PREFACE

This addendum contains additions and corrections to the *Users's Manual Supplement, Version IV.*

Chapter 9, The 8086 and 68000 Assemblers, on pages 1 through 7 of this addendum is a new chapter.

Pages 9 and 10 of this addendum, which contain corrected information concerning the Segment Alignment setting for the Z80 and 8080, the QUIET and ENABLE commands, and the EVENT I/O interface, replace pages 1 and 2 of the *Users' Manual Supplement.*

Page 11 of this addendum, which contains a corrected directory list of the current Z80 release disks, replaces page 3 of the *Users's Manual Supplement.*

Page 12 of this addendum, which contains a corrected directory list of the current 8086 release disks, replaces page 16 of the *Users' Manual Supplement.*
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. The 8086 and 68000 Assemblers</td>
<td>1</td>
</tr>
<tr>
<td>The 8086 Assembler</td>
<td>1</td>
</tr>
<tr>
<td>Notational Conventions</td>
<td>1</td>
</tr>
<tr>
<td>8086/88/87 Assembler Error Messages</td>
<td>4</td>
</tr>
<tr>
<td>68000 Assembler</td>
<td>5</td>
</tr>
<tr>
<td>Syntax Conventions</td>
<td>5</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>6</td>
</tr>
<tr>
<td>8086 Error Messages</td>
<td>7</td>
</tr>
<tr>
<td>1. New Product Package</td>
<td>9</td>
</tr>
<tr>
<td>Z80 and 8080</td>
<td>10</td>
</tr>
<tr>
<td>QUIET and ENABLE</td>
<td>10</td>
</tr>
<tr>
<td>EVENT</td>
<td>10</td>
</tr>
<tr>
<td>Z80 Adaptable System Files</td>
<td>11</td>
</tr>
<tr>
<td>8086 Adaptable System Files</td>
<td>12</td>
</tr>
</tbody>
</table>
9. THE 8086 AND 68000 ASSEMBLERS

This section contains information specific to the 8086 and 68000 versions of the UCSD p-System adaptable assembler.

The 8086 Assembler

The UCSD p-System 8086/88/87 Assembler differs in some respects from the standard Intel assembler. These differences are listed in this section. Also, the 8086/88/87 specific assembler errors are listed.

Notational Conventions

Assembler Directives. None of the Intel assembler directives or operators are implemented. Instead, the assembler directives described in the Users' Manual are available.

Parenthesis. Index or base register references in a memory operand are enclosed in parentheses, not square brackets, e.g., FIRST(BX) rather than FIRST[BX].

Immediate Byte. ADD immediate byte to memory operand is coded

    ADDBIM memop,immedbyte

to distinguish it from the ADD memop, immedword which is the default. Similarly, MOVBIM, ADDBIM, SUBBIM, SBBBIM, CMPPBIM, ANDBIM, ORBIM, XORBIM, and TESTBIM are added to the vocabulary.

Memory Byte. INC memory byte is coded:

    INCMB memop

to distinguish it from INC memory word, which is the default. Similarly, DECBMB, MULBMB, IMULMB, DIVMB, IDIVMB, NOTBMB, NEGMB, ROLOBMB, RORBMB, RCLMB, RCRMB, SALMB, SHLMB, SHRMB, SARMB were added to the vocabulary to specify memory byte operands.

MUL and DIV Byte. In MUL, IMUL, DIV, IDIV the single memory operand form, e.g.,

    MUL memop

implies a word operation. To specify a byte operation, either MULMB memop may be used, or the form
MUL AL, memop

The same holds true for IMUL, DIV, IDIV. (Note DIV AL, memop is rather misleading, as the actual operation would be AX/memory-byte.)

MOV Substitute for LEA. For LEA reg,label or LEA reg,label+const the assembler will substitute MOV reg,immed_val where immed_val = label or label+const. This saves four clock times (4 vs. 8).

IN and OUT. The normal form of IN and OUT is IN AC,port or IN AC,DX and OUT port,AC or OUT DX,AC where AC=AL denotes an 8-bit data path and AC=AX denotes a 16-bit path. Since the accumulator is the only possible register source/destination (DX specifies port=address in DX), single operand forms are also provided: INB and OUTB for byte data, and INW and OUTW for 16-bit data. The syntax is INB port or OUTB DX.

In the two-operand forms of IN and OUT, the order of the operands is not important; thus OUT AC,DX or OUT DX,AC, port will be acceptable.

String Operations. The mnemonics for the string operations are suffixed with B or W to denote byte or word operations: thus MOVS B and MOVS W, CMPS B and CMPS W, SCAS B and SCAS W, LODSB and LODSW, and STOSB and STOSW are in the vocabulary, but MOVs ... STOS are not.

Segment Override. XLAT and the string instructions have implied memory operands and nothing is required to be coded in the operand field. However, in order to permit the specification of a segment override prefix in the case of XLAT, MOVS/MOVSW, CMPSB/CMPSW, and LODSB/LODSW, the assembler permits operand expressions for these instructions.

Note, however, that only the default segment for SI, namely DS, can be overridden. The segment for DI is ES and cannot be overridden. A segment override prefix of DS applied to SI does not generate a segment override prefix.

If these operations were written with operands, they would have this syntax:

```
XLAT AL,(BX)
MOVS[B|W] (DI),[seg:](SI)
CMPS[B|W] (DI),[seg:](SI)
SCAS[B|W] (DI),AX
LODS[B|W] AX,[seg:](SI)
STOS[B|W] (DI),AX
```

The string instructions may be prefixed by a REP (repeat) instruction of some
type. The assembler flags an error if both REP and a segment override are specified.

In addition to the forms DS:memop, etc., a separate mnemonic SEG followed by a segment register name may be written in a statement preceding the instruction mnemonic.

Examples:

MOV AX,ES:VALUE

is equivalent to

SEG ES MOV AX,VALUE

Long Jumps, Calls, and Returns. Intersegment CALL, RET, and JMP are implemented as follows:

a. The mnemonics CALL, RETL, and JMPL specifically designate intersegment operations.

b. An indirect address (e.g., (reg) or (label)) is assembled in standard fashion with a "mod op r/m" effective address byte possibly followed by displacement bytes. The memory location referenced must hold the new IP, and the next higher location must hold the new CS.

c. The direct address form must have two absolute operands:

CALLL expr1,expr2

where expr1 is the new IP and expr2 becomes the new CS. Constants or external symbols (e.g., .REF definitions) qualify as absolute operands.

8087 Mnemonics. Mnemonics for the 8087 floating point operations are standard except for certain of the memory reference operations, where a letter suffix is appended to denote the operand size:

D short real or short integer (double word)

Q long real or long integer (quad word)

W integer word
Addendum
Users' Manual Supplement
Version IV

_T temporary real (ten byte)

The D and Q suffixes apply to the following real ops:

FADD, FCOM, FCOMP, FDIV, FDIVR, FMUL, FST, FSUB, FSUBR, FLD, FSTP

e.g., FADDD, FADDDQ, etc.

The T suffix applies only to FLD and FSTP.

The W and D suffixes apply to the following integer ops:

FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FIMUL, FIST, FISUB, FISUBR, FILD, FISTP

The Q suffix for long integers applies only to FILD and FISTP.

8086/88/87 Assembler Error Messages

The following error messages are specific to the 8086/88/87 Assembler:

76: Had label, open parenthesis then illegality
77: Expected absolute expression
78: Both operands cannot be a segment register
79: Illegal pair of index registers
80: Have to use BX,BP,SI or DI
81: Illegal constant as first operand
82: The first operand is needed
83: The second operand is needed
84: Expected comma before 2nd operand
85: Registers stand alone except in indirect
86: Only 2 registers per operand
87: Expected label or absolute
88: Illegal to use BP indirect alone
89: Close parenthesis expected
90: Cannot POP CS
91: Cannot have exchange of r8 with r16
92: Segment registers not allowed
93: ESC external op on left must be const < 64
94: Only one of the operands can have segment override
95: Right operand must be a memory location
96: Left operand must be a 16-bit register
97: Left operand must be memory or register alone
98: Op cannot be a segment register or immediate
99: Count must be 1 or in CL
100: A byte constant operand is required
101: Operand must use () or be a label
102: LOCK followed by something illegal
103: REP precedes only string operations
104: Not implemented
105: Expected a label
106:
107: Open parenthesis expected
108: Expected register alone as right operand
109: Segovpre then regalone, thats illegal
110: Only one operand allowed
111: Operands are AL,op2 for byte MUL, etc.
112: SP can only be used with the SS segment
113: MOVIM only for immediate to memory
114: BIMs must be immediate bytes to memory
115: Seg override on repeated instruction not ok
116: Segment register expected
117: (8087) Invalid two-operand format
118: (8087) Invalid single operand format
119: (8087) Improper operand field
120: (8087) Instruction has no operands
121: No override of ES on string destination
122: Interseg needs 2 constant or external operands
123: I/O port must be immediate byte or DX
124: I/O source/dest register must be AL or AX

68000 Assembler

Syntax Conventions

The 68000 Assembler follows Motorola standard syntax for opcode fields, register names and address modes. The following list points out some restrictions of the p-System Adaptable Assembler.

Absolute addresses in the range 0000H..7FFFFH generate the absolute short address mode. Absolute short addresses in the range F8000H..FFFFFH cannot be generated.

Absolute addresses in the range 8000H..FFFFFH and external references generate the absolute long address mode. Absolute addresses above FFFFFH cannot be generated.

Any address which can be made PC-relative generates the PC-relative
address mode.

Absolute immediates above FFFFH cannot be generated.

Opcodes which have an optional suffix of A,I,M,Q or X must contain that suffix explicitly.

Length qualifiers (.B, .W or .L) must be specified explicitly in those instructions which have a choice of length. All other instructions must not contain a length qualifier.

The following instructions MUST contain a length qualifier:

ADD, ADDA, ADDI, ADDQ, ADDX, AND, ANDI, ASL (register), ASR (register), CLR, CMP, CMPA, CMPI, CMPM, EOR, EORI, EXT, LSL (register), LSR (register), MOVE (except special forms), MOVEA, MOVEM, MOVEP, NEG, NEGX, NOT, OR, ORI, ROL (register), ROR (register), ROXL (register), ROXR (register), SUB, SUBA, SUBI, SUBQ, SUBX, TST

The following instructions must NOT contain a length qualifier:

ABCD, ASL (memory), ASR (memory), BCHG, BCLR, BSET, BTST, CHK, D8cc, DIVS, DIVU, EXG, JMP, JSR, LEA, LINK, LSL (memory), LSR (memory), MOVE to CCR, MOVE to SR, MOVE from SR, MOVE USP, MOVEQ, MULS, MULU, NBCD, NOP, PEA, RESET, RCL (memory), ROR (memory), ROXL (memory) ROXR (memory), RTE, RSR, RTS, SBCD, Sc, STOP, SWAP, TAS, TRAP, TRAPV, UNLK

The following instructions may contain an optional length qualifier of .S (generate short forward branch):

Bcc, BRA, BSR

Miscellaneous

The 68000 processor is byte-addressed and word-oriented. The byte sex is most-significant byte first.

The default constant radix is decimal, and the default list radix is hexadecimal.
68000 Error Messages

The following error messages are specific to the 68000 Assembler:

76: unrecognizable address mode
77: address register expected
78: close paren ')' expected
79: displacement out of range
80: index register expected
81: illegal length qualifier
82: illegal source address mode
83: illegal destination address mode
84: comma ',' expected
85: length qualifier required
86: length qualifier not allowed
87: data register expected
88: label expected
89: illegal register list
90: immediate operand expected
1. NEW PRODUCT PACKAGE

The release disks for the current release of the p-System are formatted into virtual floppies, as with previous Adaptable Systems. However, there are now two (rather than three) virtual floppies per disk. Each virtual floppy contains 240 blocks. The following diagram illustrates the disk images of the virtual floppies:

```
Track:  
0  1  37 38 39  75 76  
| B | 240 Block Virtual Floppy | B | 240 Block Virtual Floppy | U |
| o  | Disk Volume One | o | Disk Volume Two | n |
| r  |                     | r |                     | u |
| a  |                     | a |                     | s |
| c  |                     | c |                     | e |
| k  |                     | k |                     | d |
```

The p-System can be configured to use no real numbers, 2-word real numbers, or 4-word real numbers. For each p-System shipped, there are three Interpreters: one for each of these configurations. There are also two Operating Systems: one for 2-word and one for 4-word reals.

