buffer. The following figure illustrates this situation. The language level is responsible for keeping track, in logical block numbers and byte counts, of where the good data is.



Disk Input Requirements — On input from a disk device, it is *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 preceding section, Console Status. 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 eight bits of the data byte should be transmitted. The remote BIOS driver receives one byte at a time.

Remote Input Requirements — Input from a remote device should be buffered, if possible, by the scheme suggested in the preceding section, Type-Ahead. 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. If no capability for checking is available, it should return 0.

User-Defined Devices

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

Special BIOS Calls

These functions are provided by the BIOS to make configuration-specific functions accessible to the PME. Although these functions are not related to Input/Output, 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 a 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 is not 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 is not 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 8086-specific protocols for the Texas Instruments Professional Computer 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

Entry Point	Parameters
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)
SERREAD	device number
	single data byte
SERWRITE	device number
	single data byte
SERCTRL	device number
SERSTAT	device number
	STATREC pointer
	CONTROL word

8086-SPECIFIC BIOS CALLS

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 the beginning of the PME.

Parameters: When parameters are not being passed in a specified register, they are pushed on the Stack. Offsets from the address pointed to by SP (described as (SP)) are given, recognizing that the Stack grows down and that SP normally points to the last word pushed on the Stack.

Completion Code: Return in register AH.

Calling Sequence: The RSP will use a 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, and DI with the exception of QUIET, ENABLE, SYSTEMSTAT which may only use AX, BX, CX, and DI. Registers CS, DS, SS, and 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

Entry Point	Offset (hex)	Parameters
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

Entry Point	Offset (hex)	Parameters
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

3

The Operating System

ORGANIZATION

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 user 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

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. The area between the Heap and the code pool 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. These either allocate or de-allocate Heap space in a particular way. The rest of this section is an introduction to these routines.

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 is allocated for this particular variant, 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.

CAUTION

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 de-allocated by PERMDISPOSE(P). Even a RELEASE cannot remove it. These routines are meant for system use, and are not your routines.

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.

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 does not 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.

