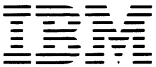


Personal Computer
Computer Language
Series

INTERNAL ARCHITECTURE GUIDE

for the UCSD p-System™ Version IV.0

Produced by SofTech Microsystems, Inc.

First Edition (January 1982)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

CONTENTS

CHAPTER 1. INTRODUCTION

Contents

NOTES

Purpose of this Guide

This guide describes the internal design of the UCSD p-System: 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 (or FORTRAN).

It should serve as a guide and reference for more advanced users of the System, but is not intended to be a standalone 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 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 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 System capabilities: for instance, new language translators, new System utilities, or Interpreters for additional processors.

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 a user to take such steps).

A Brief History of the System

The software system 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 (on campus it was popularly called "The Beast"). 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 Beast 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. Students were expected to buy their own floppy disk, and use it for storing the System and their own programs.

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

The user interface of the System, by which we mean the philosophy of displaying a promptline at every level of the System, and organizing these promptlines in a tree structure, was intended to be easy to learn for the complete novice, yet usable (i.e., not cumbersome) for the experienced user. This proved very successful, and has been retained.

The interpretive 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 codefiles as small as possible. The tradeoff in execution time was felt to be an affordable cost (time has borne out this decision).

The UCSD p-System is implemented on the IBM Personal Computer, with an Interpreter tailored to the 8088/87 processor.

NOTES

CHAPTER 2. THE P-MACHINE

Contents

Overview

The P-machine is an idealized machine. The Operating System itself, 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 codefiles in the System consist of either P-code or native code (that is, 8088 machine code).

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.

Interpretive Execution

The "P" in "P-code" and "P-machine" stands for "pseudo." The Interpreter is a program written in 8088/87 code. It is responsible for executing P-code instructions, and controlling I/O.

At runtime, the user's program (or a portion of it) is in main memory. The Interpreter fetches each P-code instruction, in sequence, and performs the appropriate action. The process of bootstrapping involves loading the Interpreter 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) Last-In-First-Out manner. Between them is an area of memory that is partly unused, but may also contain the Codepool (see below).

The Heap is more fully described in Chapter 4. The Codepool is also described in Chapter 4.

Code Segments

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

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

When a program is eX(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 runtime to make references (such as procedure calls) from one segment to another.

The segments of a running program compete with each other for space in main memory. If the System's Codepool is internal (between the Stack and the Heap) then the segments also compete with the Stack and the Heap. The principal constraint (as far as code segments are concerned) is that both the calling and called segment must both be present in main memory for an inter-segment call to succeed.

Segments in main memory are all stored contiguously in an area called the Codepool. The Codepool resides either between the Stack and the Heap (an internal pool) and may be moved about to create more room, or outside the Stack/Heap space (an external pool) and may not be moved.

Code segments are described in this chapter. Codepool handling is described in Chapter 4.

Device I/O

Device I/O and control is accomplished by calls from the language level to routines within the Interpreter. The device I/O routines then call on the routines of the Interpreter's BIOS (for Basic I/O Subsystem), and the BIOS routines control the peripheral hardware directly. I/O environment dependencies are thus isolated in the BIOS for convenience. The BIOS is dealt with in Chapter 3.

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..255.

At compile time, segments are assigned a name and a number. The name is 8 characters long. It is used by the Operating System to handle inter-segment references at associate time. It is also used when maintaining codefiles with LIBRARY. The number is used to reference the segment at runtime.

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)

Figure 2-1 illustrates a code segment as it would be loaded into memory. The various substructures of a code segment are described below.

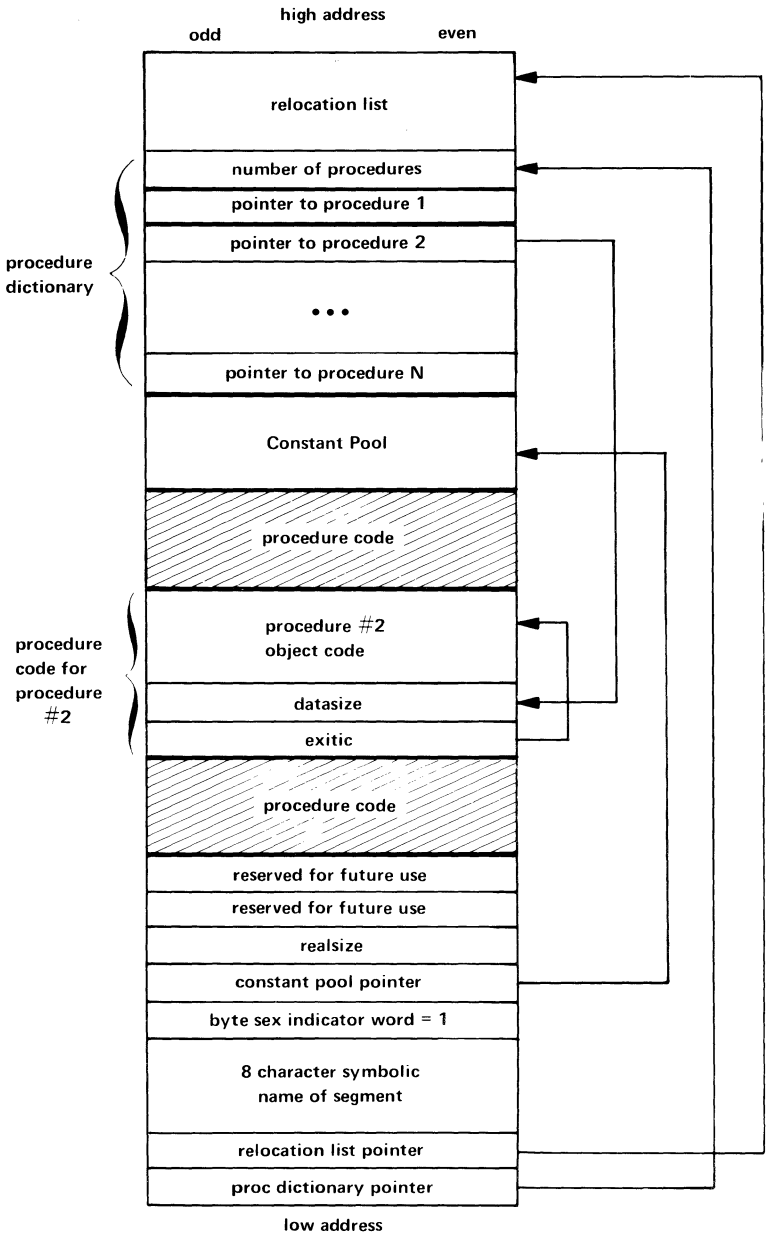


Figure 2-1. Executable Code Segment Format

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..255. A routine's number is an index into the routine dictionary: the n'th word in the dictionary contains a pointer to the code for routine n.

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

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

Routine Code

The code of a routine consists of two words: 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 runtime.

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

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

The location of the constant pool is contained in the constant pool pointer, a 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.

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

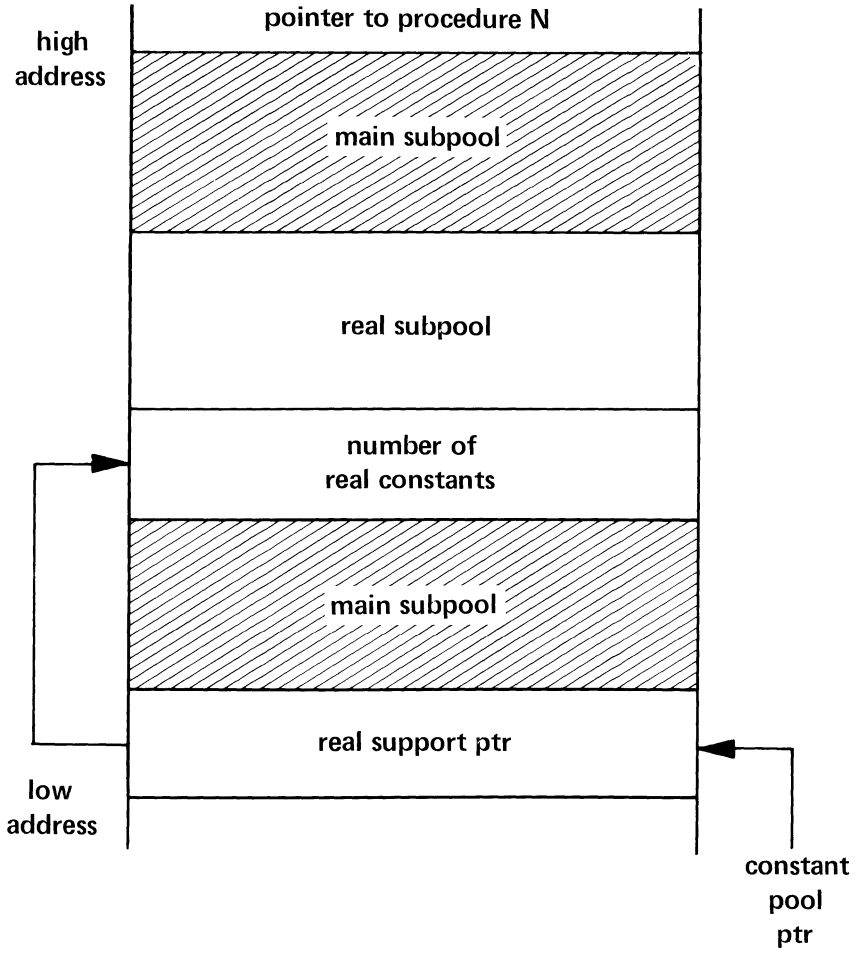


Figure 2-2. Constant Pool

Real constants are generated for either 32- or 64-bit floating point formats. Both the 32-bit and the 64-bit formats are available on the IBM Personal Computer.

The Pascal Compiler is configured (when it is 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 if it does 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 (when it is read into memory it is packed into a two word form). 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 11 decimal. The second mantissa word contains a negative value. And the exponent word is -1:

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 (when read into memory it is always packed into a 4-word form). 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 runtime overhead for programs that are memory-bound.