For whichever language(s) you use, you are shipped two versions of the compiler: one for 2-word and one for 4-word reals, as above. If the language is BASIC or FORTRAN, there are also two versions of the runtime library, one for 2-word and one for 4-word reals.

For the Adaptable System, you must supply the FULL Extended SBIOS. The printer, remote, and clock routines may simply be stubs, unless you intend to use those features, but the routines QUIET and ENABLE must be implemented. These are new SBIOS routines, and are described below.

EVENT is a new BIOS routine that may be called from the SBIOS. You must implement a keys-ready event if you wish to use print spooling (described in Chapter 9). EVENT itself is described below.

The actual catalogs of files shipped with each p-System are shown later in the section that corresponds to your processor.

For this release, extended memory is supported on the 8086, but not on the Z80 and 8080. However, one of the SETUP parameters in the section that discusses extended memory does apply to 8080 and Z80 systems. The Segment Alignment
Addendum
Users's Manual Supplement
Version IV

for the Z80 and the 8080 must be set to 2.

Z80 and 8080

This release of the p-System on the Z80 or 8080 may be brought up as described in the Version IV.0 Installation Guide. The only differences are in the format of virtual floppies on each release disk, the distinction between software components for 2-word and 4-word reals as described above, and the new SBIOS and BIOS routines described below.

QUIET and ENABLE

QUIET must disable any P-machine 'events' from occurring. The simplest way to do this is simply to disable all processor interrupts. If your hardware configuration does not allow you to do this, you must devise some other scheme for disabling interrupts.

In the SBIOS jump vector, the offset of QUIET is 54 (hex).

ENABLE allows P-machine events to occur. This may be done by simply re-enabling processor interrupts, or by a scheme that corresponds to the one used by QUIET.

In the SBIOS jump vector, the offset of ENABLE is 57 (hex).

EVENT

EVENT is a BIOS routine that may be called from the SBIOS. Its jump vector offset is 06 (hex).

When an SBIOS routine detects a hardware interrupt (such as a key pressed on the console's keyboard), it may call EVENT with an appropriate event number. This event number may be associated with a semaphore in a high-level language (in UCSD Pascal, this is accomplished by the attach intrinsic).

The events that a user may choose to signal, and what to do with them, are entirely up to the user. The event numbers 0..31 are reserved for the p-System's use, and the event numbers 32..63 are available for user definition.

However, if the user wishes to use print spooling, the SBIOS must call EVENT with an event number of 19 whenever a key is pressed on the console.
Z80 Adaptable System Files

Here are the directories of the current release disks for the Z80. Note that there is no special disk for utilities; utility programs are shipped on the second image of the SYSTEM disk itself.

The ADAPZ disk:

SYSADP:
SYSTEM,INTERP 26 23-Nov-81 6 512 Datafile
SYSTEM,LIBRARY 1 24-Nov-81 72 194 Datafile
SYSTEM,PASCAL 108 24-Nov-81 73 512 Datafile
SYSTEM,FRZER 32 25-Jul-81 141 512 Datafile
SYSTEM,LIBRARY 28 15-Jul-81 173 512 Datafile
SYSTEM,LIBRARY 9 7-Jan-81 201 512 Datafile
< UNUSED > 30 210
6/6 files listed/in-dir>, 210 blocks used, 30 unused, 30 in largest

SYSADP:
SYSTEM,INTERP 26 23-Nov-81 6 512 Datafile
SYSTEM,LIBRARY 1 24-Nov-81 72 194 Datafile
SYSTEM,PASCAL 108 24-Nov-81 73 512 Datafile
SYSTEM,FRZER 32 25-Jul-81 141 512 Datafile
SYSTEM,LIBRARY 28 15-Jul-81 173 512 Datafile
SYSTEM,LIBRARY 9 7-Jan-81 201 512 Datafile
< UNUSED > 30 210
6/6 files listed/in-dir>, 210 blocks used, 30 unused, 30 in largest

The SYSTEM disk:

STSS:
SYSTEM,EDITOR 49 28-Jul-81 6 512 Codefile
SYSTEM,SYSLIB 14 4-Dec-80 55 512 Datafile
DEBUGGER,CODE 29 22-Jul-81 69 512 Codefile
PATCH,CODE 34 3-Nov-81 98 512 Codefile
DISKSIZE,CODE 3 5-Dec-80 32 512 Codefile
MARKDUPR.COE 9 3-Dec-80 140 512 Codefile
< UNUSED > 91 149
6/6 files listed/in-dir>, 149 blocks used, 91 unused, 91 in largest

ST SS:
TAILS,CODE 12 2-Dec-80 6 512 Codefile
BOOTER,CODE 3 4-Dec-80 18 512 Codefile
DISKSIZE,CODE 3 4-Dec-80 21 512 Codefile
SAMPLEGOTO.EXE 4 17-Nov-78 24 512 Textfile
DISKSIZE,CODE 29 5-Mar-81 29 512 Codefile
COPYDUPR,CODE 2 2-Dec-80 36 512 Codefile
MARKDUPR,CODE 4 2-Dec-80 59 512 Codefile
EXEC,CODE 29 5-Dec-80 65 512 Codefile
RECODE,CODE 8 5-Dec-80 92 512 Codefile
REALOPS.A,CODE 11 21-Aug-81 100 512 Codefile
REALOPS.ZZ,CODE 9 24-Nov-81 111 512 Codefile
EXEC,CODE 65 11-Sep-81 120 512 Codefile
COMMAND,CODE 6 20-Jul-81 185 512 Codefile
SCREENPS,CODE 13 29-Jun-81 191 512 Codefile
LIBRARY,CODE 13 5-Nov-81 204 512 Codefile
ABSWRITE,CODE 4 3-Dec-80 217 512 Codefile
< UNUSED > 19 221
16/16 files listed/in-dir>, 221 blocks used, 19 unused, 19 in largest
8086 Adaptable System Files

Here are the current release disks for the 8086 with extended memory. These are the only two disks required for the 8086 system. Note that utility programs are shipped on the second image of the SYSTEM disk.

The SYSTEM disk:

```
SST1:
SISTEM.BDLOAD  49 28-Jul-81  6 512 Codefile
DIAGNOSTIC.CODE  29 22-Jul-81  95 512 Codefile
PATCH.CODE  34 3-Nov-81  84 512 Codefile
SISTEM.STTAX  14 4-Dec-80 118 512 Datafile
DISKCHECK.CODE  8 5-Dec-80 132 512 Codefile
TEMPORARY.CODES  9 3-Dec-80 140 512 Codefile
< UNUSED >  91  149
6/6 files listed in dir, 149 blocks used, 91 unused, 91 in largest
```

```
SST2:
ASMB86.CODE  60 29-May-81  6 512 Codefile
S8686U.CODES  5 5-May-81  66 52 Datafile
ASMB86.SRC  11 2-May-81  71 52 Datafile
S887.POPS  3 4-Feb-81  52 512 Datafile
YADEN.CODE  12 2-Dec-80 95 512 Codefile
BOOTC.FILE  3 4-Dec-80 97 512 Datafile
DSTKFILE.CODE  3 3-Dec-80 100 512 Codefile
SAMPLE.TXT  4 17-Nov-79 103 512 Textfile
SDECODE.CODE  1 5-Oct-81  67 512 Codefile
SOUTHPUIN.CODE  3 2-Dec-80 135 512 Codefile
MARKUP.CODE  4 2-Dec-80 198 512 Codefile
S88P.CODE  29 3-Dec-80 142 512 Codefile
RECOVER.C.CODE  8 5-Dec-80 171 512 Codefile
SCREENTEST.CODE  1 4-Jun-81 179 512 Codefile
LIBRARY.CODE  13 6-Nov-81 192 512 Codefile
< UNUSED >  35  205
15/15 files listed in dir, 205 blocks used, 35 unused, 35 in largest
```

The INTERP disk:

```
S8ST3:
SISTEM.INTERP  19 15-Dec-81  6 512 Datafile
SISTEM.MCINTSO  25 24-Nov-81 26 194 Datafile
SISTEM.PASCAL 108 24-Nov-81 26 512 Datafile
SISTEM.FILER  12 28-Jul-81 174 512 Codefile
SABT.CODE  28 5-Oct-81 166 512 Codefile
SISTEM.LIBRARY  7 8-Feb-82  94 512 Codefile
SISTEM.LINKER  26 27-Jan-81 201 512 Codefile
< UNUSED >  13  227
7/7 files listed in dir, 227 blocks used, 13 unused, 13 in largest
```

```
S8INT:
COMPRESS.CODE  0 7-Dec-80  6 512 Codefile
INTERP.CODE  20 6-Oct-81 16 512 Codefile
INTERP2.C.CODE  23 6-Oct-81 36 512 Codefile
INTERP4.C.CODE  24 6-Oct-81 59 512 Codefile
HOST.CODE  6 18-Sep-81 85 512 Codefile
BIOS.C.CODE  6 26-Sep-81 89 512 Codefile
BIOS.G.CODE  6 26-Sep-81 95 512 Codefile
BIOS.CRP.CODE  6 26-Sep-81 107 512 Codefile
TERTBOOT.CODE  7 17-Sep-81 113 512 Codefile
SBIOS.PS.S.CODE  0 21-Aug-81 141 512 Codefile
SBIOS.PS.CODE  11 21-Aug-81 130 512 Codefile
KERNEL.CODE  65 11-Sep-81 141 512 Codefile
COMMANDS.CODE  6 20-Jul-81 206 512 Codefile
SCREEN.CODE  13 29-Jun-81 212 512 Codefile
< UNUSED >  15  225
15/15 files listed in dir, 225 blocks used, 15 unused, 15 in largest
```
UCSD p-System

USERS' MANUAL SUPPLEMENT

Version IV

January 1982

SofTech Microsystems, Inc.
San Diego, California
XenoFile is a trademark of SofTech Microsystems, Inc.

UCSD, UCSD Pascal, and UCSD p-System are all trademarks of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only, and any unauthorized use is contrary to the laws of the State of California.

CP/M is a registered trademark of Digital Research, Inc.

Copyright © 1982 by SofTech Microsystems, Inc.

All rights reserved. No part of this work may be reproduced in any form or by any means or used to make a derivative work (such as a translation, transformation, or adaptation) without the permission in writing of SofTech Microsystems, Inc.
# Table of Contents

1. NEW PRODUCT PACKAGE ........................................ 1
   1  Z80 and 8080 ........................................... 1
   2  CP/M Adaptable System ................................... 5
   3  8086 ................................................... 6

2. SYMBOLIC DEBUGGER ......................................... 17

3. EXTENDED MEMORY ........................................... 25

4. NATIVE CODE GENERATION .................................... 29

5. PRINT SPOOLING ............................................... 31

6. INTERRUPT HANDLING ......................................... 33

7. XENOFILE ..................................................... 35

8. TURTLEGRAPHICS ............................................ 75
1. NEW PRODUCT PACKAGE

The release disks for the current release of the p-System are formatted into virtual floppies, as with previous Adaptable Systems. However, there are now two (rather than three) virtual floppies per disk. Each virtual floppy contains 240 blocks. The following diagram illustrates the disk images of the virtual floppies:

Track:
0 1 37 38 39 75 76

<table>
<thead>
<tr>
<th>0 T</th>
<th>240 Block Virtual Floppy</th>
</tr>
</thead>
<tbody>
<tr>
<td>o r</td>
<td>Disk Volume One</td>
</tr>
<tr>
<td>t a</td>
<td>c k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0 T</th>
<th>240 Block Virtual Floppy</th>
</tr>
</thead>
<tbody>
<tr>
<td>o r</td>
<td>Disk Volume Two</td>
</tr>
<tr>
<td>t a</td>
<td>c k</td>
</tr>
</tbody>
</table>

The p-System can be configured to use no real numbers, 2-word real numbers, or 4-word real numbers. For each p-System shipped, there are three Interpreters: one for each of these configurations. There are also two Operating Systems: one for 2-word and one for 4-word reals.

For whichever language(s) you use, you are shipped two versions of the compiler: one for 2-word and one for 4-word reals, as above. If the language is BASIC or FORTRAN, there are also two versions of the runtime library, one for 2-word and one for 4-word reals.

For the Adaptable System, you must supply the FULL Extended SBIOS. The printer, remote, and clock routines may simply be stubs, unless you intend to use those features, but the routines QUIET and ENABLE must be implemented. These are new SBIOS routines, and are described below.

EVENT is a new BIOS routine that may be called from the SBIOS. You must implement a keys-ready event if you wish to use print spooling (described in Chapter 5). EVENT itself is described below.