The Heap maintenance variables have the following structure:

```
VAR
  HeapInfo: RECORD
              Lock: semaphore;
              TopMark,
              HeapTop: MemPtr;
            END;
  PoolBase: MemPtr;
  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 fault handler uses HeapTop to determine how close the code pool can be moved towards the Heap. PoolBase points to the base of the code pool. 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; 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 DISPOSEd 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 MARKs 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_Vec'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 the 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 pointers within the SIB, and the following global values:

PoolHead: SIB_Pt	Points to the SIB of the segment at the base of the code pool (next to the Heap).
PermSIB: SIB_Ptr	Points to the SIB of the segment that is always resident in the code pool (currently, GOTOXY).
PoolBase: Mem_Ptr	Points to the memory location at the base of the code pool.
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.
HeapTop: Mem_Ptr	Points to the top of the Heap.

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 Pool_Base, which points to the low end of the code pool, and SP_Low, which points to one word above the top of the code pool. The code pool operators may move it all the way to HeapTop on the Heap side, or up to SP minus a 40-word margin on the Stack side.

The code pool may be modified by any of the following circumstances:

- 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.

- 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.

- A Heap fault or Stack fault is detected, and the code pool cannot 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.

- A Heap or Stack fault is detected, and even after swapping out all of the swappable segments, not enough space is available: a STACK OVERFLOW is reported, and the system is reinitialized.

- 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, 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, segments are swapped out to make room, and the new segment is then read in. If this last effort also fails, a STACK OVERFLOW is reported, and 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 START'ed at bootstrap time. Most of the time it is idle, WAIT'ing on a semaphore. When the semaphore is SIGNAL'ed, the Faulthandler is activated and performs its memory management functions.

Faults can be SIGNAL'ed 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 SIGNAL's 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 WAIT's: this causes a task switch back to the previously running process. The instruction that caused the fault is re-executed.

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 is not 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 is not 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 de-activation 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 does not 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 on the termination of any subsidiary processes. N_Tasks is the number of subsidiary tasks that have been START'ed.

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, SIGNAL's Task_Info.Task_Done, and then initializes and waits on 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 File Information Block (FIB). When a user declares 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 zero for unentered files, one 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.

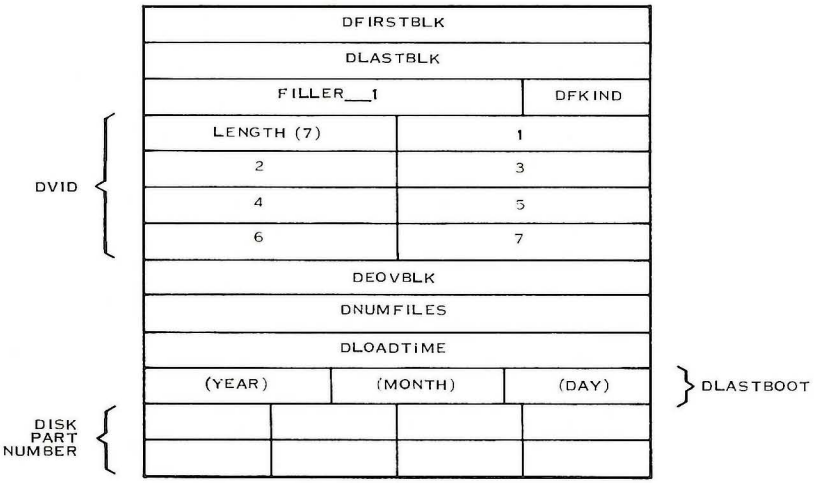
If FIsOpen is TRUE, then several other fields become relevant. FIsBlkd is TRUE if the file resides on a storage device. FDev is the number of that device, and FVolID the name of the volume. FReptCnt contains a count of the number of times the window value is valid before another GET is needed. FNxtBlk 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.

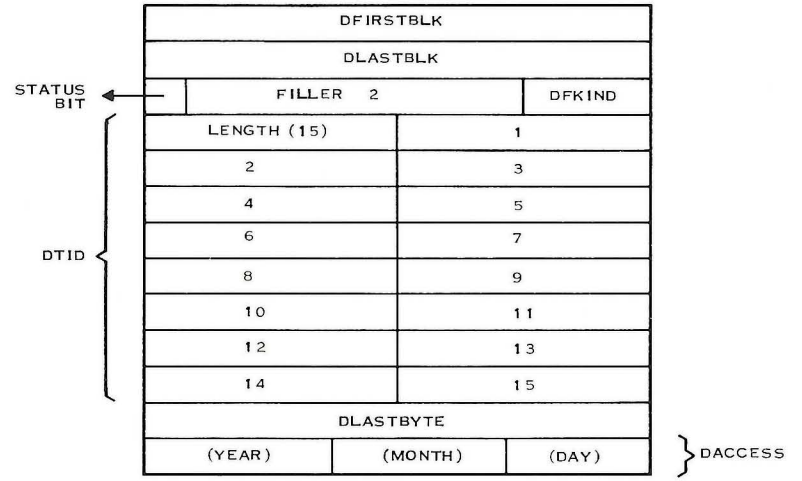
Directories

The following figure illustrates the structure of a directory, as on a disk or other storage device.