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 exclusively P-code are completely position-independent; no relocation list is needed.

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

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

```
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 Interpreter'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.

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

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 runtime 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 (i.e., local) series of segment numbers, and its own environment vector. In this way, a particular unit may be referenced by more than one unit, and each unit that references it may use a different segment number. (More about environment vectors appears in "Code Segment Environments" in this chapter.)

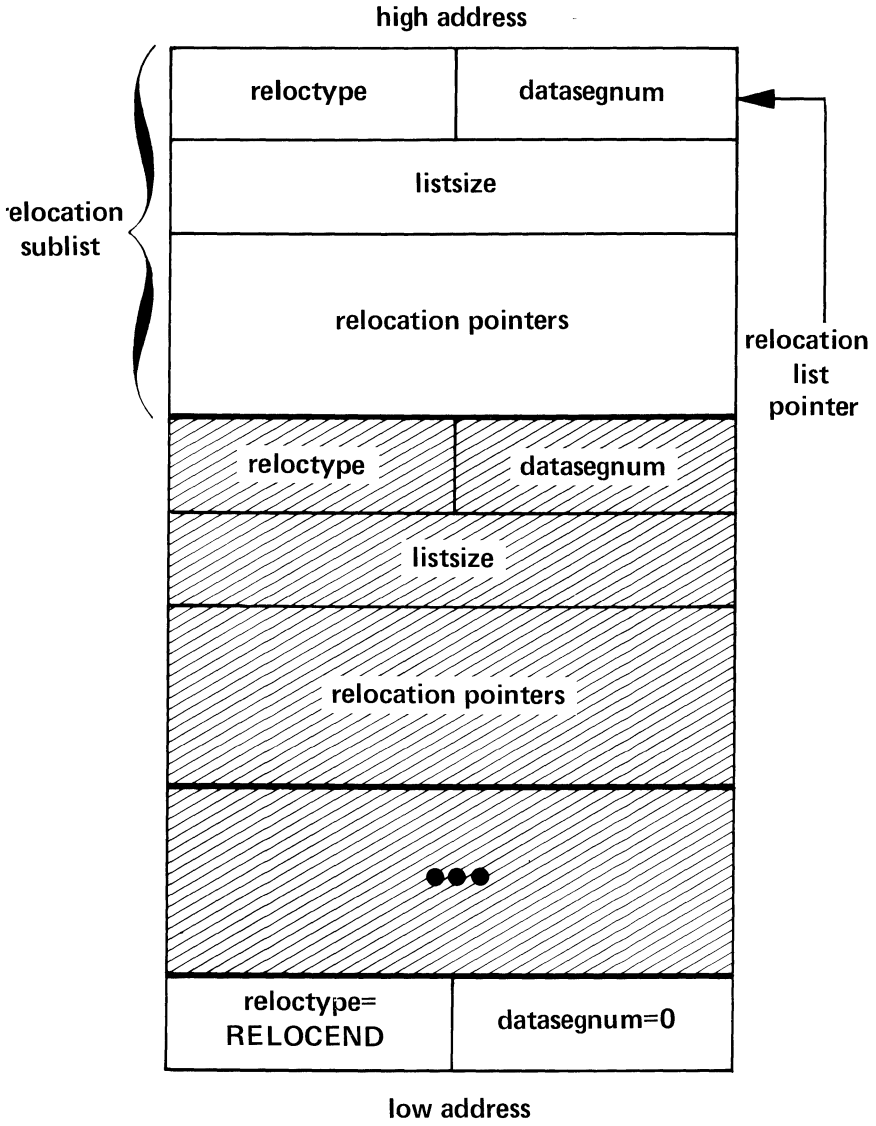


Figure 2-3. Relocation List

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

The segment reference list, when present, is located above the relocation list (it grows toward higher memory addresses). The list is used by the Operating System at associate time. It does not occupy any space in memory during the program's execution.

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

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 (described below) contains the number of words in the segment reference list. The Code_Leng field in the segment dictionary can be used as a 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 a 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 SegNum>, 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 info. Segments output by a compiler have Linker info only if they contain an EXTERNAL

routine. Only principal segments may contain EXTERNAL routines.

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

If a code segment has Linker info, the HasLinkerInfo Boolean in Seg_Misc in the segment dictionary is TRUE. The starting block of Linker info, relative to the start of the codefile, can be calculated from the formula:

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

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

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

The following fragment of pseudo-Pascal code describes a Linker info 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: (Home Proc: 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 info 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 8 words; all unused words should be zero.

For these types of Linker info 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 at by PtrList. This offset must be in the correct addressing mode: i.e., 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 at 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 at 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 at 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 info records. Name should be blank-filled.

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

		Prog_Seg	Unit_Seg	Seprt_Seg
	GlobRef			yes
	PublRef			yes
PrivRef	PrivRef			yes
	ConstRef			yes
	ExtProc	yes	yes	
ExtFunc	ExtFunc	yes	yes	
	SepProc			yes
	SepFunc			yes
	GlobDef			yes
	PublDef	yes	yes	
	ConstDef	yes	yes	
	EOFMark	yes	yes	yes

Codefile Organization

The Segment Dictionary

The first block of a codefile 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 codefile. 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: the information describing a segment is found by using a single index value to select a component from each of these arrays. Entries in the segment dictionary describe only segments whose code bodies are included in the codefile.

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_Pseudo, M_6809, M_PDP_11, M_8080,
                   M_Z_80, M_GA_440, M_6502,
                   M_6800, M_9900, M_8088,
                   M_Z8000, M_68000);

{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_Family:
  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_Family};
```

```
        Next_Dict: integer;
                {block number of next
                 dictionary record}
        Filler: ARRAY [0..6] 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 codefile). 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 8 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 info resides in the blocks that directly follow the segment reference list. Linker info 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 Codepool; 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 itself, 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 codefile. The INTERFACE text section can appear anywhere within the codefile that contains the code segment it describes. The Seg_Text array entry, in conjunction with the Text_Size field in the Seg_Famly 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 textfile, 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 be M_8088. If the segment consists exclusively of P-code, then its M Type is M Pseudo. Major_Version gives the version of the P-machine on which the codefile 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 8 characters of the parent compilation unit's name, stored in Prog_Name. If this name is not known at codefile 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 codefile. In the last record of the segment dictionary, Next_Dict is zero.

Filler is reserved for future use and is always 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 codefile. 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 codefile, 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 codefile, and if necessary, byte-swap the word-oriented fields of the dictionary. As long as all programs are compiled and run on an IBM Personal Computer, the byte sex will be consistent (least significant byte first).

Assembler-Generated Codefiles

Codefiles generated by the 8088/87 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 at 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 Data_Size word generated by the assembler for each routine should have a value of -1 (0FFFF HEX): this indicates a data size of zero that is one's complemented, to signal that the first instruction of the code body is native code.

Finally, since the assembler-generated code segments 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 is blank-filled, and the DataSegNum field in the ListHeader record of all BaseRel relocation sublists is 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 also updates the Relocatable bit in the Seg_Misc array, depending on the information supplied in Linker info.

Code Segment Environments

Segment Information Blocks (SIBs)

A Segment Information Block (SIB) is a record that contains information about an "active" code segment. A code segment is active if it may be used by a program that is running. An 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;          {points to the description
                                   of the Codepool where the
                                   segment resides; is NIL if
                                   segment is on heap}
   Seg_Base: Mem_Ptr;           {byte offset within pool
                                   of segment's memory locn}
   Ref_Count: integer;          {# of active calls to the seg}
   Activity: integer;           {memory swap activity}
   Link_Count: integer;         {number of links to the SIB}
   Residency: -1..maxint;       {-1 = PosLock, 0 = Swap,
                                n = MemLock}
Seg_Name: PACKED ARRAY [0..7] OF CHAR;
   Seg_Leng: integer;           {# of words in segment}
   Seg_Addr: integer;           {disk address of segment}
   Vol_Info: VI_Ptr;            {pointer to disk drive info}
   Data_Size: integer;          {number of words in data seg}
   Res_SIBs: RECORD             {code pool management record}
          Next_SIB: SIB_P;          {next SIB in list}
          Prev_SIB: SIB_P;          {previous SIB in list}
          CASE Boolean OF           {scratch area}
             TRUE:  (Sort_SIB: SIB_P);
                       {next SIB in sort list}
             FALSE: (New_Loc: Mem_Ptr);
                       {temporary address}

     END {of Res_SIBs};
   MType: integer;
END {of SIB};
```

Seg_Pool points to a description of the codepool where the segment resides. If Seg_Pool is nil then the segment is position-locked on the Heap. The fields of the Codepool descriptor are described in Chapter 4.

Seg_Base contains the byte offset within the codepool of the code segment. If the code segment is not in memory, Seg_Base contains NIL.

Ref_Count contains the number of outstanding calls to the segment. It is incremented whenever a routine outside the segment executes a CXG to a routine within the segment. It is decremented whenever an RET from a routine within the segment returns to a routine outside the segment.

Activity contains a value based on the number of times a segment is used; it increases over time. It is incremented by 6 whenever a call is made to a routine outside the segment. It is also incremented by 6 whenever a routine within the segment returns to a routine outside the segment. Finally, it is incremented by 6 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 (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 (i.e., 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 8 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 (VI_Ptr) 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 Codepool. All SIBs of segments in the Codepool 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 Codepool.

The Operating System uses several data structures to manage code segments by maintaining active SIBs and managing the Codepool. 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 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_SIB: SIB_P; {pointer to SIB for seg num}
           Env_Vect: E_Vect_P; {pointer to environment}

           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 invoked.)

Env_SIB points to the segment's SIB. (Also placed on the Heap when the program is invoked.)

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.

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 1. Thus it may be indexed by local segment numbers (these must be 1 or greater). The Vec_Length field of the record may be considered to occupy the zero'th position of the map.