The actual catalogs of files shipped with each p-System are shown later in the section that corresponds to your processor.

1. Z80 and 8080

This release of the p-System on the Z80 or 8080 may be brought up as described in the Version IV.0 Installation Guide. The only differences are in the format of virtual floppies on each release disk, the distinction between software components for 2-word and 4-word reals as described above, and the new SBIOS and BIOS routines described below.
QUIET and ENABLE

QUIET must disable any P-machine 'events' from occurring. The simplest way to do this is simply to disable all processor interrupts. If your hardware configuration does not allow you to do this, you must devise some other scheme for disabling interrupts.

In the SBIOS jump vector, the offset of QUIET is 38 (hex).

ENABLE allows P-machine events to occur. This may be done by simply re-enabling processor interrupts, or by a scheme that corresponds to the one used by QUIET.

In the SBIOS jump vector, the offset of ENABLE is 3A (hex).

EVENT

EVENT is a BIOS routine that may be called from the SBIOS. Its jump vector offset is 04 (hex).

When an SBIOS routine detects a hardware interrupt (such as a key pressed on the console's keyboard), it may call EVENT with an appropriate event number. This event number may be associated with a semaphore in a high-level language (in UCSD Pascal, this is accomplished by the attach intrinsic).

The events that a user may choose to signal, and what to do with them, are entirely up to the user. The event numbers 0..31 are reserved for the p-System's use, and the event numbers 32..63 are available for user definition.

However, if the user wishes to use print spooling, the SBIOS must call EVENT with an event number of 19 whenever a key is pressed on the console. See Chapter 6 for more details.
Adaptable System Files

Here are the directories of the current release disks for the Z80. Note that there is no special disk for utilities: utility programs are shipped on the second image of the SYSTEM disk itself.

The ADAPZ disk:

SYSZ808:
- SYSTEM.INTERP 26 23-Nov-81 6 512 Datafile
- SYSTEM.MISCINFO 1 24-Nov-81 32 194 Datafile
- SYSTEM.PASCAL 108 24-Nov-81 33 512 Datafile
- SYSTEM.FILER 32 28-Jul-81 141 512 Codefile
- SETUP.CODE 28 15-Jul-81 173 512 Codefile
- SYSTEM.LIBRARY 9 7-Jan-81 201 512 Datafile

< UNUSED >
- 30 210

6/6 files<listed/in-dir>, 210 blocks used, 30 unused, 30 in largest

SYSZ80D:
- SYSTEM.INTERP 26 23-Nov-81 6 512 Datafile
- SYSTEM.MISCINFO 1 24-Nov-81 32 194 Datafile
- SYSTEM.PASCAL 108 24-Nov-81 33 512 Datafile
- SYSTEM.FILER 32 28-Jul-81 141 512 Codefile
- SETUP.CODE 28 15-Jul-81 173 512 Codefile
- SYSTEM.LIBRARY 9 7-Jan-81 201 512 Datafile

< UNUSED >
- 30 210

6/6 files<listed/in-dir>, 210 blocks used, 30 unused, 30 in largest

The SYSTEM disk:

SYS1:
- SYSTEM.EDITOR 49 28-Jul-81 6 512 Codefile
- SYSTEM.SYNTAX 14 4-Dec-80 55 512 Datafile
- DEBUGGER.CODE 29 22-Jul-81 69 512 Codefile
- PATCH.CODE 34 3-Nov-81 98 512 Codefile
- LIBRARY.CODE 13 6-Nov-81 132 512 Codefile

< UNUSED >
- 95 145

5/5 files<listed/in-dir>, 145 blocks used, 95 unused, 95 in largest

SYS2:
- YALE.CODE 12 2-Dec-80 6 512 Codefile
- BOOTER.CODE 3 4-Dec-80 18 512 Codefile
- DISKCHANGE.CODE 8 5-Dec-80 21 512 Codefile
- DISKSIZE.CODE 3 3-Dec-80 29 512 Codefile
- FINDPARAMS.CODE 9 3-Dec-80 32 512 Codefile
- SAMPLEGOTO.TEXT 4 17-Nov-78 41 512 Textfile
- DECODE.CODE 28 5-Mar-81 45 512 Codefile
- COPYDUPDIR.CODE 3 2-Dec-80 73 512 Codefile
- MARKDUPDIR.CODE 4 2-Dec-80 76 512 Codefile
- XREP.CODE 29 3-Dec-80 80 512 Codefile
- RECOVER.G.CODE 8 5-Dec-80 109 512 Codefile
- REALOPS.4.CODE 11 21-Aug-81 117 512 Codefile
- REALOPS.22.CODE 9 24-Nov-81 128 512 Codefile
- KERNEL.CODE 65 11-Sep-81 137 512 Codefile
- COMMANDIO.CODE 6 20-Jul-81 202 512 Codefile
- SCREENOPS.CODE 13 29-Jun-81 208 512 Codefile
- ABSWRITE.CODE 4 3-Dec-80 221 512 Codefile

< UNUSED >
- 15 225

17/17 files<listed/in-dir>, 225 blocks used, 15 unused, 15 in largest
The INTERP disk:

ZINT:
INTERP.Z.CODE  25 11-Nov-81  6 512 Codefile
FP8.CODE       4 11-Nov-81  31 512 Codefile
FP2.Z.CODE     7 11-Nov-81  35 512 Codefile
FP4.CODE       8 11-Nov-81  42 512 Codefile
RSP.CODE       6 13-Nov-81  50 512 Codefile
BIOS.CODE      8 13-Nov-81  56 512 Codefile
BIOS.C.CODE    8 20-Oct-81  64 512 Codefile
BIOS.CR.CODE   9 23-Nov-81  72 512 Codefile
BIOS.CRP.CODE  9 23-Nov-81  81 512 Codefile
INTER.CODE     4 1-Nov-81   90 512 Codefile
INTER.X.CODE   4 1-Nov-81   94 512 Codefile
INTER.CPM1.CODE 4 1-Nov-81  96 512 Codefile
INTER.CPM2.CODE 4 1-Nov-81 102 512 Codefile
INTER.CPM4.CODE 4 1-Nov-81 106 512 Codefile
TEXTBOOT.CODE  5 15-Oct-81 110 512 Codefile
< UNUSED >     125 115
15/15 files<listed/in-dir>, 115 blocks used, 125 unused, 125 in largest

ASM:
Z80.ASSEMBLER 51 2-Dec-80  6 512 Codefile
Z80.OPCODES    3 20-Dec-78  57 68 Datafile
Z80.ERRORS     8 21-Sep-80  60 70 Datafile
COMPRESS.CODE  10 3-Dec-80  68 512 Codefile
SYSTEM.LINKER  26 27-Jan-81 78 512 Codefile
SCREENTEST.CODE 13 4-Jun-81 104 512 Codefile
CPMBOOT.CODE   22 7-Jan-81  117 512 Codefile
< UNUSED >     139 139
7/7 files<listed/in-dir>, 139 blocks used, 139 unused, 139 in largest

For 8080 Systems, the release disks are the same, except that ADAPZ is replaced by ADAPB:

SYS808:
SYSTEM.INTERP  27 23-Nov-81  6 512 Datafile
SYSTEM.MISCINFO 1 24-Nov-81  33 194 Datafile
SYSTEM.PASCAL  108 24-Nov-81 34 512 Datafile
SYSTEM.FILER   32 28-Jul-81 142 512 Codefile
SETUP.CODE     28 15-Jul-81 174 512 Codefile
SYSTEM.LIBRARY  9 7-Jan-81  202 512 Datafile
< UNUSED >     29 211
6/6 files<listed/in-dir>, 211 blocks used, 29 unused, 29 in largest

SYS808D:
SYSTEM.INTERP  27 23-Nov-81  6 512 Datafile
SYSTEM.MISCINFO 1 24-Nov-81  33 194 Datafile
SYSTEM.PASCAL  108 24-Nov-81 34 512 Datafile
SYSTEM.FILER   32 28-Jul-81 142 512 Codefile
SETUP.CODE     28 15-Jul-81 174 512 Codefile
SYSTEM.LIBRARY  9 7-Jan-81  202 512 Datafile
< UNUSED >     29 211
6/6 files<listed/in-dir>, 211 blocks used, 29 unused, 29 in largest
2. CP/M Adaptable System

This release of the Adaptable p-System for CP/M may be brought up as described in the Version IV.0 Installation Guide. When CP/M is used to bootstrap the p-System, the only differences are in the format of virtual floppies on each release disk, and the distinction between software components for 2-word and 4-word reals, as described above.

However, if the user wishes to use print spooling (described in Chapter 5), then a new Extended SBIOS must be written, and it must implement QUIET, ENABLE, and the key-ready interrupt (by calling EVENT). These routines are described above in the section on the full Adaptable System.

CP/M Adaptable System Files

The CP/M release is identical to the Z80 or 8080 release, except that the ADAPZ or ADAP8 disk is replaced by the CPMADAP disk:

```
SYSCPM:
SYSTEM.INTERP   27 23-Nov-81   6   512 Datafile
SYSTEM.MISCINFO  1 24-Nov-81  33  194 Datafile
SYSTEM.PASCAL    108 24-Nov-81 34  512 Datafile
SYSTEM.FILER     32 28-Jul-81 142  512 Codefile
SETUP.CODE       28 15-Jul-81 174  512 Codefile
CPM2.INTERP      27 24-Nov-81 262  512 Datafile
SYSTEM.LIBRARY   9  7-Jan-81  229  512 Datafile
< UNUSED >       2   238
7/7 files<listed/in-dir>, 238 blocks used, 2 unused, 2 in largest
```

```
SYS808:
SYSTEM.INTERP   27 23-Nov-81   6   512 Datafile
SYSTEM.MISCINFO  1 24-Nov-81  33  194 Datafile
SYSTEM.PASCAL    108 24-Nov-81 34  512 Datafile
SYSTEM.FILER     32 28-Jul-81 142  512 Codefile
SETUP.CODE       28 15-Jul-81 174  512 Codefile
SYSTEM.LIBRARY   9  7-Jan-81  229  512 Datafile
< UNUSED >       29   211
6/6 files<listed/in-dir>, 211 blocks used, 29 unused, 29 in largest
```

Note that as above, utility programs are shipped on the second image of the SYSTEM disk.
3. 8086

This section describes the bootstraps, SBIOS interface, and BIOS interface for the 8086. You should already be familiar with an Adaptable System, as described in the Installation Guide.

8086 systems that run Version IV can take advantage of extended memory. This is described in Chapter 3 of this booklet.

The P-machine emulator ("interpreter") for the 8086 is broken into several modules. These are:

1) The main part of the Interpreter.
2) The RSP (Runtime Support Package).
3) The BIOS (Basic I/O Subsystem).
4) The tertiary bootstrap routine.

To use the 8086 Interpreter, the following modules (with their object code filenames to the right) must be linked together:

<table>
<thead>
<tr>
<th>Name</th>
<th>Object Code Filename(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>INTERPX.CODE or INTERPX.2.CODE or INTERPX.4.CODE</td>
</tr>
<tr>
<td>Runtime Support Package</td>
<td>RSP.CODE</td>
</tr>
<tr>
<td>BIOS</td>
<td>BIOS.CODE or BIOS.C.CODE or BIOS.CR.CODE or BIOS.CRP.CODE</td>
</tr>
<tr>
<td>Tertiary Bootstrap</td>
<td>TERTBOOT.CODE</td>
</tr>
</tbody>
</table>

The user is responsible for supplying SBIOS routines, as described in the Installation Guide.

The SYSTEM.INTERP supplied on the release disks contains no floating point support or I/O character queuing (INTERPX.CODE, RSP.CODE, BIOS.CODE, and TERTBOOT.CODE).

The ‘2’ and ‘4’ in the Interpreter codefiles distinguish between the 2-word and 4-word floating point packages. The Operating System must contain routines that use the corresponding floating point size (REALOPS unit in SYSTEM.PASCAL) if real numbers are to be used.

The ‘C’, ‘CR’, and ‘CRP’ in the BIOS code files indicate console, remote port, and printer character input queuing, as stated in the Installation Guide.

After linking, the utility program COMPRESS (described in the Users’ Manual) must be run to form a memory image file of the Interpreter. The answers to the
questions asked by COMPRESS are: NO relocatable output, base address = 0, the linker output file, and the desired filename (such as SYSTEM.INTERP).

**Bootstrapping the p-System**

The steps necessary to get the Adaptable p-System started are described here. Please note that all address values are to be set relative to the CS register (i.e., the Interpreter load point specified by the user).