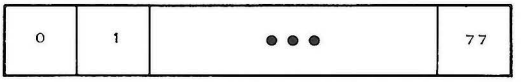
DIRENTRY RECORD (0)
FOR DFKIND=SECUREDIR , UNTYPED FILE (DIR [0])



DIRENTRY RECORD (1-77)



DIRECTORY: ARRAY [0..77] OF DIRENTRY;



2284136

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 display unit 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 a part number is added to a text file, it is stored in the last two words of the header (end ob 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 a page. Each page contains lines of text terminated by pressing the **RETURN** key. 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 will follow the Jensen and Wirth standard for files of type TEXT.

4

Program Execution

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 1, The p-Machine.

BUILDENV recursively traverses the segments used by a program. For each segment, it initializes an E_Vec, 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_Vec, 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, such as a used unit, used by the compilation. 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 the space bar, 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.

A

p-Machine Opcodes (Alphabetic Order)

Opcode	Dec	Hex	Description
ABI	224	E0	Absolute Value Integer
ABR	227	E3	Absolute Value of Real
ADI	162	A2	Add Integers
ADJ	199	C7	Adjust Set
ADR	192	C0	Add Reals
ASTR	235	EB	Assign String
BNOT	159	9F	Boolean NOT
BPT	158	9E	Breakpoint
CAP	171	AB	Copy Array Parameter
CHK	203	CB	Check Subrange Bounds
CPF	151	97	Call Formal Procedure
CPG	145	91	Call Global Procedure
CPI	146	92	Call Intermediate Procedure
CPL	144	90	Call Local Procedure
CSP	172	AC	Copy String Parameter
CSTR	236	EC	Check String Index
CXG	148	94	Call Global External Procedure
CXI	149	95	Call Intermediate External Procedure
CXL	147	93	Call Local External Procedure
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 Integers
DVR	195	C3	Divide Reals
EFJ	210	D2	Equal False Jump
EQBYT	185	B9	Equal Byte Array
EQPWR	182	B6	Equal Set
EQREAL	205	CD	Equal Real

Opcode	Dec	Hex	Description
EQSTR	232	E8	Equal String
EQUI	176	B0	Equal Integer
FJP	212	D4	False Jump
FJPL	213	D5	False Long Jump
FLT	204	CC	Float Top-of-Stack
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 Field Pointer
INCI	237	ED	Increment Integer
IND	230	E6	Index and Load Word
INN	218	DA	Set Membership
INT	220	DC	Set Intersection
IXA	215	D7	Index Array
IXP	216	D8	Index Packed Array
JPL	139	8B	Unconditional Long Jump
LAE	155	9B	Load Extended Address
LAND	161	A1	Logical AND
LAO	134	86	Load Global Address
LCO	130	82	Load Constant Offset
LDA	136	88	Load Intermediate Address
LDB	167	A7	Load Byte
LDC	131	83	Load Multiple Word Constant
LDCB	128	80	Load Constant Byte
LDCI	129	81	Load Constant Word
LDCN	152	98	Load Constant NIL
LDCRL	242	F2	Load Real Constant
LDE	154	9A	Load Extended Word
LDL	135	87	Load Local Word
LDM	208	D0	Load Multiple Words
LDO	133	85	Load Global Word
LDP	201	C9	Load a Packed Field
LDRL	243	F3	Load Real
LEBYT	186	BA	Less Than or Equal Byte Array

Opcode	Dec	Hex	Description
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	Less Than or Equal Unsigned
LLA	132	84	Load Local Address
LNOT	229	E5	Logical NOT
LOD	137	89	Load Intermediate Word
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 Integers
MPR	194	C2	Multiply Reals
NAT	168	A8	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	Reserved
RESERVE3	252	FC	Reserved
RESERVE4	253	FD	Reserved
RESERVE5	254	FE	Reserved
RESERVE6	255	FF	Reserved
RND	191	BF	Round Real
RPU	150	96	Return from Procedure
SBI	163	A3	Subtract Integers
SBR	193	C1	Subtract Reals
SCPI1	239	EF	Short Call Intermediate Procedure
SCPI2	240	F0	Short Call Intermediate Procedure
SCXG1	112	70	Short Call External Global Procedure
SCXG2	113	71	Short Call External Global Procedure

Opcode	Dec	Hex	Description
SCXG3	114	72	Short Call External Global Procedure
SCXG4	115	73	Short Call External Global Procedure
SCXG5	116	74	Short Call External Global Procedure
SCXG6	117	75	Short Call External Global Procedure
SCXG7	118	76	Short Call External Global Procedure
SCXG8	119	77	Short Call External Global Procedure