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

```
FUNCTION Build_Env (Seg_Dict): E_Rec_P;
  BEGIN
  IF outer block segment E_Rec exists in Unit_List THEN
     BEGIN
       increment Link_Count;
       return existing E_Rec_P
  END  ELSE  BEGIN
       create E_Vect;
       create Env_Data for outer block data space;
       IF there are USEd units indicated in Seg_Dict THEN
         FOR all 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 de-link 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.Seg_Vect DO
     BEGIN
       de_link E_REC^.SEG_SIB;
       DISPOSE (those E_Recs);
     END;
     DISPOSE (E_Rec.Seg_Vect);
  END
  END
```

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

Task Environments

A task is a routine that is executed concurrently with other routines. A task is implemented by three data structures: the body, the Task Information Block (TIB), and the task 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, and
- 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 (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;
                BP: MSCW_Ptr;
                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 limits on SP for this task.

MP and BP designate (respectively) the local and global 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..255. The lower 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 on a semaphore, Hang_Ptr points to that semaphore. If the task is not waiting on 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 is reserved for future use.

Env is a pointer to the task's E_Rec. The task's SIB (Segment Information Block) may be found through the E_Rec.

TIBIOResult contains the IORESULT that is local to the task.

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

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

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

Further information about tasks appears in Chapter 4. Figure 2-4 shows the layout of main memory while the System is running, including the location of task stacks as discussed in this section.

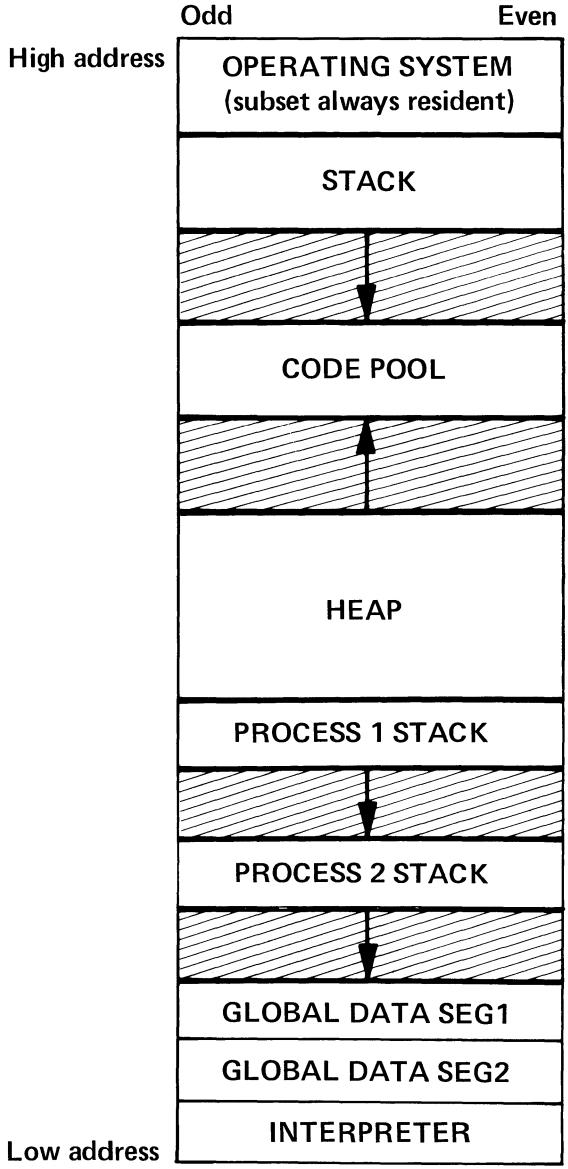


Figure 2-4. Main Memory Usage

P-Machine Instructions

The Intrinsic P_MACHINE

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

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

P_MACHINE
(<P-machine item> {, <P-machine item>})

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

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

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

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

3) Address Reference: if the first token of the item is ^, then the item is the specification of a variable, and P-code is generated which leaves the address of that variable on the stack.

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

Example:

Given these declarations:

```
CONST STO = 196;


TYPE Records = RECORD
                 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..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..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..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..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..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..32767. The word is always least-significant-byte-first.

Dynamic Operands. In the P-machine instruction descriptions below, stack-oriented dynamic operands of the P-codes 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..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, etc.

Word-block
A group of zero or more words.

Activation Records. An activation record is created for each invocation of an active routine. Figure 2-5 illustrates an activation record.

The parts of an activation record are:

1) Mark Stack. Five (full) words of housekeeping information:
   a) MSSTAT - pointer to the activation record of the lexical parent.

   b) MSDYN - pointer to the activation record of the caller.

   c) MSIPC - seg-relative byte pointer to point of call in the caller.

   d) MSENV - E_Rec pointer of the caller.

   e) MSPROC - procedure number of caller.

2) Local and temporary variables. This area is DataSize words long.

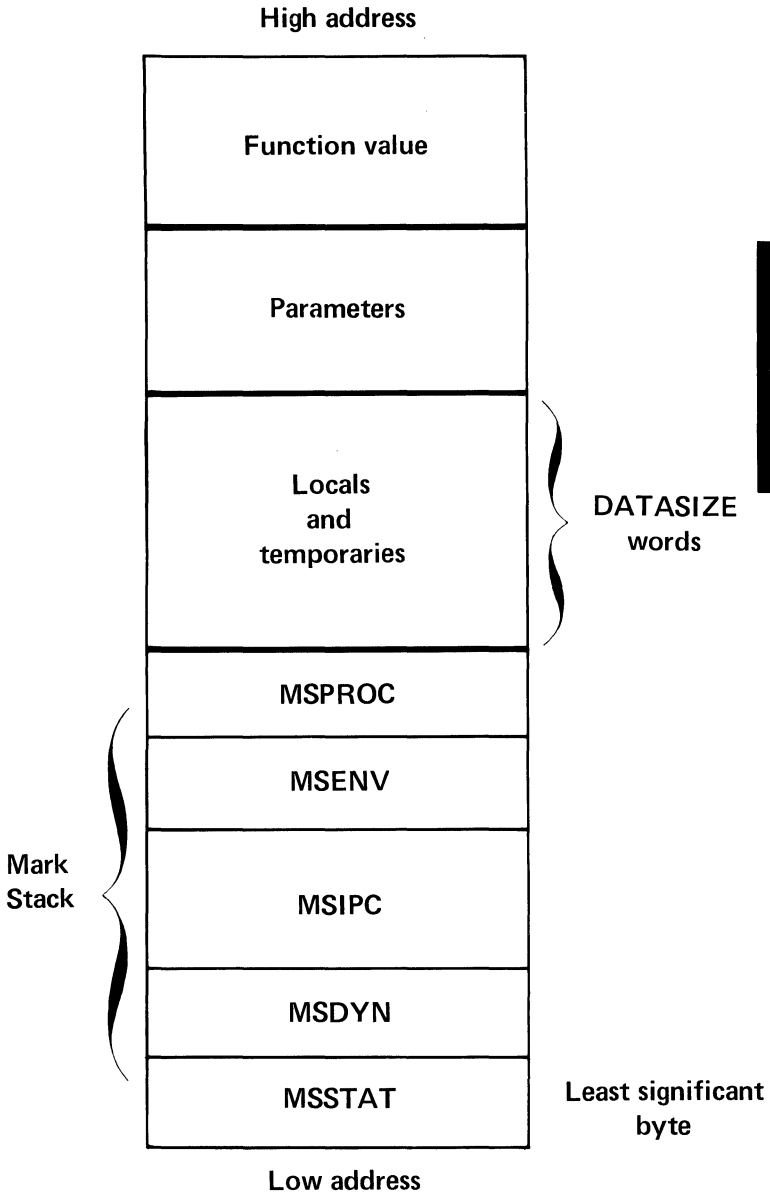


Figure 2-5. Procedure Activation Record

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

   a) Addresses - for VAR parameters, and record and array value parameters.

   b) Values - for other value parameters.

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

Conventions. The individual P-code instructions are 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 1).

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.

The Individual P-Code Instructions

Constant One-Word Loads.

SLDC	0..31 <>:<word>	Short Load Word Constant. Push the opcode, with the high byte zero.
LDCN	152 <>:<NIL>	Load Constant NIL. Push NIL.
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 constant pool of the current segment. Convert B to a seg-relative word offset. If operating on a byte-addressed machine, then convert to a byte offset. Push the offset on the Stack.

Local One-Word Loads and Stores

SLDL1 ... SLDL16	32 ... 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 ... SLLA8	96 ... 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 ... SSTL8	104 ... 111 <word>:<>	Short Store Local Word. Store TOS in the indicated offset in the local activation record.
STL	164 B <word>:<>	Store Local Word. Store TOS into word with offset B in the local activation record.

Global One-Word Loads and Store.

SLDO1 ... SLDO16	48 ... 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 Loads and Store.

SLOD1 SLOD2	173 B 174 B <>:<word>	Short Load Intermediate Word. Push the word at offset B in the activation record of the parent (LOD1) or grandparent (LOD2) 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 Loads and Store.

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 Loads and Store.

SIND0 ... SIND7	120 ... 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 B'th word in the record.
STO	196 <addr,word>: <>	Store Indirect. Store TOS into the word pointed to by TOS-1.

Multiple-Word Loads and Stores.

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. If less than B+20 words available to the Stack, issue a Stack fault.

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. If less than UB+20 words available to the Stack, issue a Stack fault.
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, 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 Packed Array of Char Parameter Copying. To copy value parameters of type string or packed array of char 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 to UB, the declared size (in bytes) of the destination formal parameter causes 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 in TOS-1.

Byte Load and Store.

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 significant byte of zero.
STB	200 <byte-ptr, word>:<>	Store Byte. Store byte TOS into the location specified by byte pointer TOS-1.

Packed Field Load and Store.

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 Indexing and Assignment.

MOV	197 UB, B <addr, addr>:<>	Move. Move B words from the source designated by TOS to the location 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 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 into 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. If TOS is 0, cause an execution error.
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 runtime 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> <Bool>	Greater than or Equal Integer. Push Boolean result of integer comparison TOS-1 > = TOS.

Real Arithmetic. All overflows and underflows cause a runtime 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 into TOS-1.
DVR	195 <real,real>: <real>	Divide Reals. Divide TOS into TOS-1. If TOS is 0, cause a runtime error.

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 (adding zeroes "between" TOS and TOS-1) or compression (chopping of high words of set), and discard its length word. After this operation, if less than 20 words are available to the Stack, cause a Stack fault.

SRS	188 <int,int>:<set>	Build a Subrange Set. The integers TOS and TOS-1 must be in [0..4079]. If not, cause a runtime error, else push the set [TOS-1..TOS]. If TOS-1 > TOS, push the empty set. Before this operation, if less than 20 words available to the Stack, cause a Stack fault.
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, 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 (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 from the host, and the corresponding mode is 2, swap the bytes of each word of that operand before doing the comparison. Push 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. 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 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 byte offset SB.
FJP	212 SB <Bool>:<>	False Jump. Jump by byte offset SB if TOS is false.
TJP	241 SB <Bool>:<>	True Jump. Jump by byte offset SB is TOS is true.

EFJ	210 SB <int,int>:<>	Equal False Jump. Jump by byte offset SB if TOS <> TOS-1.
NFJ	211 SB <int,int>:<>	Not Equal False Jump. Jump by byte offset SB if TOS = TOS-1.
JPL	139 W <>:<>	Unconditional Long Jump. Jump W bytes from current location.
FJPL	213 W <Bool>:<>	False Long Jump. Jump W bytes 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, 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..W2, then jump W3 words from the current location, where W3 is the contents of the word pointed at by TOS. Otherwise do nothing.

Routine Calls and Returns. For all procedure call instructions, after the MSCW and Datasize words have been pushed on the Stack, a check is made to see that there are still at least 40 words available between the Stack and the Codepool. If there are not, a Stack fault is issued.

For all calls to external procedures, issue a segment fault if the desired segment is not already in memory.

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 **SCPI2**	239 UB 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 ... SCXG8	112 UB ... 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. SCXG1 may refer to a procedure embedded in the Interpreter. If this is the case, an Interpreter table contains the procedure's location.
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. If the segment number is 1, then the procedure code may be embedded in the Interpreter; an Interpreter table contains its location.

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 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. If returning to different segment (Mark Stack E_Rec <> current E_Rec) then issue a segment fault if necessary. If procedure number in MSCW is < 0, return to EXITIC procedure, not MSCW's IPC.
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, is 0) or the offset of a string constant in the current segment. Cause a string overflow 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 <>:<>	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 current 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 value 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 8088-dependent.
NAT-INFO	168 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 it as a long form of NOP.
RESERVE1	250	These codes are reserved for use by the compiler to identify embedded compiler directives. They must not be explicitly generated by programs.
...	...	
RESERVE6	255	

NOTES

CHAPTER 3. LOW-LEVEL I/O

Contents

NOTES

Introduction to the I/O Subsystem

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

The RSP/IO is machine-independent, except for a portion called the Basic Input/Output Subsystem (BIOS). The BIOS is specific to the IBM Personal Computer hardware. Calls to routines in the BIOS are clearly defined.

Thus, we have the I/O Hierarchy shown in Figure 3-1. The user's I/O calls (e.g., READLN, WRITELN) are mapped by the Compiler and Operating System into calls to the RSP (i.e., UNITREAD, UNITWRITE). The RSP/IO itself calls the BIOS which controls the actual device operations. It is important for the reader to recognize that here we are discussing a synchronous I/O system. In other words, when an I/O request has been initiated by a user 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.

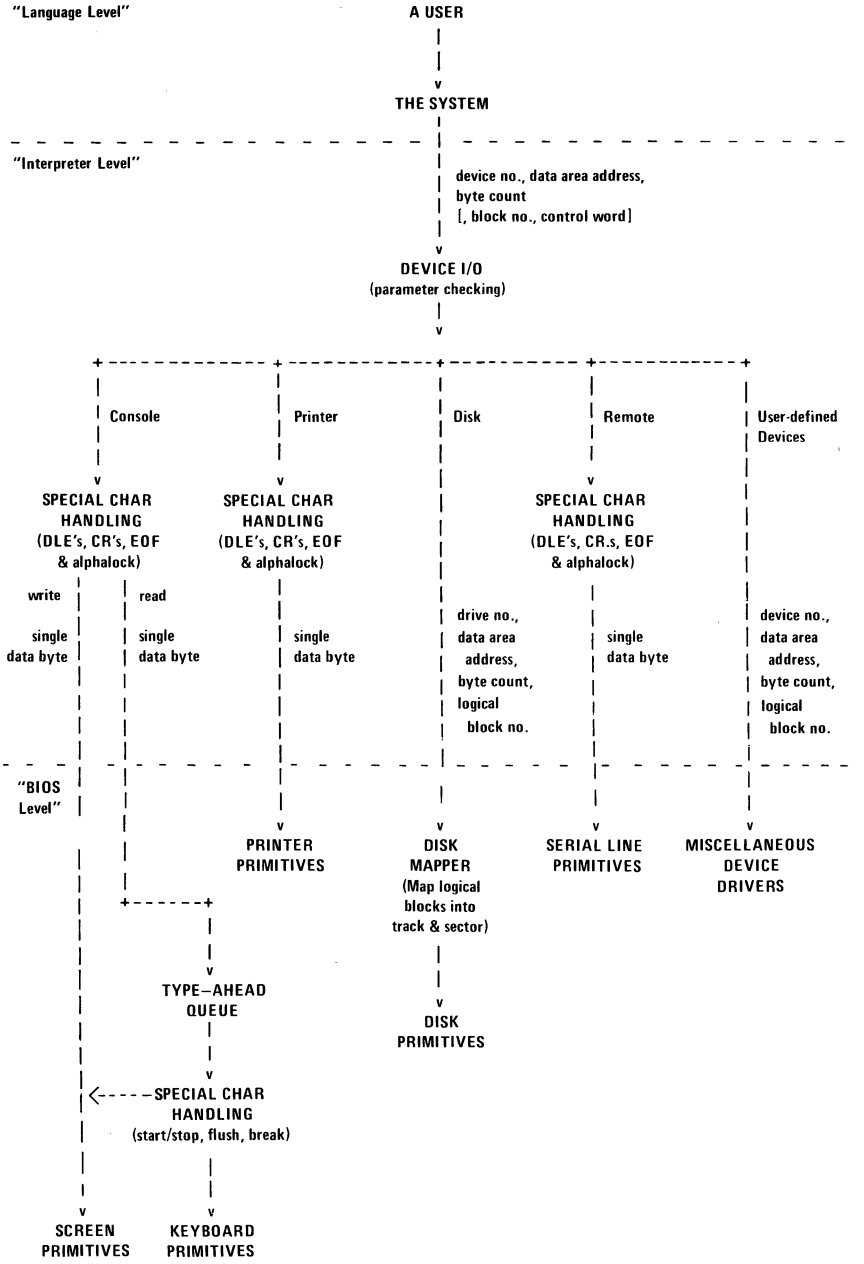


Figure 3-1. I/O Subsystem Hierarchy

The Language Level: Device I/O Routines

As mentioned above, all language-level I/O requests are eventually mapped by the Compiler and Operating System into calls to a group of intrinsic routines known as the Device I/O Routines. The programmer may call the Device Routines directly, or may use the standard I/O syntax of the language in use. The exact details of how this mapping is accomplished 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 should be realized that all I/O support at or below the device I/O level is implemented in 8086 assembly language.

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 Interpreter itself, instead of in KERNEL.

To the user making direct calls to Device I/O Routines, they look like any other intrinsic routine. If they actually were declared in Pascal, the declarations would have the following format (allowing a few "invalid" constructions 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> );

PROCEDURE UNITCLEAR (UNITNUMBER: 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 declarations above 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. Figure 3-2 is a list of the pre-defined unit numbers associated with each physical unit. The meaning of the other parameters is discussed later in this chapter.

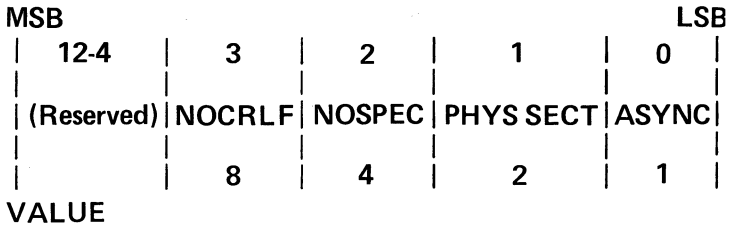
Unitnumber	Volume Name
0	<Reserved for the System>
1	CONSOLE:
2	SYSTERM:
3	<Reserved for the System>
4	<System disk>
5	<User disk>
6	PRINTER:
7	REMIN:
8	REMOUT:
9-127	<Reserved for future expansion>

Figure 3-2. Unitnumbers

High Device Numbers. The System reserves all device numbers above 8 for future expansion and special devices.

CONTROL Parameters

The CONTROL parameter to UNITREAD and UNITWRITE is a word used to pass special information to the RSP/IO and BIOS regarding the handling of the I/O request. The formats of the CONTROL words are shown in Figures 3-3 and 3-4.



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 PHYSSECT
Set implies "Physical Sector Mode" for disk I/O. Reset implies "Logical Block Mode" for disk I/O.

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

Bit 3 NOCRLF
Set implies no LFs are to be appended CRs during non-disk I/O. Reset implies LFs are to be appended to CRs during non-disk I/O.

Bits 4-15
Reserved for future expansion.

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

Figure 3-3. CONTROL word format for UNITREAD and UNITWRITE

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, or UNITCLEAR causes a "completion code" to be set

in the SYSCOM data area (SYSCOM, for SYStem COMmunication area, is conventionally the only data space that may be directly accessed by both the Operating System and the Interpreter). Programmers may test the completion code by using IORESULT.

The standard completion codes are given in Figure 3-4 below.

Code	Meaning
0	No error
1	Bad block, CRC error (parity)
2	Bad device number
3	Invalid I/O request
4	Data-com timeout
5	Volume is no longer on-line
6	File is no longer in directory
7	Invalid filename
8	No room; insufficient space on disk
9	No such volume on-line
10	No such filename 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	Invalid block number
18	Invalid buffer address
19 - 255	Reserved for future expansion

Figure 3-4. I/O Completion Codes

Logical Disk Structure

The System views a disk as a zero-based linear array of 512-byte logical blocks. The physical allocation units of a disk are known as sectors. The BIOS is responsible for mapping the logical structure of a System disk onto the physical structure of the device, i.e., mapping logical blocks onto physical sectors. It happens that on the IBM Personal Computer, physical sectors also have a size of 512 bytes.

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).

2) The parameter BYTESTOTRANSFER must be 0.

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..511 {BYTESperSECTOR-1} ]
     of BYTE;

TRACK = array[1..8 {SECTORSperTRACK} ]
     of SECTOR;

DISK = array[0..39 {TRACKSperDISK-1} ]
     of TRACK;

          (Note that here, we are using the convention that
          track numbers are zero-based but sector
          numbers start from one.)

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

type

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

          We 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. An I/O request in Physical Sector Mode simply causes a full sector (512 bytes) to be transferred. The programmer is responsible for ensuring that the data area is at least large enough for one physical sector.

The Interpreter Level: The RSP/IO

This section details the design and operation of the Input/Output portion of the Runtime Support Package (RSP/IO). While the design itself is processor- and hardware-independent, it is implemented in 8088 native code. Thus, the final product is 8088-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 such that an implementor of the RSP would know how to pop parameters off the Stack when the RSP is called, and how the Stack should look when the RSP returns. The detailed semantics of each routine are discussed in "Semantics" in this chapter.

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 the user's 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. The effective address is computed by simply adding the base and the offset. 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 in the case that 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.

The third item in the parameter list, BYTESTOTRANSFER, contains the number of bytes to move between the user's 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; if absent, the Compiler assigns them both the default value zero. LOGICALBLOCK is only relevant for disk reads or writes; as discussed in "Logical Disc Structure" in this chapter, it specifies the Pascal logical block to be accessed.

Parameter Stack Format. UNITREAD and UNITWRITE (see Figure 3-5) receive their parameters on the evaluation stack in the following order (each box represents a 16-bit quantity):

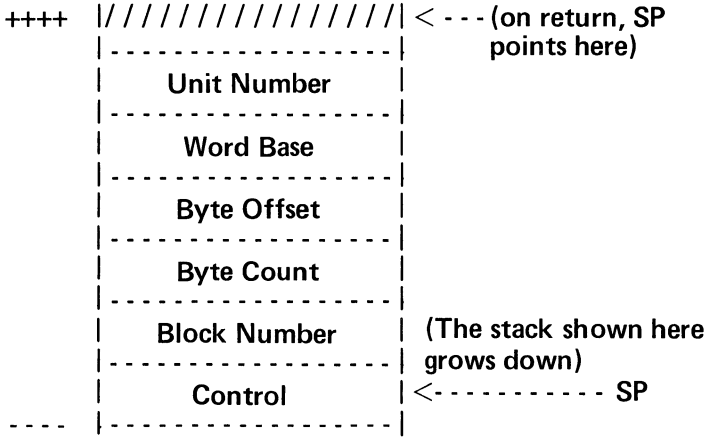


Figure 3-5. Stack State on Entering UNITREAD or UNITWRITE

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

UNITCLEAR

PROCEDURE UNITCLEAR (UNITNUMBER: INTEGER);

UNITCLEAR restores the specified unit to its "initial" state. At the RSP level, this means clearing any state flags pertaining to the specified device (see "Parameter Description" and "Parameter Stack Format" in this chapter). The "initial" state for each device at the BIOS level is defined in "Semantics" in this chapter. The stack format is as shown in Figure 3-6:

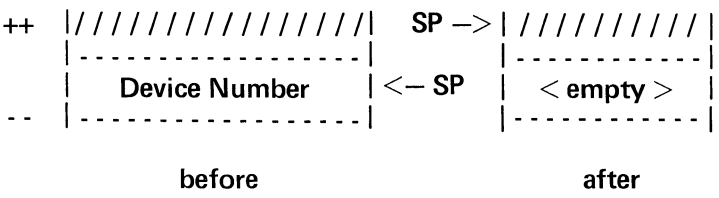


Figure 3-6. Stack States for UNITCLEAR

Semantics

This section details the processing performed by the RSP/IO. The primary function of the RSP/IO is to manage calls to the BIOS. Its secondary function is to handle some odd details that are described in this section.

Special Character Handling on Output

Output to the printer, console or remote units must properly handle Blank Compression Codes and Carriage Returns.

Blank Compression Code (DLE's). The System supports textfiles 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 (i.e., 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. Note also 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 NOSPEC Bit in CONTROL Parameter).

Carriage Return -- Line Feed. Textfiles contain ASCII CR's (decimal 13) at the end of lines. We define this character as meaning "New Line", i.e., 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 in CONTROL Parameter. When bit 3 (value 8) of the CONTROL parameter is set, the special handling accorded CR's is turned off, i.e., a LF is not automatically appended, and they are sent out like other characters.

Special Character Handling on Input

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

EOF Character. The EOF character, when received from the console, printer or remote devices, signals that the "end-of-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. On the IBM Personal Computer it is <ctrl-C>. Further discussion of the

soft characters used by the I/O Subsystem may be found in "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 we are reading from UNIT 1 (CONSOLE:), then the rest of the user's buffer, starting at the current position, is packed with nulls (decimal 0). For UNIT 2 (SYSTERM:), the printer and the remote, the EOF character is put into the user's 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 lower case alphabetic characters (i.e., a to z) received after the ALPHALOCK character will be converted to upper case. Receipt of another ALPHALOCK character will cause the case to revert back to non-converting 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.

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

NOSPEC Bit in CONTROL Parameter

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

The Machine Level: The BIOS

As explained above, the Basic Input/Output Subsystem is responsible for providing the actual access to I/O devices. This section describes the BIOS in detail.

The general scheme discussed below uses vectors from the RSP/IO to the BIOS subroutines for reading, writing, initializing and controlling, and answering status requests. The exact vector scheme and means of passing parameters on the IBM Personal Computer are shown in Appendix B for this chapter.

Design Goals

The speed of the BIOS code is fairly insignificant compared to the (slow) speed of the I/O devices that it handles. 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 for a BIOS (assuming correctness!) are (1) compactness and (2) clarity.

Completion Codes

All read, write, and initialization calls to the BIOS 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 "IORESULT and Completion Codes" in this chapter. 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 Invalid device number
3 Invalid operation on device
4 Undefined hardware error
5 Ring Buffer Overflow
6 Write protect; write attempt to protected disk
7 Invalid block number
8 Invalid buffer address
9 Device not on line

All other errors are considered hardware-dependent. For these, the BIOS returns codes in the range 128..255 (most are unused).

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 return a completion-code byte. For a summary of the BIOS calling requirements, see Appendix A to this chapter.

Console

Only one parameter is used for reading and writing: the data byte itself. The status request uses two parameters: the CONTROL word and the pointer to the status record. For initialization and control of the console, the BIOS uses 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 used: the data byte itself. To check the status, the CONTROL word and the pointer to the status record are used. For initialization and control, no parameters are used.

Disks

Reading and writing with disk devices uses five parameters:

1) A starting logical block number as described above.

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

3) The address of the data area in main memory.

4) A drive number (0..1).

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 uses a single parameter for reading and writing: the data byte itself. As with the devices just described, the status call uses the CONTROL word and the pointer to the status record. Initialization and control of the remote device uses no parameters.

Character Codes

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

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

Semantics

Console

In the following discussion, the console device is assumed to be a CRT display (as on the IBM Personal Computer).

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

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

LF <line feed> (hex 0A)
Moves 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 remain the same while the cursor moves downward. If the cursor is on the last line when the LF is issued, it remains in the same position while the rest of the display scrolls upward one line and the bottom line clears.

BEL <bell> (hex 07)
Sounds the speaker.

SP <space> (hex 20)
Writes a space at the current cursor position (erasing whatever is there) and advances the cursor position by one column. If the cursor is already at the last position in a line, it remains in its prior position. If the cursor is in the last column of the last line on the screen it remains where it was and the screen does not scroll.

NUL <null> (00)
Delays for the time required to write one character. The state of the console does not change.

any printable character
Writes the character, handling the console as described for SP.

Note: The effect of sending non-printable characters other than those described above is not defined.

Output Options. The following set of cursor and screen functions are optional in the sense that almost all major functions of the System will still be available even if they are not provided. The display of the IBM Personal Computer provides them.

The control characters or sequences of characters which provide these functions are left unspecified (these are soft characters). Their definition is stored in the SYSCOM data area (in main memory).

Reverse Line Feed
Moves the cursor to the next line higher on the screen without changing the column or the contents of the screen. If the cursor is already on the top line, the screen is redisplayed with the proper cursor location.

Non-destructive Forward and Backward Space
Moves the cursor in the direction indicated without changing the contents of the screen (hence "non-destructive"). If the cursor is at the beginning or end of a line, it remains where it was.

Cursor HOME
Moves the cursor to the upper left-hand corner of the screen without changing the contents of the screen. This is position (0,0).

Cursor X,Y Positioning
Moves the cursor to some absolutely determined row and column without altering the contents of the screen. X is the column co-ordinate (0..79) and Y the row co-ordinate (0..23). If X or Y is out of range, the cursor is moved to the closest edge of the screen.

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

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

Input Requirements. Input from the console is not echoed to the screen by the BIOS; this function is handled by RSP/IO. Keys that represent ASCII characters generate 8-bit codes between 0 and 127.

Input Options. The console input BIOS also handles the following special functions:

START/STOP. The START/STOP character controls console output (it is <ctrl-S> on the IBM Personal Computer). When START/STOP (a soft character) is received, console output is suspended until (a) another START/STOP character is received, (b) a FLUSH character is received, (c) the console BIOS is reinitialized, or (d) the BREAK character is received. The actions of the last three cases are discussed below. Should another START/STOP character be received, the suspended activities resume exactly as they left off. The chief benefit of this arrangement is that the user can suspend output processes that are proceeding too fast: e.g., 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 queuing of keyboard input continues during the suspension.)

FLUSH. FLUSH is another soft control character (<ctrl-F>). When FLUSH is typed, the console output BIOS discards all output characters (i.e., does not display them) until (a) FLUSH is typed again, (b) input is requested from the console BIOS, (c) the console BIOS is re-initialized or (d) 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 cancelled and FLUSH has its usual effect. This feature is useful when a long textfile is being displayed on the console and you're tired of looking at it. Push 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 (<ctrl-@>) is typed, the console input BIOS immediately gives control to a special Interpreter routine. The vector to this routine is passed at console initialization time. After execution of the BREAK routine, the BIOS continues as before. The BREAK routine is responsible for notifying the Interpreter that a BREAK must be executed before the next P-code is interpreted.

> **Note:** The BREAK character is never returned to the RSP.

Receipt of BREAK should terminate any START/STOP or FLUSH suspension pending. (BREAK is also a soft character.)

Type-Ahead. When non-special characters (i.e., not described in the sections above) are received from the keyboard when no read request is pending, they are queued until the next read request. The next read request removes the first character from the queue. When characters in excess of the maximum queue size are received, they are ignored and the queue remains intact. On the IBM Personal Computer, the queue is at least 40 characters long. The speaker sounds for each character that is typed after the queue is full.

Initialization and Control. The initialization and control part of the console BIOS is responsible for the following tasks:

Soft character recognition
The System stores the soft characters START/STOP, FLUSH and BREAK in a data area called SYSCOM. One parameter to console initialization and control is a pointer to the start of the SYSCOM area. The offsets to these characters from that pointer are (expressed as positive byte offsets):

FLUSH
83 decimal (53 hex and 123 octal)

BREAK
84 decimal (54 hex and 124 octal)

STOP/START
85 decimal (55 hex and 125 octal)

BREAK vector
Another initialization and control parameter is the address of the Interpreter routine which handles BREAK. The console initialization code sets up a vector to this address via its private data area and the BIOS calls this routine when the BREAK character is received.

Flags
Initialization causes the START/STOP and FLUSH flags to be cleared.

Type-ahead queue
Initialization discards any characters currently waiting in the type-ahead queue.

Printer

The printer is conceived as being a line printer or other hardcopy device. In actual practice, any ASCII display may be used.

Output Requirements. In order to serve the widest variety of hardcopy 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. If this is the case, the BIOS must recognize the end of a line. EOLN is signalled by a certain character: the possibilities are listed below:

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

LF <line feed> (hex 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 (i.e., return and line feed) is the only way your printer can print a line, then do so at an LF: don't do anything about a CR.

FF <form feed> (hex 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, i.e., a return followed by a line feed.

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 returns all eight data bits unchanged.

Initialization and Control. Initialization of the printer device makes it ready to print at the begininng of a blank line. Any characters that have been buffered but not printed are lost. The printer does not do a form feed each time it is initialized.

Disk

This discussion is general, and meant to describe the BIOS, not the particulars of the IBM Personal Computer disks.

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

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

Bootstrap Location. While bootstrap schemes vary, typical implementations make use of a hardware (usually ROM) bootstrap to load and execute a primary software bootstrap which, in turn, loads and executes a secondary software bootstrap.

The secondary bootstrap then loads the Interpreter and Operating System, performs required initializations, and starts the System.

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

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

Logical blocks 0 and 1 of each disk are reserved for bootstrap code (a total of 1024 bytes).

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 Physical Sector Addressing Mode.

Output Requirements. The disk device BIOS transfers 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 (i.e., a block is partially written to), the remaining contents of the last block are undefined. This makes it possible to write as much of whatever garbage remains in the buffer, if that is most convenient, to fill up a whole sector. Figure 3-7 illustrates this situation. The language level is responsible for keeping track (in logical block numbers and byte counts) of where the good data is.

Example: **Write to disk.**

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

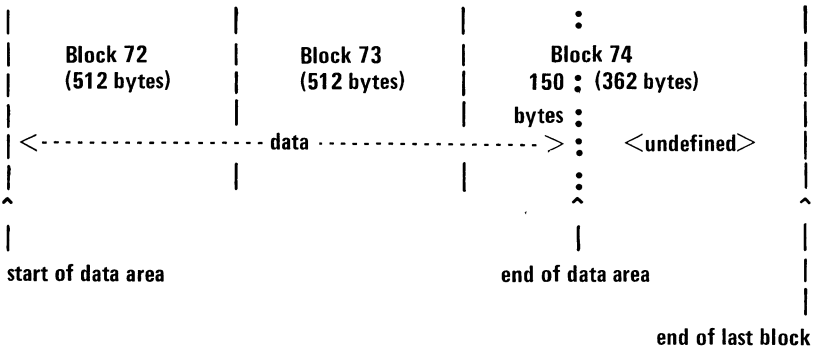


Figure 3-7. State of Blocks on Disk After a Write

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 transfers no more than the number of bytes requested.

Initialization and Control. Initialization of a disk device brings it to a state in which it is ready to read or write from any given track or sector. Any buffered data is lost.

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.

Output Requirements. As noted above, all eight bits of the data byte are transmitted. The remote BIOS driver is sent one byte at a time.

Input Requirements. Input from a remote device is buffered as with the type-ahead queue. As noted above, all eight data bits are returned.

Initialization and Control. Initialization of the remote device brings it to a state in which it is ready to read or write.

Special BIOS Calls

These functions are provided by the BIOS to make configuration-specific functions accessible to the Interpreter. 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 returns a completion code.

System Output

This is reserved for future expansion.

System Input

This is also reserved for future expansion.

System Initialization and Control

The System Initialization and Control BIOS routine initializes the interrupt system.

System Status

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

Word 1 - The address of the last word in accessible contiguous Random Access Memory, e.g., on a system with 64K bytes of Random Access Memory, the last byte address may be FFFF, but the last word address is FFFE.

Word 2 - Equals 0.

Word 3 - Equals 0.

CHAPTER 4. THE OPERATING SYSTEM

Contents

NOTES

Organization

Structured Overview of the System

The IV.0 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

Unit Name	Function
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 Codepool, handle faults, and read segments. The Kernel also contains four subsidiary segments, which are swappable:

GETCMD processes user input at the main command level, and builds a user program's runtime environment;

USERPROG is the reserved segment slot for the user's program (at bootstrap time it contains the Pascal-level code which builds the initial runtime environment for the Operating System);

INITIALIZE is called when the System is booted or re-initialized: it reads SYSTEM.MISCINFO, locates the System codefiles, and sets up the table of devices;

PRINTERROR prints runtime error messages.

The Operating System UNITs are compiled separately. They are bound together in a single codefile, 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 Codepool if the Codepool is internal. The area between the Heap and the Codepool (or the heap and the stack if Codepool is external) 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 deallocate 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 for this particular variant is allocated (which may be less than the largest variant in that record).

VARNEW(P,NWords), where P is a pointer to type T, causes NWords to be allocated on the Heap. T would most commonly be an array. NWords (indirectly) determines how many elements of the array are actually available in this instance. P returns the address of the first location allocated on the Heap.

VARNEW is a function, and returns the number of words that actually were allocated: this should equal NWords; if it is 0, then there was less than NWords of available space, and if it is some other number, something went wrong.

DISPOSE and VARDISPOSE. DISPOSE and VARDISPOSE deallocate 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 deallocated by PERMDISPOSE(P). Even a RELEASE cannot remove it. These routines are meant for System use, and are not user 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.

Heap Implementation

Operating System Interface

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 Codepool if the pool is internal or possibly by the low bound of the stack if the pool is external.

The Heap maintenance variables have the following structure:

```
VAR
  HeapInfo: RECORD
              Lock: semaphore;
              TopMark,
              HeapTop: MemPtr;
    END;
  PermList: MemPtr;
```

The Lock semaphore guarantees that the Heap is modified by only one process at a time. TopMark points to the highest MARK. HeapTop points to the highest allocated space on the Heap. The fault handler uses HeapTop to determine how close the Codepool can be to the Heap. A base value is computed and is either the base of the codepool (if internal) or SP-LOW (if external). PermList points to a linked list of PERMNEW'ed variables. The list is identical in structure to an Avail_List, but each NWords indicates the number of words allocated by a PERMNEW. If PermList is NIL, then no variables have been PERMNEW'ed.

Tactics. In general, a request for Heap space through a MARK, NEW, VARNEW, or PERMNEW causes HeapTop to be set to the new top of the Heap. The fault handler always places the Codepool (located at PoolBase) above Heaptop if the pool is internal. If the pool is external then the stack and the heap are allowed to grow towards each other. If they meet, a STACK OVERFLOW condition exists. Thus, HeapTop reserves space for the Heap as soon as a Heap operator requests it. This is necessary because of possible interactions between Stack fault handling and Heap space allocation.

The Operating System uses the global variable SysCom^.GDirP (global directory pointer) to allocate a disk directory on the Heap. The Operating System's use of this Heap space is meant to be

invisible to the user. Therefore, before any Heap operation (except DISPOSE), SysCom^.GDirP is DISPOSE'd to make the space occupied by the directory available again.

Runtime Environment. Since both the user and the Operating System use the Heap, the Operating System MARK's the Heap immediately before the execution of a user program by the call:

```
MARK (EMPTYHEAP);
```

... after the user program terminates, the Operating System calls:

```
RELEASE (EMPTYHEAP);
```

Thus, all user space is freed after the program terminates, unless space has been allocated by one or more calls to PERMNEW.

MARK (EMPTYHEAP) occurs after the runtime environment for the user program has been built. The program's runtime 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 the user program and any units it USES also appears before EMPTYHEAP. Heap space that follows EMPTYHEAP is intended only for the local use of the user program.

The Heap is shared by all tasks in the System.

The Codepool

The Codepool may reside in main memory between the Stack and the Heap or in memory external to the Stack/Heap space. The behavior of the pool depends on whether it is internal (between Stack and Heap) or external (outside Stack/Heap space). It contains executable code segments that may possibly be discarded, or swapped in from disk again. Thus, the size, the contents and the position of the codepool only changes if it is located between the Stack and the Heap. The flexibility of the Codepool handling can provide a running program with more free memory space than in previous versions previous to IV.0.

A segment in the Codepool 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 Codepool 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 Codepool.

Segments in the Codepool are organized into a doubly-linked list by pointers in each segment's SIB (described in Chapter 1).

The routines that manage the Codepool are in the Operating system's KERNEL unit. They make use of the pointers within the SIB. The following Pascal fragment shows declarations for the Codepool description and the global pointer to that description.