1. Load the primary bootstrap (Track 0, Sectors 1 and 2) and the user supplied SBIOS.

2. Set the Stack Pointer to any unused (but VALID) address value (such as C000). The SP register will be reset to MEMTOP at the start of the TERTBOOT routine, so the value set here is not critical. Push the Adaptable System parameters onto the Stack as indicated in the **Installation Guide**. As previously stated, all address values are relative to the Interpreter load point. (See below.)

3. Set the CS register to the desired base (load point) of the Interpreter.

4. Do a short (intrapsegment) jump to the bootstrap address (8000H -- CS-relative). (The CS register may also be set by making a long (intrapsegment) jump.) This invokes the primary bootstrap.

The primary bootstrap reads the secondary bootstrap from disk and jumps to it. The secondary bootstrap finds the Interpreter, reads it in, and jumps to the tertiary bootstrap (which is part of the Interpreter). Once the System is up and running, a user may wish to simplify the bootstrapping mechanism (see the **Installation Guide**).
### SBIOS Interface

The SBIOS routines are called from the BIOS through an address vector (NOT a jump vector, as with other processors). The following table briefly defines this interface. Addresses pointed to by the vector offsets are CS-relative.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Vector offset</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSINIT</td>
<td>00</td>
<td>AX = pointer to Interpreter jump table</td>
<td></td>
</tr>
<tr>
<td>SYSHALT</td>
<td>02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONINIT</td>
<td>04</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>CONSTAT</td>
<td>06</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>CONREAD</td>
<td>08</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>CONWRT</td>
<td>0A</td>
<td>AL = char</td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>SETDISK</td>
<td>0C</td>
<td>AL = current disk</td>
<td></td>
</tr>
<tr>
<td>SETTRAK</td>
<td>0E</td>
<td>AL = current track</td>
<td></td>
</tr>
<tr>
<td>SETSECT</td>
<td>10</td>
<td>AL = current sector</td>
<td></td>
</tr>
<tr>
<td>SETBUF</td>
<td>12</td>
<td>AX = buffer addr (ES-relative)</td>
<td></td>
</tr>
<tr>
<td>DSKREAD</td>
<td>14</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>DSKWRT</td>
<td>16</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>DSKINIT</td>
<td>18</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>DSKSTRRT</td>
<td>1A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSKSTOP</td>
<td>1C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRNINIT</td>
<td>1E</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>PRNSTAT</td>
<td>20</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>PRNREAD</td>
<td>22</td>
<td></td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>PRNWRT</td>
<td>24</td>
<td>AL = char</td>
<td>AH = iorestart</td>
</tr>
<tr>
<td>REFINIT</td>
<td>26</td>
<td></td>
<td>AH = iorestart</td>
</tr>
</tbody>
</table>
REMSTAT  28
       AH = ioreresult
       AL = char present
REMREAD  2A
       AH = ioreresult
       AL = char
REMWRITE 2C
       AH = ioreresult
USRINIT  2E
       AL = unit #
USRSTAT  30
       TOS(SP)->return
    i/o toggle
   *statrec
   device #
USRREAD  32
       TOS(SP)->return addr
      block #
      byte count
   *buffer (DS-rel)
   device #
   control word
USRWRITE 34
       TOS(SP)->return addr
      block #
      byte count
   *buffer (DS-rel)
   device #
   control word
CLKREAD  36
       AH = ioreresult
       DX = high word
       CX = low word
QUIET    38
ENABLE   3A

Notes on the SBIOS:

1. The SBIOS must be assembled relative to zero (i.e., with no ORG
directive), and relocated to its load address (relative to the CS or DS register)
by using the COMPRESS utility.

2. The SBIOS is responsible for ensuring that the SS, DS, and CS registers are
restored to the same state as at entry. All other registers are available for
use.

3. The SBIOS vector contains only addresses, not jump instructions.
4. All SBIOS routines are called indirectly from the BIOS through the vector. To return to the BIOS, a RET instruction is all that is needed (these calls and returns are short (intrasegment) CALLs and RETs).

5. The 8086 Adaptable System requires an Extended SBIOS (all of the above entry points). If user, remote, clock, or printers are not present, the routine should simply return an iorestart of 9 (offline).

6. QUIET and ENABLE are new entry points not documented in the Installation Guide. QUIET contains the functions necessary to disable P-machine "events" from occurring. ENABLE allows events to occur. In most systems, the disable and enable interrupts functions (respectively) are all that are necessary (CLI and STI processor commands). A few hardware configurations may not permit the global disabling of processor interrupts, so some other scheme must be devised.

7. The routine vector passed to SYSINIT is described below.

**BIOS Routines Accessible to the SBIOS**

The use of the new routine EVENT is described in Chapter 6 of this booklet, which discusses interrupt handling.

If you intend to use print spooling, then you must use EVENT to signal a keys-ready interrupt (event number 19). See Chapter 5 of this booklet for more details.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Vector offset</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLUNITS</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>DSKCHNG</td>
<td>02</td>
<td>BX = disk descriptor block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(#tracks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#sectors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sector size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>interleaving</td>
</tr>
<tr>
<td></td>
<td></td>
<td>skew</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first track</td>
</tr>
<tr>
<td>EVENT</td>
<td>04</td>
<td>DI = event #</td>
</tr>
</tbody>
</table>

**The Primary Bootstrap**

The primary bootstrap must be assembled relative to zero (i.e., without an ORG directive), and relocated to 8000H by using the utility COMPRESS (described in the Users' Manual).

The address of the Interpreter must be set to zero. The CS register must be set to the desired Interpreter base (shifted right by 4 bits). The bootstrapping and interpretation are relative to this CS value. The DS, ES, and SS registers will be
set to the same value by the bootstrap. (ALL address values within the SB IOS must be relative to this CS value.)

The following parameters must be on the stack:

TOS --> SB IOS tester parameter (ignored by primary boot)
address of Interpreter (CS-relative)
address of SB IOS (CS-relative)
address of low word of contiguous memory (CS-relative)
address of high word of contiguous memory (CS-relative)
number of tracks on boot disk
number of sectors per track
number of bytes per sector
interleaving factor
first interleaved track
track-to-track skew
maximum number of sectors in table
maximum number of bytes per sector

The primary bootstrap must pop these values from the Stack, load the SB IOS, and call the SB IOS SYSINIT routine. It must then read in the secondary bootstrap, which is located on Track 0.

Next, the primary bootstrap must push the following values onto the Stack:

TOS --> address of Interpreter (relative to CS)
address of SB IOS (CS-relative)
address of low word of contiguous memory (CS-relative)
address of high word of contiguous memory (CS-relative)
number of tracks on boot disk
number of sectors per track
number of bytes per sector
interleaving factor
first interleaved track
track-to-track skew
maximum number of sectors in table
maximum number of bytes per sector

(Note that these values are the same as before, except that the SB IOS test parameter is no longer present.)

Finally, the primary bootstrap must jump to the secondary bootstrap.

The Secondary Bootstrap

Like the primary bootstrap, the secondary bootstrap must be assembled relative to zero (no ORG). It must be linked with a copy of the BIOS, and relocated to 8200H by using the utility COMPRESS.

Again, the CS register must be set to the Interpreter load address, and the DS,
ES, and SS registers must be set to the same value.

The secondary bootstrap carries out the following steps:

1. Pop the parameters off the Stack.

2. Allocate the sector translation table in high memory.

3. Allocate the partial sector read buffer.

4. Update the value of the "address of the highest word of contiguous memory" parameter (to protect the tables just allocated).

5. Call the BIOS routines SYSin, CONSin, and DSKIN (for the booting drive). These routines must be called in this order.

6. Read the directory from the booting disk.

7. Read the Interpreter into the desired location (i.e., the value in CS).

8. Restack the parameters for the tertiary bootstrap, as follows:

   TOS --> unit number of boot disk
   address of Interpreter   (CS-relative)
   address of SBIOS    (CS-relative)
   addr of low word of contiguous memory (CS-relative)
   addr of high word of contiguous memory (CS-relative)
   number of tracks on boot disk
   number of sectors per track
   number of bytes per sector
   interleaving factor
   first interleaved track
   track-to-track skew
   pointer to sector translation table (CS-relative)
   pointer to partial sector read buffer (CS-relative)

9. Do a short (inrasegment) jump to the tertiary bootstrap (indirectly through the Interpreter load address).

The Tertiary Bootstrap

The tertiary bootstrap must be linked with the Interpreter, RSP, and BIOS. It is the LAST "module" within this codefile.

Once the tertiary bootstrap has been run, the memory it occupies is once again available to the System.

These are the steps that the tertiary bootstrap must follow:

1. Set the DS and ES registers to the value in CS (Interpreter load point).
2. Pop the parameters off the Stack, and pass them to the BIOS.
3. Call the BIOS SYSIN routine.
4. Do a **unitclear** on the console.
5. Get the "address of the highest word of contiguous memory" parameter, and set the Stack pointer to this value.
6. Do a **unitclear** on the booting disk drive.
7. Read the directory of the booting drive, find SYSTEM.PASCAL, and read in its segments 1 and 15.
8. Build the Operating-System data structures.
9. Set up the divide-by-zero interrupt.
10. Put the CS value into the common block of the Interpreter (this is for the Native Code Generator (CODEGEN) jump).
11. Jump to the Interpreter’s fetch loop.
The BIOS Interface

The BIOS routines are called from the RSP through an address vector. The following table briefly defines this interface. The return address in the RSP is always on the top of stack at the call.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Vector offset</th>
<th>Inputs/Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSRD</td>
<td>00</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AL = char</td>
</tr>
<tr>
<td>CONSWR</td>
<td>02</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td>CONSIN</td>
<td>04</td>
<td>AL = char</td>
</tr>
<tr>
<td>CONSST</td>
<td>06</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td>PRNRD</td>
<td>08</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AL = char</td>
</tr>
<tr>
<td>PRNWRI</td>
<td>0A</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td>PRNIN</td>
<td>0C</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td>PRNST</td>
<td>0E</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td>DSKRD</td>
<td>10</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOS = block #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*buffer (ES-relative)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drive # (0..5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>control word</td>
</tr>
<tr>
<td>DSKWR</td>
<td>12</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOS = block #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*buffer (ES-relative)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drive # (0..5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>control word</td>
</tr>
<tr>
<td>DSKIN</td>
<td>14</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CL = drive # (0..5)</td>
</tr>
<tr>
<td>DSKST</td>
<td>16</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CL = drive # (0..5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOS = *status record</td>
</tr>
<tr>
<td></td>
<td></td>
<td>control word</td>
</tr>
<tr>
<td>REMRD</td>
<td>18</td>
<td>AH = ioresult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AL = char</td>
</tr>
<tr>
<td>Command</td>
<td>Value</td>
<td>AL</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>REMWR</td>
<td>1A</td>
<td>AL = char</td>
</tr>
<tr>
<td>REMIN</td>
<td>1C</td>
<td></td>
</tr>
<tr>
<td>REMST</td>
<td>1E</td>
<td>TOS = &quot;status record control word&quot;</td>
</tr>
<tr>
<td>USERRD</td>
<td>20</td>
<td>TOS (\rightarrow) block # byte count &quot;buffer (DS-rel) device # control word</td>
</tr>
<tr>
<td>USERWR</td>
<td>22</td>
<td>TOS (\rightarrow) block # byte count &quot;buffer (DS-rel) device # control word</td>
</tr>
<tr>
<td>USERIN</td>
<td>24</td>
<td>AL = unit #</td>
</tr>
<tr>
<td>USERST</td>
<td>26</td>
<td>TOS (\rightarrow) i/o toggle &quot;statrec device #&quot;</td>
</tr>
<tr>
<td>SYSRD</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>SYSWR</td>
<td>2A</td>
<td></td>
</tr>
<tr>
<td>SYSIN</td>
<td>2C</td>
<td></td>
</tr>
<tr>
<td>SYSST</td>
<td>2E</td>
<td>TOS (\rightarrow) &quot;statrec control word&quot;</td>
</tr>
</tbody>
</table>
8086 Adaptable System Files

Here are the directories of the current release disks for the 8086:

The SYSTEM disk:

SYS1:
SYSTEM.EDITOR  49 28-Jul-81  6  512 Codefile
DEBUGGER.CODE  29 22-Jul-81 55  512 Codefile
PATCH.CODE     34 3-Nov-81  84  512 Codefile
SYSTEM.SYNTAX  14 4-Dec-80 118  512 Datafile
< UNUSED >      100  132
4/4 files<listed/in-dir>, 132 blocks used, 108 unused, 108 in largest