SIGNAL	222	DE	Signal
SIND0	120	78	Short Index and Load Word
SIND1	121	79	Short Index and Load Word
SIND2	122	7A	Short Index and Load Word
SIND3	123	7B	Short Index and Load Word
SIND4	124	7C	Short Index and Load Word
SIND5	125	7D	Short Index and Load Word
SIND6	126	7E	Short Index and Load Word
SIND7	127	7F	Short Index and Load Word
SLDC0	0	00	Short Load Word Constant
SLDC1	1	01	Short Load Word Constant
SLDC2	2	02	Short Load Word Constant
SLDC3	3	03	Short Load Word Constant
SLDC4	4	04	Short Load Word Constant
SLDC5	5	05	Short Load Word Constant
SLDC6	6	06	Short Load Word Constant
SLDC7	7	07	Short Load Word Constant
SLDC8	8	08	Short Load Word Constant
SLDC9	9	09	Short Load Word Constant
SLDC10	10	0A	Short Load Word Constant
SLDC11	11	0B	Short Load Word Constant
SLDC12	12	0C	Short Load Word Constant
SLDC13	13	0D	Short Load Word Constant
SLDC14	14	0E	Short Load Word Constant
SLDC15	15	0F	Short Load Word Constant
SLDC16	16	10	Short Load Word Constant

Opcode	Dec	Hex	Description
SLDC17	17	11	Short Load Word Constant
SLDC18	18	12	Short Load Word Constant
SLDC19	19	13	Short Load Word Constant
SLDC20	20	14	Short Load Word Constant
SLDC21	21	15	Short Load Word Constant
SLDC22	22	16	Short Load Word Constant
SLDC23	23	17	Short Load Word Constant
SLDC24	24	18	Short Load Word Constant
SLDC25	25	19	Short Load Word Constant
SLDC26	26	1A	Short Load Word Constant
SLDC27	27	1B	Short Load Word Constant
SLDC28	28	1C	Short Load Word Constant
SLDC29	29	1D	Short Load Word Constant
SLDC30	30	1E	Short Load Word Constant
SLDC31	31	1F	Short Load Word Constant
SLDL1	32	20	Short Load Local Word
SLDL2	33	21	Short Load Local Word
SLDL3	34	22	Short Load Local Word
SLDL4	35	23	Short Load Local Word
SLDL5	36	24	Short Load Local Word
SLDL6	37	25	Short Load Local Word
SLDL7	38	26	Short Load Local Word
SLDL8	39	27	Short Load Local Word
SLDL9	40	28	Short Load Local Word
SLDL10	41	29	Short Load Local Word
SLDL11	42	2A	Short Load Local Word
SLDL12	43	2B	Short Load Local Word
SLDL13	44	2C	Short Load Local Word
SLDL14	45	2D	Short Load Local Word
SLDL15	46	2E	Short Load Local Word
SLDL16	47	2F	Short Load Local Word
SLDO1	48	30	Short Load Global Word
SLDO2	49	31	Short Load Global Word
SLDO3	50	32	Short Load Global Word
SLDO4	51	33	Short Load Global Word
SLDO5	52	34	Short Load Global Word
SLDO6	53	35	Short Load Global Word
SLDO7	54	36	Short Load Global Word

Opcode	Dec	Hex	Description
SLDO8	55	37	Short Load Global Word
SLDO9	56	38	Short Load Global Word
SLDO10	57	39	Short Load Global Word
SLDO11	58	3A	Short Load Global Word
SLDO12	59	3B	Short Load Global Word
SLDO13	60	3C	Short Load Global Word
SLDO14	61	3D	Short Load Global Word
SLDO15	62	3E	Short Load Global Word
SLDO16	63	3F	Short Load Global Word
SLLA1	96	60	Short Load Local Address
SLLA2	97	61	Short Load Local Address
SLLA3	98	62	Short Load Local Address
SLLA4	99	63	Short Load Local Address
SLLA5	100	64	Short Load Local Address
SLLA6	101	65	Short Load Local Address
SLLA7	102	66	Short Load Local Address
SLLA8	103	67	Short Load Local Address
SLOD1	173	AD	Short Load Intermediate Word
SLOD2	174	AE	Short Load Intermediate Word
SPR	209	D1	Store Processor Register
SRO	165	A5	Store Global Word
SRS	188	BC	Build a Subrange Set
SSTL1	104	68	Short Store Local Word
SSTL2	105	69	Short Store Local Word
SSTL3	106	6A	Short Store Local Word
SSTL4	107	6B	Short Store Local Word
SSTL5	108	6C	Short Store Local Word
SSTL6	109	6D	Short Store Local Word
SSTL7	110	6E	Short Store Local Word
SSTL8	111	6F	Short Store Local Word
STB	200	C8	Store Byte
STE	217	D9	Store Extended Word
STL	164	A4	Store Local Word
STM	142	8E	Store Multiple Words
STO	196	C4	Store Indirect
STP	202	CA	Store into a Packed Field
STR	166	A6	Store Intermediate Word
STRL	244	F4	Store Real

Opcode	Dec	Hex	Description
SWAP	189	BD	Swap
TJP	241	F1	True Jump
TNC	190	BE	Truncate Real
UJP	138	8A	Unconditional Jump
UNI	219	DB	Set Union
WAIT	223	DF	Wait
XJP	214	D6	Case Jump

B

P-Machine Opcodes (Numeric Order)

Dec	Hex	Opcode	Description
0	00	SLDC0	Short Load Word Constant
1	01	SLDC1	Short Load Word Constant
2	02	SLDC2	Short Load Word Constant
3	03	SLDC3	Short Load Word Constant