```
TYPE
  CodePool: ^ PoolDes;
    {Points to a description of the Codepool.}

  PoolDes: RECORD
           PoolBase: FullAddress;
             {The first address in the Codepool.
              A 32-bit quantity.}
           PoolSize: Integer;
             {Size of the Codepool in words.
              This value is only referenced
              if the pool is external to the
              Stack/Heap space.}
           MinOffset: Memptr;
             {Byte offset of the lowest
              useable value in the pool.
              If the pool is internal this
              value is dependent on Heaptop;
              otherwise it is dependent only
              on segment alignment
              requirements (if any).}
           MaxOffset: Memptr;
             {Byte offset one
              word passed the segment in
              the highest position in the
              codepool. If the pool is
              internal, it is also equal
              to the SP_LOW value of the
              maintask.}
           Resolution: Integer;
             {Segment alignment
              requirements (machine
              dependent) in bytes.}
           PoolHead: SIB_P;
             {Points to the SIB of the
              segment at the base of the
              Codepool. If Codepool is
              internal, this is the segment
              nearest the Heap.}
           PermSIB: SIB_P;
             {Points to the SIB of the
              segment that is always resident
              in the Codepool.}
```

```
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 Codepool if the
   Codepool is in between the
   Stack and the Heap.}
HeapTop: Mem_Ptr;
  {Points to the top of the Heap.}
Extended: Boolean;
  {True if the Codepool is external
   to the Stack/Heap area.}
```

If the pool is internal when space is requested either for the Heap or the Stack, the Codepool management routines first attempt to reposition the Codepool without swapping out any segments.

The actual bounds of the Codepool are in MinOffset, which points to the low end of the Codepool, and MaxOffset, which points to one word above the top of the Codepool. The Codepool 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 (if pool is internal). MaxOffset is the same as SP_LOW if the Codepool is internal.

The Codepool may be modified by any of the following circumstances (if an only if it resides between Stack/Heap):

1) A Heap fault is detected, and the Codepool is moved up in memory toward the Stack to free the needed number of words for the Heap.

2) A Stack fault is detected, and the Codepool is moved down in memory toward the Heap to free the needed number of words for the Stack.

3) A Heap fault or Stack fault is detected, and the Codepool cannot be moved to allocate the space: one or more segments are swapped out, the remaining segments are moved together, and the Codepool is positioned to allow for the needed Heap or Stack space.

4) 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 re-initialized.

5) A segment fault is detected. The Codepool management routines first try to read the segment in at either end of the Codepool without moving it. If this is impossible and if the pool is internal they attempt to create more room by moving the Codepool toward either the Stack or the Heap, and then read the segment. If this too is impossible or the pool is external, segments are swapped out to make room, and the new segment is then read in. If this last effort also fails, a POOL OVERFLOW is reported, and the System is re-initialized, if the pool is external. A STACK OVERFLOW is reported if the pool is internal.

The Codepool 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 Codepool freely, without fear of causing a Stack fault.

Fault Handling

When memory space is required by the Stack or Heap, or entry into a non-resident segment is attempted, a fault is issued. The Faulthandler process is activated, and uses the Codepool 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 Interpreter (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 .. Heap_Fault;
                    END;

Fault Sem: RECORD
           Real_Sem, Message_Sem: semaphore;
           Message: Fault_Message;
             END;
```

The Interpreter detects only Stack and segment faults. When the Interpreter 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 Codepool, 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 Codepool 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 Codepool 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, the BIOS routines ATTACH, QUIET, and ENABLE, and the Interpreter routine EVENT.

Concurrency support in Version IV.0 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.

ATTACH is a Pascal-level intrinsic that can also be accessed by a CXG 29 instruction. QUIET is called by a CXG 27, and ENABLE by a CXG 28.

QUIET inhibits all Pascal-level events. ENABLE restores the prior event state. Calls to them must be paired, and not nested.

ATTACH may be called to associate a semaphore with an event number. When an event occurs, the BIOS calls the Interpreter routine EVENT. If the event has been ATTACH'ed, EVENT signals the appropriate semaphore. EVENT can be called only after a call to QUIET.

These are the event numbers on the IBM Personal Computer:

0..16	reserved for the System;
17	break;
18	execution error;
19	keysready (used by the print spooler)

Only event 19 is accessible to a user program. Other events are reserved for the System.

I/O Support

FIBs

File I/O is controlled with a structure called a FIB (File Information Block). 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;
            FDev: DevNum;
            FVolID; VolID;
            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 & Wirth) file, an INTERACTIVE file awaiting a character, or an INTERACTIVE file with a character. FRecSize is 0 for untyped files, 1 for INTERACTIVE files and textfiles; 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 block-structured 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 block-structured volumes, except untyped files.

If FSoftBuf is TRUE, then the last set of FIB fields are used: FNxtByte and FMaxByte are used for buffer handling, FBufChngd indicates that the buffer contents have been modified, and FBuffer is the buffer itself.

Directories

Figure 4-1 illustrates the structure of a directory (as on a disk or other block-structured volume):

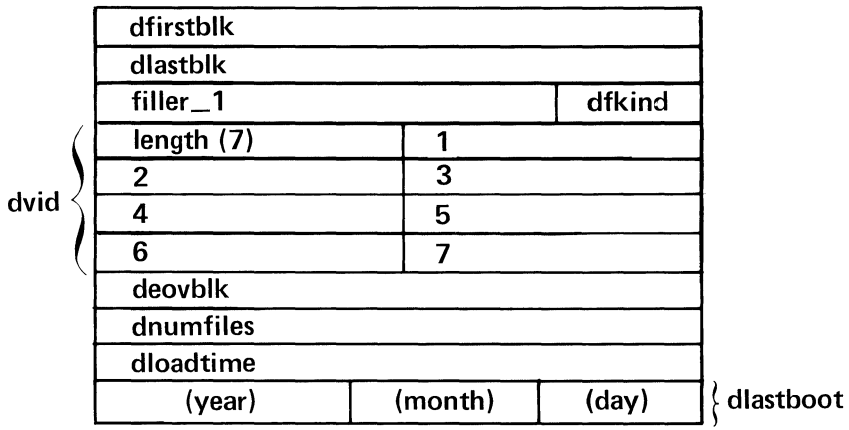
Varieties of I/O

Record I/O
Record I/O applied to typed Pascal files, using the intrinsics GET and PUT.

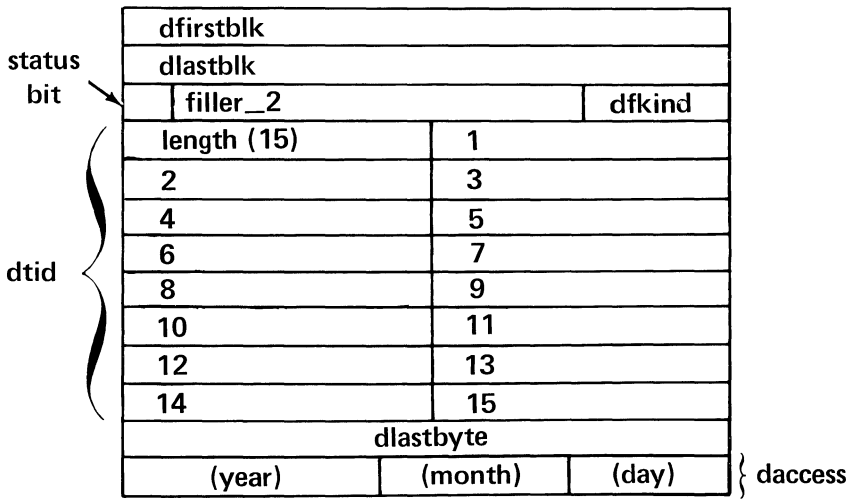
Screen I/O
Screen I/O may be handled by the unit SCREENOPS, whose routines are described in the *UCSD p-System User's* manual.

DIRENTRY RECORD (0)

for dfkind=securedir, untyped file (dir[0])



DIRENTRY RECORD (1-77)



DIRECTORY: array [0..77] of direntry;

Figure 4-1. Directory Format

Input from the screen is accomplished by the procedure CHAR_DEV_GET, which uses SC_CHECK_CHAR (in SCREENOPS) and SYSCOM^.MISCINFO to determine whether any special handling needs to be done.

Output to the screen is accomplished by a simple UNITWRITE.

Block I/O

Block I/O applies to untyped 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 untyped file, all other file formatting is disabled.

Text I/O

A textfile is a file of ASCII characters. It has a 2-block header that contains formatting information used by the Screen Oriented Editor. When a textfile is used by a System program other than the Editor, the Operating System ignores this header. When a new textfile is created, the Operating System writes a 2-block header filled with NULs.

Textfiles always have an even number of blocks. Thus, the smallest possible textfile is 4 blocks long. Any extra space is padded with NULs.

Each record in a textfile is one line of text, terminated by a <return> character. If the first character in a textfile record is a DLE (decimal 16), it is interpreted as a blank-compression code: the following byte is (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).

User programs typically handle textfiles with READ, READLN, WRITE, and WRITELN. GET and PUT may be used, and follow the Jensen & Wirth standard for files of type TEXT.

NOTES

CHAPTER 5. PROGRAM EXECUTION

Contents

NOTES

Runtime Environment

The runtime environment for a user program is created by the Operating System's GETCMD unit. GETCMD starts the execution of System programs such as the Compiler, Linker, Filer, etc., and user programs named in the eX(ecute command. In all such cases, GETCMD calls the procedure ASSOCIATE, which finds the appropriate codefile, and then calls BUILDENV. BUILDENV constructs a program's runtime environment, as outlined in Chapter 1.

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 codefile.