SYS2:
ASM8086.CODE   60 29-May-81  5  512 Codefile
8086.OPCOPDES  5 5-May-81  66  52 Datafile
8086.ERRORS    11 8-May-81  71  130 Datafile
8086.POPS      1 4-Feb-81  82  512 Codefile
YALOE.CODE     12 2-Dec-80  85  512 Codefile
BOOTER.CODE    3 4-Dec-80  97  512 Codefile
DISCMCHANGE.CODE 8 5-Dec-80 130  512 Codefile
DISKSIZE.CODE   3 3-Dec-80 100  512 Codefile
FINDPARAMS.CODE 9 3-Dec-80 111  512 Codefile
SAMPLEGOTO.TEXT 4 17-Nov-78 120  512 Textfile
CODE.ECODE     28 5-Mar-81 124  512 Codefile
COPYUPDIR.CODE  3 2-Dec-80 152  512 Codefile
MARKUPDIR.CODE  4 2-Dec-80 155  512 Codefile
XREF.CODE      29 3-Dec-80 159  512 Codefile
RECOVER.G.CODE  9 5-Dec-80 108  512 Codefile
SCREENTEST.CODE 13 4-Jun-81 196  512 Codefile
LIBRARY.CODE    13 6-Nov-81 209  512 Codefile
ABWRITE.CODE   14 3-Dec-80 222  512 Codefile
< UNUSED >      14  226
18/18 files<listed/in-dir>, 226 blocks used, 14 unused, 14 in largest

The INTERP disk:

86SYS:
SYSTEM.INTERP 19 15-Dec-81  6  512 Datafile
SYSTEM.MISCINFO 1 24-Nov-81 25 194 Datafile
SYSTEM.PASCAL 100 24-Nov-81 26  512 Datafile
SYSTEM.FILER 22 29-Jul-81 134  512 Codefile
SETUP.CODE   20 15-Jul-81 166  512 Codefile
SYSTEM.LIBRARY 9 23-Sep-81 194  512 Datafile
SYSTEM.LINKER 26 27-Jan-81 203  512 Codefile
< UNUSED >    11  229
7/7 files<listed/in-dir>, 229 blocks used, 11 unused, 11 in largest

86INT:
COMPRESS.CODE  10 3-Dec-80  6  512 Codefile
INTERP.CODE     30 6-Oct-81 14  512 Codefile
INTERPX.2.CODE  23 6-Oct-81 36  512 Codefile
INTERPX.4.CODE  24 6-Oct-81 59  512 Codefile
RSP.CODE        6 18-Sep-81 83  512 Codefile
BIOS.CODE       6 26-Sep-81 89  512 Codefile
BIOS.C.CODE     6 26-Sep-81 95  512 Codefile
BIOS.CR.CODE    6 26-Sep-81 101  512 Codefile
BIOS.CRP.CODE   6 26-Sep-81 107  512 Codefile
TERMBOOT.CODE   7 17-Sep-81 113  512 Codefile
REALOPS.2.CODE  10 21-Aug-81 120  512 Codefile
REALOPS.4.CODE  11 21-Aug-81 130  512 Codefile
KERNEL.CODE     65 11-Sep-81 141  512 Codefile
COMMANDIO.CODE  6 28-Jul-81 206  512 Codefile
SCREENOPS.CODE  13 29-Jun-81 212  512 Codefile
< UNUSED >      5  225
15/15 files<listed/in-dir>, 225 blocks used, 15 unused, 15 in largest

These are the only two disks required for the 8086 system. Note that utility programs are shipped on the second image of the SYSTEM disk.
2. SYMBOLIC DEBUGGER

This product is available on all processors that support the current release.

This chapter describes the Debugger utility. The Debugger can be used as an aid to debugging compiled programs. It can be invoked from the main System promptline, or during the execution of a program (when a breakpoint is encountered). Memory may be displayed and altered, P-code may be single-stepped, Markstack chains may be displayed and traversed, and so forth.

There are no promptlines explaining the Debugger commands because such prompts would detract from the information displayed by the Debugger itself. When a command is entered, there are usually several prompts that may ask for further information such as a segment name, variable offset, and so forth.

In order to use the Debugger properly, it is necessary to be familiar with the UCSD P-machine architecture. The user should understand the P-code operators, Stack usage, variable and parameter allocation, and so forth. These topics are discussed in the Internal Architecture Guide.

When using the Debugger, it is also useful to have a compiled listing of the program being debugged. This listing helps determine P-code offsets and similar information, and should be current.

You should be aware that the Debugger is a low-level tool, and as such, must be used with caution. If the Debugger is used incorrectly, the System can die.

It is easier to use the Debugger if the code being debugged has been compiled with the $D+$ option (the Users' Manual describes compiler options in general). The $D$ option (which defaults to $D-$) instructs the Compiler to output symbolic debugger information for those portions of a program that are compiled with $D+$ turned on. Variables within a given routine may be specified by name (rather than data segment offset number) if at least one statement within that routine is compiled $D+$. Breakpoints may be specified by line number (rather than P-code offset number) for all statements covered by the $D+$ option. Once a program is debugged, however, it should be recompiled without symbolic debugger information, because this information increases the size of the codefile.

Basic Debugger Commands

The Debugger may be entered from the main System promptline by typing 'D'. Whenever the Debugger is entered in a 'fresh' state, the prompt, 'DEBUG [version #]', appears and a '(' is displayed on the second line. Being in a 'fresh' state means that the Debugger was not previously active and no breakpoints are currently enabled. If the Debugger is entered in a 'non-fresh' state, only the '(' appears.

Many of the Debugger commands require two characters (such as 'LP' for L(is) P(code), or 'LR' for L(is) R(e)gister). If, after typing the first character, you decide to exit the command, simply type <space> and the main mode of the
Debugger is re-invoked.

The Debugger may be exited by typing Q uit, R esume, or S tep. If the Debugger is exited using the Q uit option, it is disabled. If it is later re-invoked, it is in a 'fresh' state. If the Debugger is exited using the R esume option, execution continues from where it left off and the Debugger is still active. If it is then re-invoked, it is in a 'non-fresh' state. If the Debugger is exited by using the S tep option, a single P-code operator is executed and then the Debugger is automatically re-invoked (in a 'non-fresh' state).

If a program is running under the Debugger's R esume command, it may force a return to the Debugger by calling the halt intrinsic (described in the Users' Manual). In fact, ANY runtime error causes a return to the Debugger, if the Debugger is active while the program is running.

If you wish to enter the Debugger while a program is running, but do not wish to alter the program's code at all, the Debugger itself may be used to set breakpoints.

Breakpoints are handled by typing 'B' for B reakpoint. After 'B' is typed, one of the following commands must be used: S et, R emove, or L ist.

If 'S' is typed (after the 'B'), then a breakpoint may be set. The user may have, at most, five breakpoints numbered 0 through 4. The first prompt is 'Set Break # ?': a digit 0..4 should be typed followed by <space>. The next prompt is 'Segname?': the name of the desired segment should be typed followed by <space>. Then 'Procname or #?' appears: the number of the desired procedure OR the first eight characters of the valid procedure name should be typed followed by <space>. If a procedure number is entered, then 'Offset #?' appears: the desired offset within the procedure should be typed followed by <space>. If a procedure name is entered after the 'Procname or #?' prompt, the following line is displayed: 'First # _ Last # _ Line #?'. The underlines are actually numbers that indicate the first and last line numbers. The desired line number within the specified range should be entered. A breakpoint is then set and if, during execution resumption, that segment, procedure, and offset are encountered, the Debugger is automatically re-invoked.

Note: Under the B reakpoint command, the 'First # _ Last # _ ' information can be displayed ONLY if the Debugger finds symbolic debugger information in the codefile; that is, this mode can only be used if you have included a [$D+] compiler option within the procedure where you wish to place a breakpoint.

Whether you set breakpoints by segment and offset number, or by procedure name and line number, it should be evident that having a compiled listing of your program will make it far easier to determine where the breakpoint is.

When setting a breakpoint, a space may be typed for the break number, segment name, etc. Rather than exiting the breakpoint command (as would happen with other commands), the previous breakpoint's information is used. For example, if it is desired to break in the same segment and procedure but with a different offset, a space may be typed for everything except the offset.
If, after typing B(reakpoint), an ‘R’ is typed, a breakpoint may be removed. The prompt ‘Remove break #?’ appears. To remove a breakpoint, type its number followed by a <space>.

If after typing B(reakpoint), an ‘L’ is typed, the current breakpoints are listed.

The Debugger may be memlocked or memswapped (see the descriptions of those intrinsics) by using the M(emory command at the outer level. ‘ML’ memlocks and ‘MS’ memswaps the Debugger.

Displaying and Altering Memory

The V(ar command allows data segment memory to be displayed. This is another two-character command and may be followed by G(labal, L(ocal, I(ntermediate, E(xtended, or P(rocEDURE. If ‘G’ or ‘L’ is typed, the prompt ‘Vamame or Offset #?’ appears; the desired offset into the data segment or variable name should be typed. (Note: Only ‘Offset #’ appears if symbolic debugger information cannot be found.) If ‘I’ is typed, ‘Delta Lex Level?’ is also prompted (when an offset number is input). If ‘E’ is typed, the prompts ‘Seg #’ and ‘Offset #’ are displayed (extended variables may not be specified symbolically). If ‘P’ is typed, an offset within a specified procedure may be displayed: ‘Segment name?’, ‘Procname or #?’ and ‘Vamame or Offset #?’ are all prompted in sequence.

When any of the non-symbolic options are used, a line similar to the following is displayed:

```
( l) S=INIT  P#1  VO#1  2C1A: 0B 05 53 43 41 4C 43 61  --SCALCa
```

In this example, a Local (l) segment of memory is displayed. The segment is INIT, procedure 1, variable offset 1 at absolute hex location 2C1A. Following this, eight bytes are displayed, first in HEX and then in ASCII (a ‘-‘ indicates that the character is not a printable ASCII character).

If the desired variable had been entered symbolically, the following line would appear:

```
( l) S=INIT  P=FILLTABL  V=TABLE1  2C1A: 0B 05 53 43 41 4C 43 61  --SCALCa
```

It is possible to change the frame of reference from which the global, local, and intermediate variables are viewed. This can be done by using the C(hain command. After ‘C’ is typed the following three options are available: U(p, D(own and L(ist. If ‘L’ is typed, all of the currently existing mark stacks are displayed, with the most recently created one first. An entry in the list resembles the following:

```
(ms) S=HEAPOPS P#3 O#23 mstat=347C msdyn=F0A0 mspcn=01DA msenv=FEE8
```

If the U(p or D(own options are used, the frame of reference moves up or down one link and variable listings (using the ‘V’ command) change accordingly.
After a line has been displayed by the V(ar command, a ‘+’ or ‘-’ may be typed. This displays the succeeding or preceding eight bytes of memory. If a ‘/’ is typed, then the line displayed above it may be altered in hex mode. If a ‘/’ is typed, then the line displayed above it may be altered in ASCII mode. When altering in hex mode, any characters that are to be left unchanged may be skipped by typing <space>. In the ASCII mode, any characters to be left unchanged may be skipped by typing <return>.

A text file may be viewed from the debugger by typing F(ile. The ‘Filename?’ ‘First line #?’ and ‘Last line #?’ prompts are then displayed. This command lists as many lines as possible in the window between ‘first line’ and ‘last line’ of the indicated file.

Further Single-Stepping Options

When the single-stepping mode (the S tep command described above) is used, one P code operator is executed at a time. When control is returned to the Debugger, it displays various pieces of information if they are desired. In order to select what will be displayed, the E(nable mode should be used. After typing ‘E’, the following options are available: R(egister, P(code, M(arkstack, A ddress, and L(oad. Any or all of these options may be enabled at the same time.

If R(egister is enabled, a line such as the following is displayed after each single step:

(rg) mp=F082 sp=F09C ered=FE8 seg=9782 icp=01C3 tib=0493 rdyq=2EBC

If P(code is enabled, a line such as the following is displayed after each step:

(cd) S=HEAPOPS P#3 O#23 LLA 1

If M(arkstack is enabled, a line such as the following is displayed after each step:

(ms) S=HEAPOPS P#3 O#23 mastat=347C msdyn=F0A0 msipc=01DA mserv=FEE8

If A ddress is enabled, a line such as the following is displayed after each step:

(a ) S=HEAPOPS P#3 O#23 2C1A: 0B 05 53 43 41 4C 43 61 --SCALCa

In order to initialize this address to a given value, there is an A ddress mode at the outer level. When ‘A’ is typed, ‘Address ?’ appears. An absolute address, in hex, should be typed in. At this point, eight bytes are displayed starting at that address. Also, that address is now displayed if the E nable A ddress option is on.