4	04	SLDC4	Short Load Word Constant
5	05	SLDC5	Short Load Word Constant
6	06	SLDC6	Short Load Word Constant
7	07	SLDC7	Short Load Word Constant
8	08	SLDC8	Short Load Word Constant
9	09	SLDC9	Short Load Word Constant
10	0A	SLDC10	Short Load Word Constant
11	0B	SLDC11	Short Load Word Constant
12	0C	SLDC12	Short Load Word Constant
13	0D	SLDC13	Short Load Word Constant
14	0E	SLDC14	Short Load Word Constant
15	0F	SLDC15	Short Load Word Constant
16	10	SLDC16	Short Load Word Constant
17	11	SLDC17	Short Load Word Constant
18	12	SLDC18	Short Load Word Constant
19	13	SLDC19	Short Load Word Constant
20	14	SLDC20	Short Load Word Constant
21	15	SLDC21	Short Load Word Constant
22	16	SLDC22	Short Load Word Constant
23	17	SLDC23	Short Load Word Constant
24	18	SLDC24	Short Load Word Constant
25	19	SLDC25	Short Load Word Constant
26	1A	SLDC26	Short Load Word Constant
27	1B	SLDC27	Short Load Word Constant
28	1C	SLDC28	Short Load Word Constant
29	1D	SLDC29	Short Load Word Constant

Dec	Hex	Opcode	Description
30	1E	SLDC30	Short Load Word Constant
31	1F	SLDC31	Short Load Word Constant
32	20	SLDL1	Short Load Local Word
33	21	SLDL2	Short Load Local Word
34	22	SLDL3	Short Load Local Word
35	23	SLDL4	Short Load Local Word
36	24	SLDL5	Short Load Local Word
37	25	SLDL6	Short Load Local Word
38	26	SLDL7	Short Load Local Word
39	27	SLDL8	Short Load Local Word
40	28	SLDL9	Short Load Local Word
41	29	SLDL10	Short Load Local Word
42	2A	SLDL11	Short Load Local Word
43	2B	SLDL12	Short Load Local Word
44	2C	SLDL13	Short Load Local Word
45	2D	SLDL14	Short Load Local Word
46	2E	SLDL15	Short Load Local Word
47	2F	SLDL16	Short Load Local Word
48	30	SLDO1	Short Load Global Word
49	31	SLDO2	Short Load Global Word
50	32	SLDO3	Short Load Global Word
51	33	SLDO4	Short Load Global Word
52	34	SLDO5	Short Load Global Word
53	35	SLDO6	Short Load Global Word
54	36	SLDO7	Short Load Global Word
55	37	SLDO8	Short Load Global Word
56	38	SLDO9	Short Load Global Word
57	39	SLDO10	Short Load Global Word
58	3A	SLDO11	Short Load Global Word
59	3B	SLDO12	Short Load Global Word
60	3C	SLDO13	Short Load Global Word
61	3D	SLDO14	Short Load Global Word
62	3E	SLDO15	Short Load Global Word
63	3F	SLDO16	Short Load Global Word
64	40		Unused
.	.		Unused
.	.		Unused

Dec	Hex	Opcode	Description
.	.		Unused
95	5F		Unused
96	60	SLLA1	Short Load Local Address
97	61	SLLA2	Short Load Local Address
98	62	SLLA3	Short Load Local Address
99	63	SLLA4	Short Load Local Address
100	64	SLLA5	Short Load Local Address
101	65	SLLA6	Short Load Local Address
102	66	SLLA7	Short Load Local Address
103	67	SLLA8	Short Load Local Address
104	68	SSTL1	Short Store Local Word
105	69	SSTL2	Short Store Local Word
106	6A	SSTL3	Short Store Local Word
107	6B	SSTL4	Short Store Local Word
108	6C	SSTL5	Short Store Local Word
109	6D	SSTL6	Short Store Local Word
110	6E	SSTL7	Short Store Local Word
111	6F	SSTL8	Short Store Local Word
112	70	SCXG1	Short Call External Global Procedure
113	71	SCXG2	Short Call External Global Procedure
114	72	SCXG3	Short Call External Global Procedure
115	73	SCXG4	Short Call External Global Procedure
116	74	SCXG5	Short Call External Global Procedure
117	75	SCXG6	Short Call External Global Procedure
118	76	SCXG7	Short Call External Global Procedure
119	77	SCXG8	Short Call External Global Procedure
120	78	SIND0	Short Index and Load Word
121	79	SIND1	Short Index and Load Word
122	7A	SIND2	Short Index and Load Word
123	7B	SIND3	Short Index and Load Word

Dec	Hex	Opcode	Description
124	7C	SIND4	Short Index and Load Word
125	7D	SIND5	Short Index and Load Word
126	7E	SIND6	Short Index and Load Word
127	7F	SIND7	Short Index and Load Word
128	80	LDCB	Load Constant Byte
129	81	LDCI	Load Constant Word
130	82	LCO	Load Constant Offset
131	83	LDC	Load Multiple Word Constant
132	84	LLA	Load Local Address
133	85	LDO	Load Global Word
134	86	LAO	Load Global Address
135	87	LDL	Load Local Word
136	88	LDA	Load Intermediate Address
137	89	LOD	Load Intermediate Word
138	8A	UJP	Unconditional Jump
139	8B	JPL	Unconditional Long Jump
140	8C	MPI	Multiply Integers
141	8D	DVI	Divide Integers
142	8E	STM	Store Multiple Words