SEGREFs are segment reference assignments emitted by the Compiler. Segment numbers are local to a code segment. The main program is segment 2, and subsidiary segments, if any, are numbered starting from 3. Segment 1 is always the Operating System's KERNEL unit. SEGREFs are emitted for any principal segments used by the compilation (such as a used unit). At associate time, BUILDENV uses the SEGREF list to find the segments that the program uses.

All runtime errors detected by the System cause the current program to halt. The System displays an error message, and when the user types a <space>, the System is reinitialized. The program's runtime 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 re-re-started with the U(ser restart command. The System may or may not need to call BUILDENV again.

APPENDIXES

Contents

Appendixes 2

APPENDIX A. SUMMARY OF BIOS CALLING SEQUENCES

The following is a summary of the calling conventions described in "Calling Mechanisms" in Chapter 3. Specific protocols for the IBM Personal 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

Entry Point	Parameters
REMOTEREAD	single data byte
REMOTEWRITE	single data byte
REMOTECTRL	(none)
REMOTESTAT	STATREC pointer
	CONTROL word
USERREAD	block number
	byte count
	data area address
	device number
	CONTROL word
USERWRITE	(same as USERREAD)
USERCTRL	device number
USERSTAT	device number
	STATREC pointer
	CONTROL word
SYSREAD	block number
	byte count
	data area address
	device number
	CONTROL word
SYSWRITE	(same as SYSREAD)
SYSCTRL	device number
SYSSTAT	STATREC pointer
	CONTROL word

APPENDIX B. IBM PERSONAL COMPUTER BIOS CALLS

Entry Points: All BIOS entry points are given as positive offsets from the beginning of the BIOS code space. These locations contain appropriate addresses of routines within the BIOS.

Parameters: When parameters are not being passed in a specified register, they are pushed on the stack. Offsets from top-of-stack are given, recognizing that the stack grows down.

Completion Code: Return in register AH.

Calling Sequence: The RSP will use the CALL instruction to call the BIOS. Thus the return address is at (SP),(SP)+1. All registers are available for use by the BIOS. The BIOS should clean off the stack before returning to the RSP.

Entry Point	Offset (hex)	Parameters
CONSOLEREAD	00	return data byte in Reg AL
CONSOLEWRITE	02	write data byte in Reg 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

Entry Point	Offset (hex)	Parameters
PRINTERREAD	08	return data byte in Reg AL
PRINTERWRITE	0A	write data byte in Reg AL
PRINTERCTRL	0C	(none)
PRINTERSTAT	0E	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
DISKREAD	10	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 drive number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
DISKWRITE	12	(same as DISKREAD)
DISKCTRL	14	drive number in Reg CL
DISKSTAT	16	drive number in Reg CL STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
REMOTEREAD	18	return data byte in Reg AL
REMOTEWRITE	1A	write data byte in Reg AL
REMOTECTRL	1C	(none)
REMOTESTAT	1E	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5

Entry Point	Offset (hex)	Parameters
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)+A,(SP)+B
USERWRITE	22	(same as USERREAD)
USERCTRL	24	device number in Reg CL
USERSTAT	26	device number in Reg CL STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
SYSREAD	28	programmed halt (see "System Input" in Chapter 3)
SYSWRITE	3F	programmed halt (see "System Output" in Chapter 3)
SYSCTRL	2C	(none)
SYSSTAT	2E	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5

APPENDIX C. P-CODES

SLDC	0..31	Short Load Word Constant
LDCN	152	Load Constant NIL
LDCB	128	Load Constant Byte
LDCI	129	Load Constant Word
LCO	130	Load Constant Offset
SLDL1	32	Short Load Local Word
...	...	
SLDL16	47	
LDL	135	Load Local Word
SLLA1	96	Short Load Local Address
...	...	
SLLA8	103	
LLA	132	Load Local Address
SSTL1	104	Short Store Local Word
...	...	
SSTL8	111	
STL	164	Store Local Word
SLDO1	48	Short Load Global Word
...	...	
SLDO16	63	
LDO	133	Load Global Word
LAO	134	Load Global Address
SRO	165	Store Global Word

SLOD1	173	Short Load Intermediate Word
SLOD2	174	
LOD	137	Load Intermediate Word
LDA	136	Load Intermediate Address
STR	166	Store Intermediate Word
LDE	154	Load Extended Word
LAE	155	Load Extended Address
STE	217	Store Extended Word
SIND0	120	Short Index and Load Word
...	...	
SIND7	127	
IND	230	Index and Load Word
STO	196	Store Indirect
LDC	131	Load Multiple Word Constant
LDM	208	Load Multiple Words
STM	142	Store Multiple Words
LDCRL	242	Load Real Constant
LDRD	243	Load Real
STRL	244	Store Real
CAP	171	Copy Array Parameter
CSP	172	Copy String Parameter
LDB	167	Load Byte
STB	200	Store Byte
LDP	201	Load a Packed Field
STP	202	Store into a Packed Field
MOV	197	Move
INC	231	Increment Field Pointer
IXA	215	Index Array
IXP	216	Index Packed Array

LAND	161	Logical And
LOR	160	Logical Or
LNOT	229	Logical Not
BNOT	159	Boolean Not
LEUSW	180	Less Than or Equal Unsigned
GEUSW	181	Greater Than or Equal Unsigned
ABI	224	Absolute Value Integer
NGI	225	Negate Integer
INCI	237	Increment Integer
DECI	238	Decrement Integer
ADI	162	Add Integers
SBI	163	Subtract Integers
MPI	140	Multiply Integers
DVI	141	Divide Integers
MODI	143	Modulo Integers
CHK	203	Check Subrange Bounds
EQUI	176	Equal Integer
NEQI	177	Not Equal Integer
LEQI	178	Less Than or Equal Integer
GEQI	179	Greater Than or Equal Integer
FLT	204	Float Top-of-Stack
TNC	190	Truncate Real
RND	191	Round Real
ABR	227	Absolute Value of Real
NGR	228	Negate Real
ADR	192	Add Reals
SBR	193	Subtract Reals
MPR	194	Multiply Reals
DVR	195	Divide Reals
EQREAL	205	Equal Real
LEREAL	206	Less Than or Equal Real
GEREAL	207	Greater Than or Equal Real

ADJ	199	Adjust Set
SRS	188	Build a Subrange Set
INN	218	Set Membership
UNI	219	Set Union
INT	220	Set Intersection
DIF	221	Set Difference
EQPWR	182	Equal Set
LEPWR	183	Less Than or Equal Set
GEPWR	184	Greater Than or Equal Set
EQBYT	185	Equal Byte Array
LEBYT	186	Less Than or Equal Byte Array
GEBYT	187	Greater Than or Equal Byte Array
UJP	138	Unconditional Jump
FJP	212	False Jump
TJP	241	True Jump
EFJ	210	Equal False Jump
NFJ	211	Not Equal False Jump
JPL	139	Unconditional Long Jump
FJPL	213	False Long Jump
XJP	214	Case Jump
CPL	144	Call Local Procedure
CPG	145	Call Global Procedure
SCPI1	239	Short Call Intermediate Procedure
SCPI2	240	
CPI	146	Call Intermediate Procedure
CXL	147	Call Local External Procedure
SCXG1	112	Short Call External Global Procedure
...	...	
SCXG8	119	

CXG	148	Call Global External Procedure
CXI	149	Call Intermediate External Procedure
CPF	151	Call Formal Procedure
RPU	150	Return from Procedure
LSL	153	Load Static Link
BPT	158	Breakpoint
SIGNAL	222	Signal
WAIT	223	Wait
EQSTR	232	Equal String
LESTR	233	Less Than or Equal String
GESTR	234	Greater Than or Equal String
ASTR	235	Assign String
CSTR	236	Check String Index
LPR	157	Load Processor Register
SPR	209	Store Processor Register
DUP1	226	Duplicate One Word
DUPR	198	Duplicate Real
SWAP	189	Swap
NOP	156	No Operation
NAT	168	Native Code
NAT-INFO	169	Native Code Information
RESERVE1	250	reserved
...	...	
RESERVE6	255	

APPENDIX D. AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

Glossary

This is intended as an aid to readers who are unfamiliar with many "buzz words" used in this document, and is not meant to be either comprehensive or precise.

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, i.e. at the same time. Synonymous with multitasking.

Dynamic: Information which changes during program execution (or is not known before runtime).

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.

Inter-Segment: 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, i.e. at the same time. A task is a PROCESS from the user's 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 Interpreter implements a "pseudo-machine."

Postprocessor: A program which is executed after the completion of some other program, and uses as input the output of that previous program. A postprocessor that creates output which can be used by still another program is often called a "filter".

Principal Segment: A segment that has a segment reference list, i.e., 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.

Recursion: The continued repeating of the same operation or group of operations.

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.

Runtime: That part of a program's lifetime in which it is being executed (or "run").

Self-Modifying: Code which overwrites or modifies itself during execution, thus changing its meaning. This 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 which does not change throughout program execution (it is known before runtime).

Subsidiary Segment: A segment that has no segment reference list, i.e., 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 once 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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Personal Computer
Computer Language Series

Product Comment Form

Internal Architecture Guide

6936557

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

Name _____

Address _____

City _____ State _____

Zip Code _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

ntinued from inside front cover

ME STATES DO NOT ALLOW THE CLUSION OF IMPLIED ARRANTIES, SO THE ABOVE CLUSION MAY NOT APPLY TO)U. THIS WARRANTY GIVES YOU ECIFIC LEGAL RIGHTS AND YOU AY ALSO HAVE OTHER RIGHTS HICH VARY FROM STATE TO ATE.

M does not warrant that the functions ntained in the program will meet your uirements or that the operation of the)gram will be uninterrupted or error e.

wever, IBM warrants the diskette(s) or sette(s) on which the program is fur- hed, to be free from defects in materials d workmanship under normal use for a iod of ninety (90) days from the date of ivery to you as evidenced by a copy of ır receipt.

MITATIONS OF REMEDIES

M's entire liability and your exclusive nedy shall be:

the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or

if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

NO EVENT WILL IBM BE LIABLE) YOU FOR ANY DAMAGES, CLUDING ANY LOST PROFITS,)ST SAVINGS OR OTHER CIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328–W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

6936557

Printed in United States of America