Enabling E very causes all of the above options to be enabled.

The D(isable mode disables any of the options just described. The L ist mode lists any of the above options.
Also, at the outer level, there is a P(code option). This option asks for 'Segment name?', 'Procname or #?', and either 'First #_ Last #', 'Start Line #?', 'End Line #?' or 'Start Offset #?', 'End Offset #?'. This command disassembles the indicated portion of code. This may be useful during single-step mode if you wish to look ahead in the P-code stream. This mode may be exited before it reaches the ending offset by typing <break>; control returns to the Debugger.

Example of Debugger Usage

Suppose the following program is to be debugged:

```
Pascal Compiler IV.0

1 0 0:d 1 { $L LIST.TEXT }  
2 2 1:d 1 PROGRAM NOT DEBUGGED;  
3 2 1:d 1 VAR I,J,K:INTEGER;  
4 2 1:d 4 B1,B2:BOOLEAN;  
5 2 1:0 0 BEGIN  
6 2 1:1 0 I:=1;  
7 2 1:1 3 J:=1;  
8 2 1:1 6 IF K <> 1 THEN WRITELN  
    ( 'What's wrong?' );  
9 2 :0 0 END.

End of Compilation.
```

First we enter the Debugger and set a breakpoint at the beginning of the IF statement:

(BS) Set break #? 0 Segname? NOTDEBUG Procname or #? 1 Offset #? 6
(EP)
(R)

After setting the breakpoint we enable P-code (EP) and resume (R). Now we execute the program above, and when it reaches offset 6, the Debugger is entered. We single-step twice:

Hit break #0 at S=NOTDEBUG P#1 0#6
(cd) S=NOTDEBUG P#1 0#6 SLDO1
(cd) S=NOTDEBUG P#1 0#7 SLDC1
(cd) S=NOTDEBUG P#1 0#8 NEQUI

We see that our first single-step did a short load global 1. (Note: This puts K on the stack. K is NOT global 3; I is global 3, J is global 2, and K is global 1. Every string of variables (such as 'I, J, K' in a declaration) is allocated in reverse order. Boolean B1, which follows, is at offset 5, and B2 is at offset 4. Parameters, on the other hand, ARE allocated in the order in which they appear.) The second single-step did a short load constant 1 onto the Stack. Now we are about to do an integer comparison (<>). But this is where our error shows up, so
we decide to look at what is on the Stack before doing this comparison:

(LR)
(rg)  mp=EB62  sp=EB82  erec= ...
(A )  Address? EB82
(a )  EB82: 01 00 C5 14 ...

We list the registers and then look at the memory address that sp points to. What we discover is a 1 on top of the stack (01 00: this is a least-significant-byte-first machine) followed by a word of what appears to be garbage. This leads us to suspect that K was not initialized. Looking over the listing, we quickly realize that this is the case.
Summary of the Commands

A(dress) Displays a given address
B(reak point) Segment, procedure and offset must be specified
       S(et) Allows a break point (0 through 4) to be set
       R(emove) Allows a break point to be removed
       L(ist) Lists current break points
C(hain) Changes frame of reference for V(ariable command
       U(p) Chains up mark stack links
       D(own) Chains down mark stack links
       L(ist) Lists current mark stacks
D(isable) Disables the following from being displayed
E(nable) Enables the following to be displayed during single step
F(ile) Allows viewing of text files
L(ist) Lists the following
       R(egister) The registers: mp, sp, ered, seg, ipc, tib, rdyq
       P(code) Current P-code mnemonic
       M(arkstack) Mark stack display
       A(dress) A given address
       E(very) All of the above
M(emory)
       L(ock) Memlocks the Debugger
       S(wap) Memswaps the Debugger
P(code) Dissassembles a given procedure
Q(uit) Quits the Debugger, ‘fresh’ state if re-entered
R(esume) Exits Debugger, Debugger remains active, ‘non-fresh’
S(tep) Single steps P-code and returns to Debugger
V(ariable)
       G(lbal) Displays global memory
       L(ocal) Displays local memory
       I(nter) Displays intermediate memory
       P(rocedure) Displays data segment of given procedure
       E(xtended) Displays variables in another segment
3. EXTENDED MEMORY

Extended Memory is available only on systems that use an 8086 processor.

If your system is configured with extended memory, then the Codepool resides in a different area than the Stack and the Heap. This allows much more space for code and data, and greatly reduces the chances of a Stack overflow.

A Codepool that resides between the Stack and the Heap (as in all Version IV.0 Systems) is called an "internal" Codepool. A Codepool that occupies a memory page of its own (as in systems with extended memory) is called an "external" Codepool.

The segments in an external Codepool may not need to be moved or swapped as often as the segments in an internal Codepool, so extended memory provides a speed advantage as well.

The description of Codepool handling that appears in Chapter 3 of the Internal Architecture Guide still applies, except that the Operating System's Codepool descriptor has been changed, as described below.

The following fragment of Pascal shows the declarations for a Codepool descriptor in the current Operating System:

```pascal
type
  CodePool: ^PoolDes;
  {points to a description of the Codepool}

PoolDes: record
  PoolBase: FullAddress;
  {the 1st physical 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.}
  MinOffset: Memptr;
  {Byte offset of the lowest useable value in the pool.
   If the pool is internal this value depends on HeapTop;
   otherwise it depends only on segment alignment requirements (if any)}
  MaxOffset: Memptr;
  {Byte offset one word past the highest position in the pool. If the pool is internal, it also equals the SP_LOW value of the main task.}
```

25
Resolution: integer;
   {Segment alignment
    requirements in bytes
    (machine-dependent).}
PoolHead: SIB P;
   {Points to the SIB of the
    segment at the base of
    the pool. If the pool is
    internal, this is the
    segment nearest the Heap.}
PermSIB: SIB P;
   {Points to the SIB of the
    first segment that is
    locked in the pool.}
SP_Low: Memptr;
   {The lowest possible bound
    of the Stack; if the pool
    is external, this is ignored;
    if the pool is internal, this
    is one word above the top of
    the pool.}
HeapTop: Memptr;
   {Points to the top of the
    Heap; ignored if the pool
    is external.}
Extended: Boolean;
   {True if the Codepool is
    external.}

end;

The Codepool is managed only by the FAULTHANDLER segment within KERNEL.

The p-System is shipped with extended memory disabled. If you have sufficient memory in your 8086 system, you may enable extended memory by doing the following steps.

Execute the SETUP utility, and set the following data items in SYSTEM.MISCINFO to the values described:

CODEPOOL BASE[First Word]
CODEPOOL BASE[Second Word]

This is a 32-bit address that denotes the lowest address of the memory page in which you want the Codepool to reside. ‘First Word’ contains the four high-order hex digits, and ‘Second Word’ contains the four low-order hex digits. The least significant hex digit of the second word must be zero, so that memory segments will align properly. The PoolBase field in the Pooledes record will be set to this value.

Important: If you do NOT use extended memory, then the value of both words MUST be set to zero.
CODE POOL SIZE

This should equal the number of WORDS of memory in the Codepool area, MINUS ONE. This value can be as high as 32767 (a 64K-byte area). It can also be smaller, if desired. The Poolsize field in Pooldes will be set to this value. This value is ignored if HAS EXTERNAL MEMORY is false.

HAS EXTENDED MEMORY

This should be set to true if you are to use extended memory. This should be set to false if 64K (or less) memory is to be used. The Extended field in Pooldes will be set to this value.

SEGMENT ALIGNMENT

This should be set to 16 (base ten), which is the segment alignment for 8086 processors. The Resolution field in Pooldes will be set to this value.

After running SETUP and changing these values to fit your hardware, you must reboot in order for the changes to take effect.
4. NATIVE CODE GENERATION

This product is presently available for systems with Z80 and 8086 processors.

The Native Code Generator is a utility that takes an executable P-code codefile as input, and produces another executable codefile. The output file, however, contains a mixture of P-code and native code (N-code) for either the Z80 or the 8086 (whichever processor you are using). The Code Generator selectively translates embedded sections of the P-code input file into equivalent N-code.

Generally, N-code executes much more quickly than P-code, but requires more memory space. Time-critical code may take fuller advantage of the processor's speed if it is translated into N-code.

The selection of which code is translated into native code has been left under user control. The user specifies what code he wishes translated to N-code by enclosing the desired sections with the compile-time switches $N+$ and $N-$. When the Compiler encounters the $N+$ option, it begins emitting additional P-codes that contain information necessary for the Code Generator to perform its translation. When it encounters the $N-$ option, it discontinues the generation of the additional P-codes. The default setting for this compiler option is $N-$.

An entire routine (Procedure, Function etc.) is the smallest unit that can be translated into Native Code at one time (in the current implementation). The $N+$ must occur before the first begin within that routine. The Code Generator will not generate any Native Code unless at least one entire routine is included between a $N+$ and the corresponding $N-$.

The object code produced by the Compiler from source containing the $N+$ option is executable like any other P-code file. The only difference is a very slight increase in codefile size due to the extra P-code hooks placed there for possible native code generation.

If there are any references to assembly language routines within a codefile, these routines must be linked in before that codefile may be processed through the Code Generator (see the chapter on the Adaptable Assembler and the section on the Linker in the Users' Manual for information about linking assembly language routines into codefiles).

The Code Generator will not necessarily translate all P-codes within the section(s) of code specified by the user into N-code. Due to the unwieldy nature of their machine code equivalents, some P-codes will be left in their original P-code form. Also, only those sections of P-code that encompass an entire routine (procedure, function, or process in UCSD Pascal) will be translated.

The Code Generator accepts codefiles generated by any of the compilers released as part of the UCSD p-System. The Code Generator will only fail to generate an output file if the input file is not a valid executable codefile.

The Code Generator produces an object codefile whose execution behavior is identical to that of the P-code codefile, except for differences with respect to
execution speed, object code size, or implementation dependencies (for example, the evaluation of conditional expressions is often implementation-dependent). If a loop were constructed in the source program that checked the value of a variable and used the variable to index into an array in the same expression, it would be possible for the P-code object file to give a value range error for some value of the variable. The corresponding N-code however, might (in some situations) short-circuit the array indexing and subsequent value-range check if the first test failed.

Concurrency is implemented in such a way that P-codes are uninterruptable operations. If a P-machine interrupt occurs during the execution of a P-code, the event is queued until the P-code finishes. In this respect, any embedded N-code in the codefile behaves as if it were a single P-code. If the user has a native code routine bound into a codefile, then the execution of that routine appears as a single uninterruptable P-code. Any P-machine interrupts that occur during the execution of the native code routine are queued until the end of the routine. Since the Code Generator does not, in general, translate all P-codes in the selected sections of the input file into N-code, it is typically a much smaller sequence of N-code that appears as a single P-code to the P-machine.

It is possible to force a poll within a routine that will be translated into native code by the Code Generator. To do this, simply include the pseudo-comment "\$N-N" in the source code.

When you run the Code Generator, it prompts you for an input file name and an output file name. If the input file name does not have `.CODE' appended to it, and a file with that name cannot be found in the directory, the Code Generator appends `.CODE' to the name and tries again. This does NOT happen when you specify the output file name. If you wish the output file to have a name with a `.CODE' suffix (which is generally the case), you must type `.CODE' yourself.

Finally, the Code Generator can optionally output an assembly language format listing for each routine that it translates.
5. PRINT SPOOLING

This product is available on systems with an 8086, Z80, or 8080 processor. Users of CP/M Adaptable Systems MUST write their own SBIOS in order to use print spooling. The requirements are mentioned in this chapter, and described in Chapter 1.

The print spooler allows you to make a queue of files that are printed concurrently with normal execution of the p-System. The program itself is called SPOOLER.CODE, and it MUST reside on the System disk. The queue it creates is a file called *SYSTEM.SPOOLER.

When SPOOLER is eX(ecuted), the following promptline appears:

Spool: P(rint, D(elete, L(ist, S(uspend, R(esume, A(bort, C(lear, Q uit

P(rint prompts for the name of a file to be printed. This name is then added to the queue. If SYSTEM.SPOOLER does not already exist, it is created. In the simplest case, P(rint may be used to send a single file to the printer. Up to 21 files may be placed in the print queue.