143	8F	MODI	Modulo Integers
144	90	CPL	Call Local Procedure
145	91	CPG	Call Global Procedure
146	92	CPI	Call Intermediate Procedure
147	93	CXL	Call Local External Procedure
148	94	CXG	Call Global External Procedure
149	95	CXI	Call Intermediate External Procedure
150	96	RPU	Return from Procedure
151	97	CPF	Call Formal Procedure
152	98	LDCN	Load Constant NIL
153	99	LSL	Load Static Link
154	9A	LDE	Load Extended Word
155	9B	LAE	Load Extended Address
156	9C	NOP	No Operation
157	9D	LPR	Load Processor Register
158	9E	BPT	Breakpoint
159	9F	BNOT	Boolean NOT
160	A0	LOR	Logical OR

Dec	Hex	Opcode	Description
161	A1	LAND	Logical AND
162	A2	ADI	Add Integers
163	A3	SBI	Subtract Integers
164	A4	STL	Store Local Word
165	A5	SRO	Store Global Word
166	A6	STR	Store Intermediate Word
167	A7	LDB	Load Byte
168	A8	NAT	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 Word
174	AE	SLOD2	Short Load Intermediate Word
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	Build a Subrange Set
189	BD	SWAP	Swap
190	BE	TNC	Truncate Real
191	BF	RND	Round Real
192	C0	ADR	Add Reals
193	C1	SBR	Subtract Reals
194	C2	MPR	Multiply Reals
195	C3	DVR	Divide Reals
196	C4	STO	Store Indirect
197	C5	MOV	Move

Dec	Hex	Opcode	Description
198	C6	DUPR	Duplicate Real
199	C7	ADJ	Adjust Set
200	C8	STB	Store Byte
201	C9	LDP	Load a Packed Field
202	CA	STP	Store into a Packed Field
203	CB	CHK	Check Subrange Bounds
204	CC	FLT	Float Top-of-Stack
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 Words
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 Word
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 of Real
228	E4	NGR	Negate Real
229	E5	LNOT	Logical NOT
230	E6	IND	Index and Load Word
231	E7	INC	Increment Field Pointer
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

Dec	Hex	Opcode	Description
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	Short Call Intermediate Procedure
241	F1	TJP	True Jump
242	F2	LDCRL	Load Real Constant
243	F3	LDRL	Load Real
244	F4	STRL	Store Real
245	F5		Unused
.	.		Unused
.	.		Unused
.	.		Unused
249	F9		Unused
250	FA	RESERVE1	Reserved
251	FB	RESERVE2	Reserved
252	FC	RESERVE3	Reserved
253	FD	RESERVE4	Reserved
254	FE	RESERVE5	Reserved
255	FF	RESERVE6	Reserved

C

ASCII Codes

Decimal	Octal	Hexadecimal	Character
0	000	00	NUL
1	001	01	SOH
2	002	02	STX
3	003	03	ETX
4	004	04	EOT
5	005	05	ENQ
6	006	06	ACK
7	007	07	BEL
8	010	08	BS
9	011	09	HT
10	012	0A	LF
11	013	0B	VT
12	014	0C	FF
13	015	0D	CR
14	016	0E	SO
15	017	0F	SI
16	020	10	DLE
17	021	11	DC1
18	022	12	DC2
19	023	13	DC3
20	024	14	DC4
21	025	15	NAK
22	026	16	SYN
23	027	17	ETB
24	030	18	CAN
25	031	19	EM
26	032	1A	SUB
27	033	1B	ESC
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US
32	040	20	SP

Decimal	Octal	Hexadecimal	Character
33	041	21	!
34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	'
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I

Decimal	Octal	Hexadecimal	Character
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	/
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	‘
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s

Decimal	Octal	Hexadecimal	Character
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL

Glossary

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 that changes during program execution (or is not 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 does not 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 that is executed after the completion of some other program, and uses as input the output of that previous program. A postprocessor that creates output that 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 that overwrites or modifies itself during execution, thus changing its meaning. This is not 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 that does not change throughout program execution and which 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. 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 — 16 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

THREE-MONTH LIMITED WARRANTY TEXAS INSTRUMENTS PROFESSIONAL COMPUTER SOFTWARE MEDIA

TEXAS INSTRUMENTS INCORPORATED EXTENDS THIS CONSUMER WARRANTY ONLY TO THE ORIGINAL CONSUMER/PURCHASER.