D(elete) prompts for a filename to be taken out of the print queue. All occurrences of that file name are taken out of the queue.

L(ist) displays the files currently within the queue.

S(uspend) temporarily halts the printing of the current file.

R(esume) continues the printing of the current file after a S(uspend. R(esume also starts printing the next file in the queue after an error or an A(bort.

A(bort) permanently stops the printing process of the current file and takes it out of the queue.

C(lear) deletes all file names from the queue.

Q(uit) exits the Spooler utility and starts transferring files to the printer.

If an error occurs (e.g., a nonexistent file is specified in the queue), the error message appears only when the p-System is at the main System promptline. If necessary, the Spooler waits until the user returns to the outer level.

Program I/O to the printer may run concurrently with spooled I/O. The Spooler finishes the current file and then turns the printer over to the user program. (The user program is suspended while it waits for the printer.) The user program should only do Pascal (or other high-level) writes to the printer. If the user program does printer I/O using unitwrite, the I/O is sent immediately and appears randomly interspersed with the I/O going on in the background.

The utility SPOOLER.CODE makes use of the Operating System unit SPOOLOPS. Within this unit there is a process called Spooltask. Spooltask is start'ed at boot
time and runs concurrently with the rest of the UCSD p-System. When the file
*SYSTEM.SPOOLER exists, Spooltask prints the files that it names. Spooltask runs
as a "background" to the main operations of the p-System.

*SPOOLER.CODE interfaces with SPOOLOPS and uses routines within it to
generate and alter the print queue within *SYSTEM.SPOOLER.

To enable spooling, the HAS SPOOLING data item in *SYSTEM.MISCINFO must be
TRUE. In addition, the QUIET and ENABLE routines in the Extended SBIOS must
be implemented (see Chapter 1). Finally, the Interpreter's EVENT routine must be
called from the Extended SBIOS with a keys-ready interrupt.

Whenever a key is struck on the console, the SBIOS must call EVENT with an
event number of 19. At present, this "keys-ready" interrupt is used only by
SPOOLOPS, and the event number 19 is reserved for this purpose.

For more information on EVENT, see the following chapter, Chapter 6.
6. INTERRUPT HANDLING

The programmer may define interrupts and how to handle them for systems with any currently supported processor. At present, the only "predefined" interrupt that is visible to the user is the keys-ready interrupt required for print spooling (see the preceding chapter, and Chapter 1).

Compiled programs handle interrupts by use of the attach intrinsic. To use this facility, it is necessary to make some provisions in the SBIOS. These are described in this chapter.

The current SBIOS is capable of calling these Interpreter routines:

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Vector Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLUNITS</td>
<td>0</td>
<td>polls character I/O devices</td>
</tr>
<tr>
<td>DSKCHNG</td>
<td>1</td>
<td>changes disk format values</td>
</tr>
<tr>
<td>EVENT</td>
<td>2</td>
<td>signals an event</td>
</tr>
</tbody>
</table>

(For the 8086, these routines are called through an address vector rather than a jump table. See Section 1.3.)

EVENT is a new routine that can signal some event (typically a hardware event). There are no currently defined user events, but the programmer may define such events in the SBIOS, and use EVENT to implement them.

Each event that the programmer wishes to recognize is given an event number. When the SBIOS detects one of these events, it passes the appropriate number to EVENT.

If a Pascal program has attached a semaphore to an event number (as described in the Users’ Manual), then a call to EVENT with that event number signals the semaphore. The signal takes place before the next P-code is executed. If the event number has not been attached, then the call to EVENT is ignored.

The SBIOS programmer may define event numbers in the range 32..63. The numbers 0..31 are reserved for the System’s use, and must NOT be used (except to enable print spooling, as described in the note below).

EVENT destroys ALL processor registers except the Stack Pointer.

Note: If you wish to support print spooling, then the Extended SBIOS must call EVENT with an event number of 19 whenever a key is pressed on the console. This is a System-supported event number, and there is no need to attach it.
Using KERNEL

1:u 1

CONST

1:u 1 VERSION = 'IV.1 B48J';

1:u 1 MMAXINT = 32767;  { MAXIMUM INTEGER VALUE }

1:u 1 MAXDIR = 77;      { MAX NUMBER OF ENTRIES IN A DIRECTORY }

1:u 1 VIOLEN = 7;       { NUMBER OF CHARS IN A VOLUME ID }

1:u 1 TIOLEN = 15;      { NUMBER OF CHARS IN TITLE ID }

1:u 1 MAXSEG = 15;      { MAX CODE SEGMENT NUMBER }

1:u 1 FBLKSIZE = 512;   { STANDARD DISK BLOCK LENGTH }

1:u 1 DIRBLK = 2;       { DISK ADDR OF DIRECTORY }

1:u 1 AGELIMIT = 300;   { MAX AGE FOR G9IRF...IN TICKS }

1:u 1 EOL = 13;         { END-OF-LINE...ASCII CR }

1:u 1 DLE = 16;         { BLANK COMPRESSION CODE }

1:u 1 NAME_LENG = 23;   { Number of characters in a full file name}

1:u 1 SNAPING = 0;      { Swapping segment status}

1:u 1 P_LOCKED = -1;    { Position locked segment status}

1:u 1 STACK_SLOP = 40;  { Number of words of temp for procedure stack}

1:u 1 MEM_LINK_SIZE = 4; { Number of words in heap record}

sys_error = 0; { Unknown system error}

proc_error= 3; { Unknown procedure error}

stk_error = 4; { Stack overflow error}

sys_io_error = 9; { System I/O error}

halt_error = 14; { Programmed halt }

heap_error= 15; { Heap operation error}

seg_fault = 128; { Segment fault }

stk_fault = 129; { Stack fault }

heap_fault= 130; { Heap operation fault }

pool_fault= 131; { Used to consolidate pool after purge}

TYPE

1:u 1

IORSLTWD = (INOERROR,IBADBLOCK,IBADUNIT,IBADMODE,ITIMEOUT,
            ILOSTUNIT,ILOSTFILE,IBADTITLE,INOROOM,INOUNIT,
            INOFILE,INDUPLICATE,INOTCLOSED,INOPEN,IBADFORMAT,
            IBUFVFL, i_writeprot, i_ill_block, i_ill_buf);

1:u 1

(CMDSTATE...SEE GETCMD )

CMDSTATE = (sys_boot, sys_init, sys_halt, UPROGUOK, UPROGUOU, SYSPROG,
            COMPONLY, COMPODGO, LINKANDGO, sys_debug);

(CODE FILES USED IN GETCMD )

SYSFILE = (ASSEMBLER, COMPILER, EDITOR, FILER, LINER);

(ARCHIVAL INFO...THE DATE )

DATEREC = PACKED RECORD

   MONTH: 0..12; { O IMPLIES DATE NOT MEANINGFUL }

   DAY: 0..31;  { DAY OF MONTH }

   YEAR: 0..100 { 100 IS TEMP DISK FLAG }

END ( DATEREC );
UNITNUM = 0.127;

VID = STRING VIDLENG;

DIRENTRY = PACKED RECORD
  DFIRSTBLK: INTEGER; (* FIRST PHYSICAL DISK ADDR *)
  DLASTBLK: INTEGER; (* POINTS AT BLOCK FOLLOWING *)
  CASE DFKIND: FILE_KIND OF
     SECUREDIR, ...
   UNYPEDFILE: (ONLY IN DIR[0]...VOLUME INFO)
     (filler_1: 0..2048; (13 bits)
     DVID: VID; (* NAME OF DISK VOLUME)
     DEOVBLK: INTEGER; (* LASTBLK OF VOLUME *)
     DNUMFILES: DIRENTRY; (* NUM FILES IN DIR *)
     DLOADTIME: INTEGER; (* TIME OF LAST ACCESS *)
     DLASTBOOT: DATEREC; (* MOST RECENT DATE SETTING *)
     XDSKFILE, CODEFILE, TEXTFILE, INFOFILE,
     DATAFILE, GRAFFILE, FOTOFILE, SECURED, SUBSVOL:
     (filler_2: 0..1024; (12 bits)
     status: BOOLEAN; (* Filer kludge temporary *)
     DTID: TID; (* TITLE OF FILE *)
     DLASTBYTE: 1..FBLKSIZE; (* NUM BYTES IN LAST BLOCK *)
     DACCESS: DATEREC; (* LAST MODIFICATION DATE *)
   END (* DIRENTRY *)

DIRP = "DIRECTORY;"

DIRECTORY = ARRAY [DIRENTRY] OF DIRENTRY;

CLOSETYPE = (CNORMAL, CLOCK, CPURGE, CCRUNCH);

WINDOW = "WINDOW;"

WINDOW = PACKED ARRAY [0..0] OF CHAR;

FIB = "FIB;"

FIB = RECORD
  FWINDOW: WINDOW; (* USER WINDOW...F", USED BY GET-PUT *)
  FEFOF, FEOLN: BOOLEAN;
  FSTATE: (FJANDW, FNEECHAR, FGOTCHAR);
  FRESIZE: INTEGER; (* IN BYTES...0+BLOCKFILE, 1+CHARFILE *)
  F_LOCK: SEMAPHORE;
  CASE FISOPEN: BOOLEAN OF
    TRUE: (FISBLKID: BOOLEAN; (* FILE IS ON BLOCK DEVICE *)
    UNIT: UNITNUM; (* PHYSICAL UNIT # *)
    FVID: VID; (* VOLUME NAME *)
    FPREPONT, (* # TIMES F' VALID W/O GET *)
114 2 14u 1 FNXTBLK,  (NEXT REL BLOCK TO IO)
115 2 14u 1 FMAXBLK: INTEGER;  (MAX REL BLOCK ACCESSED)
116 2 14u 1 FMODIFIED: BOOLEAN;  (PLEASE SET NEW DATE IN CLOSE)
117 2 14u 1 FHEADER: DIRENTRY;  (COPY OF DISK DIR ENTRY)
118 2 14u 1 CASE FSOFTBUF: BOOLEAN OF (DISK GET-PUT STUFF)
119 2 14u 1 TRUE: FNXBYTE,FMAXBYTE: INTEGER;
120 2 14u 1 FBUFSIZE: BOOLEAN;
121 2 14u 1 FBUFSIZE: ARRAY [0..FBLKSIZE] OF CHAR));
122 2 14u 1 END (FIB);
123 2 14u 1
124 2 14u 1 (USER WORKFILE STUFF)
125 2 14u 1
126 2 14u 1 INFOREC = RECORD
127 2 14u 1 SYMIBP,SYMPBP,FIBP;  (WORKFILES FOR SCRATCH)
128 2 14u 1 ERRSYM,ERRBLK,ERRNUM: INTEGER;  (ERROR STUFF FOR EDIT)
129 2 14u 1 SLOWTERM,STUPID: BOOLEAN;  (STUDENT PROGRAMMER ID!!)
130 2 14u 1 ALTMODE: CHAR;  (WASHOUT CHAR FOR COMPILER)
131 2 14u 1 GOTSYS,GOCTYPE: BOOLEAN;  (TITLES ARE MEANINGFUL)
132 2 14u 1 WORKVID,SYMVID,CODEVID: VID;  (PERM&CUR WORKFILE VOLUMES)
133 2 14u 1 WORKTID,SYNRTID,COTETID: TID;  (PERM&CUR WORKFILES TITLE)
134 2 14u 1 END (INFOREC);
135 2 14u 1
136 2 14u 1
137 2 14u 1 int_p = 'INTEGER;
138 2 14u 1 tib_p = 'tib;
139 2 14u 1 sib_p = 'sib;
140 2 14u 1 e_rec_p = 'e_rec;
141 2 14u 1 e_vec_p = 'e_vec;
142 2 14u 1 sem_p = 'sem;
143 2 14u 1 mscwp = 'mscwp;
144 2 14u 1 p_mem_chunk = 'mem_chunk;
145 2 14u 1 vip = 'vinfo;
146 2 14u 1
147 2 14u 1 byte = 0..255;
148 2 14u 1 mem_chunk = ARRAY [0..0] OF INTEGER;  (Accessed $R$)
149 2 14u 1 alpha = PACKED ARRAY [0..73] OF CHAR;  (Identifier name)
150 2 14u 1
151 2 14u 1 mem_ptr = RECORD
152 2 14u 1 CASE INTEGER OF
153 2 14u 1 0 : (m : 'mem_link);
154 2 14u 1 1 : (i : int_p);
155 2 14u 1 2 : (c : p_mem_chunk);
156 2 14u 1 J : (t : INTEGER);
157 2 14u 1 END (of mem_ptr);
158 2 14u 1
159 2 14u 1 mem_link = RECORD
160 2 14u 1 avail_list : mem_ptr;
161 2 14u 1 n_words : INTEGER;
162 2 14u 1 CASE BOOLEAN OF
163 2 14u 1 TRUE : (last_avail,
164 2 14u 1 prev_mark : mem_ptr);
165 2 14u 1 END (of mem_link);
166 2 14u 1
167 2 14u 1 vinfo = record
168 2 14u 1 segunit : integer;
169 2 14u 1 segvid : vid;
170 2 14u 1 end (of vinfo);
poolptr = 'pooldes;
sib = RECORD
  seg_pool : poolptr;
  seg_base : mem_ptr;
  seg_refs : INTEGER;  (* # of active calls *)
  timestamp : INTEGER;  (* Memory swap priority *)
  link_count:INTEGER;  (* Number of Links to SIB *)
res_sibs : RECORD  (* Memory residency status *)
  residency: p_locked..MAX_INT;  (* memory residency status *)
  seg_name : alpha;  (* Segment name *)
  seg_len : INTEGER;  (* # of words in segment *)
  seg_addr : INTEGER;  (* Disk address of segment *)
  vol_info : viq;  (* Disk unit and vol id of segment *)
  data_size : INTEGER;  (* Number of words in data segment *)
  next_sib,  (* Pointer to next sib *)
  prev_sib : sib_p;  (* Pointer to previous sib *)
  CASE BOOLEAN OF
  | TRUE : (next_sort : sib_p);  (* Scratch area *)
  | FALSE : (new_loc : mem_ptr)
  END (of res_sibs);
  mtype:integer;
END (of sib);