WARRANTY DURATION

The media is warranted for a period of three (3) months from the date of original purchase by the consumer.

Some states do not allow the exclusion or limitation of incidental or consequential damages or limitations on how long an implied warranty lasts, so the above limitations or exclusions may not apply to you.

WARRANTY COVERAGE

This limited warranty covers the cassette or diskette (media) on which the computer program is furnished. It does not extend to the program contained on the media or the accompanying book materials (collectively the Program). The media is warranted against defects in material or workmanship. THIS WARRANTY IS VOID IF THE MEDIA HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE, OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIALS OR WORKMANSHIP.

PERFORMANCE BY TI UNDER WARRANTY

During the above three-month warranty period, defective media will be replaced when it is returned postage prepaid to a Texas Instruments Service Facility listed below or an authorized Texas Instruments Professional Computer Dealer with a copy of the purchase receipt. The replacement media will be warranted for three months from date of replacement. Other than the postage requirement (where allowed by state law), no charge will be made for the replacement. TI strongly recommends that you insure the media for value prior to mailing.

WARRANTY AND CONSEQUENTIAL DAMAGES DISCLAIMERS

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER ARISING OUT OF THE PURCHASE OR USE OF THE MEDIA. THESE EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED BY, COST OF REMOVAL OR REINSTALLATION, OUTSIDE COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF BUSINESS.

LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

U.S. Residents:

Texas Instruments
Service Facility
P.O. Box 1444, MS 7758
Houston, Texas 77001

Canadian Residents:

Geophysical Service Inc.
41 Shelley Road
Richmond Hill, Ontario
Canada L4C 5G4

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information.

Texas Instruments
Consumer Service
831 South Douglas St.
Suite 119
El Segundo, California 90245
(213) 973-2591

Texas Instruments
Consumer Service
6700 S.W. 105th
Kristin Square, Suite 110
Beaverton, Oregon 97005
(503) 643-6758

IMPORTANT NOTICE OF DISCLAIMER REGARDING THE PROGRAM

The following should be read and understood before using the software media and Program.

TI does not warrant that the Program will be free from error or will meet the specific requirements of the purchaser/user. The purchaser/user assumes complete responsibility for any decision made or actions taken based on information obtained using the Program. Any statements made concerning the utility of the Program are not to be construed as expressed or implied warranties.

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE PROGRAM AND MAKES ALL PROGRAMS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THE PROGRAM. THESE EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED BY, COST OF REMOVAL OR REINSTALLATION, OUTSIDE COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF BUSINESS. THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE PROGRAM. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER BY ANY OTHER PARTY AGAINST THE PURCHASER/USER OF THE PROGRAM.

COPYRIGHT

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Internal Architecture
TI Part No. 2232400-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

City/State/Zip Code: _____

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Internal Architecture
TI Part No. 2232400-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

City/State/Zip Code: _____

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Internal Architecture
TI Part No. 2232400-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

City/State/Zip Code: _____

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

Texas Instruments reserves the right to change its product and service offerings at any time without notice.

TEXAS INSTRUMENTS

2232400-0001

Printed in U.S.A.