e_vec = RECORD  (* Environment vector *)
  vect_length : INTEGER;
  map : ARRAY [1..1] OF e_rec_p;  (* Accessed #R-- *)
END (of e_vec);
e_rec = RECORD  (* Environment record *)
  env_data : mem_ptr;  (* Pointer to base data segment *)
  env_vec : e_vec_p;  (* Pointer to environment vector *)
  env_sib : sib_p;  (* Pointer to associated segment *)
  CASE BOOLEAN OF
  | TRUE : (link_count : INTEGER);
  | next_rec : e_rec_p;
  END (of e_rec);

mscw = PACKED RECORD  (* Mark stack control *)
  ms_stat : mscw_p;  (* Lexical parent pointer *)
  ms_dynl : mscw_p;  (* Ptr to caller's mscw *)
  ms_ipc : INTEGER;  (* Byte inx in return code seg *)
  ms_env : e_rec_p;  (* Environment of caller code *)
  ms_proc: integer;  (* Proc # of caller *)
END (of mscw);

tib = RECORD  (* Task information block *)
  regs : PACKED RECORD
  wait_q : tib_p;  (* Queue link for semaphores *)
  prior : byte;  (* Task's cpu priority *)
  flags : byte;  (* State flags...not defined yet *)
  sp_low : mem_ptr;  (* Lower stack pointer limit *)
  sp_upr : mem_ptr;  (* Upper limit on stack *)
  sp : mem_ptr;  (* Actual top of stack pointer *)
  mp : mscw_p;  (* Active procedure MScw ptr *)
  reserved : integer;
  ipc : INTEGER;  (* Byte ptr in current code seg *)
228  2  i1u  1    env : e_rec_p;  ( Ptr to current environment )
229  2  i1u  1    procnum:byte;
230  2  i1u  1    tibioresult:byte;
231  2  i1u  1    hang_p : sem_p;   ( Which task is waiting on )
232  2  i1u  1    m_depend : INTEGER; ( Reserved for interpreter )
233  2  i1u  1        ( initiated to 0 when process started )
234  2  i1u  1    END ( of regs ) ;
235  2  i1u  1    main_task : BOOLEAN ;
236  2  i1u  1    start_mscv : mscv_p ;
237  2  i1u  1    END ( of tib ) ;
238  2  i1u  1
239  2  i1u  1    sem = RECORD ( Semaphore format )
240  2  i1u  1        sem_count : INTEGER; ( Number outstanding signals )
241  2  i1u  1        sem_wait_q : tib_p  ( List of tasks waiting on sem )
242  2  i1u  1    END ( of sem ) ;
243  2  i1u  1
244  2  i1u  1    fault_message = RECORD
245  2  i1u  1        fault_tib : tib_p ;
246  2  i1u  1        fault_e_rec : e_rec_p ;
247  2  i1u  1        fault_words : INTEGER;  
248  2  i1u  1        fault_type : seg_fault..pool_fault;
249  2  i1u  1    END ( of fault_message ) ;
250  2  i1u  1
251  2  i1u  1    fulladdress = array[0..1] of integer; ( 32 bits )
252  2  i1u  1
253  2  i1u  1    utableentry = record
254  2  i1u  1        uvid : vid;   ( volume id for unit )
255  2  i1u  1        case uisblkd : boolean of 
256  2  i1u  1            true : (ueovblk : integer; 
257  2  i1u  1            uphysvol : unitnum; 
258  2  i1u  1            ublkoff : integer; 
259  2  i1u  1            upvid : vid)
260  2  i1u  1    end (utable) ;
261  2  i1u  1    utable = array [unitnum] of utableentry ; ( 0 not used )
262  2  i1u  1
263  2  i1u  1    ( SYSTEM COMMUNICATION AREA )
264  2  i1u  1    ( SEE INTERPRETERS...NOTE )
265  2  i1u  1    ( THAT WE ASSUME BACKWARD )
266  2  i1u  1    ( FIELD ALLOCATION IS DONE )
267  2  i1u  1    syscomrec = record ( modified for IV.1 )
268  2  i1u  1        iorslt : iorsltwd;   ( RESULT OF LAST IO CALL )
269  2  i1u  1        rsrvd1 : integer;
270  2  i1u  1        sysunit : unitnum;  ( PHYSICAL UNIT OF BOOTLOAD )
271  2  i1u  1        rsrvd2 : integer;
272  2  i1u  1        gdirp : dirp;  ( GLOBAL DIR POINTER,SEE VOLSEARCH )
273  2  i1u  1        fault_sem : RECORD
274  2  i1u  1            real_sem, message_sem : SEMAPHORE;
275  2  i1u  1            message : fault_message;
276  2  i1u  1        END ( of fault_sem ) ;
277  2  i1u  1    ( starting unit number for subsidiary volumes)
278  2  i1u  1    subsidstart : unitnum;
279  2  i1u  1    rsrvd3 : integer;
280  2  i1u  1    spool_avail : boolean;
281  2  i1u  1    poolinfo : record
282  2  i1u  1        pooloutside : boolean;
283  2  i1u  1        poolsize : integer;
284  2  i1u  1        poolbase : fulladdress;
resolution : integer;
end;

timestamp : integer;
unitable : "utable;
unitdivision : packed record
  serialmax : byte; (number of user serial units)
  subsidmax : byte; (max number of subsid vols)
end;

expansion : packed record
  insertchar,deletchar:char;
  expand1,expand2:integer;
end;

pmachver : (pre_iu_i, iv_i, post_iu_i);
realsize : integer;

MISCINFO: PACKED RECORD
  NOBREAK,STUPID,SLOWTERM,
  HASXYCRT,HASCCRT,HAS8510A,HASCLOCK: BOOLEAN;
  USERKIND:(NORMAL,AQUIZ,BOOKER,PQUIZ)
END;

CRTCNTYPE: INTEGER;
CRTCNTL: PACKED RECORD
  RLF,NDFS,ERASECOL,ERASEE0S,HOME,ESCAPE: CHAR;
  BACKSPACE: CHAR;
  FILLCOUNT: 0..255;
  CLEARSCREEN, CLEARLINE: CHAR;
  PREFIRED: PACKED ARRAY [0..8] OF BOOLEAN
END;

CRTCINFO: PACKED RECORD
  WIDTH,HEIGHT: INTEGER;
  RIGHT,LEFT,DOWNN,UP: CHAR;
  BADCH,CHARDEL,STOP,BREAK,FLUSH,EOF: CHAR;
  ALTMODE,LINEDEL: CHAR;
  alphabet, char_mask, ETX, PREFIX: CHAR;
  PREFIRED: PACKED ARRAY [0..15] OF BOOLEAN
END

END (SYSCOM);

MISCINFOREC = RECORD
  MSYSCOM: SYSCOMREC
END;

poolads = record
  poolbase : fulladdress;
  poolsize : integer;
  minoffset : memptr;
  maxoffset : memptr;
  resolution : integer; (in bytes)
  poolhead : sibp;
  permisb : sibp;
  extended : boolean;
end;
bytearray = packed array [0..0] of byte;

VAR
  SYSCOM: "SYSCOMREC;
  ( MAGIC PARAM...SET UP IN BOOT )
  GFILES: ARRAY [0..5] OF FIBP;
  ( GLOBAL FILES, 0=INPUT, 1=OUTPUT )
USERINFO: INFOREC;  \{ WORK STUFF FOR COMPILER ETC \}
EMPTYHEAP: 'integer;  \{ HEAP MARK FOR MEM MANAGING \}
main: task : tib.p;  \{ taskinfo block of op syš prog \}
Has_PM: BOOLEAN;  \{ performance monitor in use \}
INPUTFIB,OUTPUTFIB,SYSTEM: FIBP;  \{ CONSOLE FILES...FILES ARE COPIES \}
SVVID,DKVID: VID;  \{ SYSUNIT VALID & DEFAULT VALID \}
THEDATE: DATEREC;  \{ TODAY...SET IN FILER OR SIGN ON \}
STATE: CMOSTATE;  \{ FOR GETCOMMAND \}
heap_info: RECORD ( Stuff for heap management )
  Lock: SEMAPHORE;
  top_mark,
  heap_top: mem_ptr;
END ( of heap_info );
task_info: RECORD ( Stuff for task management )
  Lock,
  task_done: SEMAPHORE;
  n_tasks: INTEGER;
END ( of task_info );
IPOT: ARRAY CO..4] OF INTEGER;  \{ INTEGER POWERS OF TEN \}
FILLER: STRING[111];  \{ NULLS FOR CARRIAGE DELAY \}
DIGITS: SET OF '0'..'9';
PL: STRING;
maxunit: unitnum;
FILENAME: file_table;
junk : processid;
fault_sem: RECORD
  real_sem, message_sem: SEMAPHORE;
message : fault_message;
END (of fault_sem);
unit_list: e_rec.p;
user_env_vec,
sys_env_vec: e_vec.p;
dir_lock : SEMAPHORE;  \{ for volume and directory exclusion \}
ininerr: boolean;  \{ set when processing execution errors
  so that file system locks are not enforced \}
fliplignt: boolean;  \{ true when gdirp contains a flipped directory \}
debugging: boolean;
permlist: mpatr;  \{ List of "permanent" new's \}
suspects: set of unitnum;
atsysprompt: boolean;  \{ true if at system prompt \}
userlib: fullid;  \{ Library text file \}
syslist: e_rec.p;  \{ Last unit placed on unit_list that is a
  system unit; all subsequent units added
  to the unit_list are user units \}
nofit: boolean;  \{ set to true if dumpmsgs can't get any more room \}
codepool: poolptr;
resolving_break: boolean;  \{ true if in debugger and a code segment
  must be read in without breakpoints \}
procedure exec_error(bad_e_rec.p:e_rec.p; n_words,err:integer);
procedure loadsseg(segrec:e_rec.p);
procedure rlocseg(segrec:e_rec.p);
function ptr_add ( p : mem_ptr; n_words : integer ) : int_p;
function ptr_sub ( p_0, p_1 : mem_ptr ) : integer;
function ptr_less ( p_1, p_2 : mem_ptr ) : boolean;
function ptr_gtr ( p_1, p_2 : mem_ptr ) : boolean;
function ptr_geq ( p_1, p_2 : mem_ptr ) : boolean;
procedure print(s:string);
procedure printint(i:integer);
procedure writeln(s:string);
procedure checkunit(lvid:vid; unit:unitnum);
procedure moveseg(segsib:sibp; srcpool:poolptr; srcoffset:memptr);

End of Compilation.
Using COMMANDI

2  2  1:u  1  type bigstring=string[255];
3  2  1:u  1  var havechain,inredirect,outredirect,monitoropen,inmonitor:boolean;
4  2  1:u  6  function redirect(command:bigstring):boolean;
5  2  1:u  1  procedure exception(stopchainning:boolean);
6  2  1:u  1  procedure chain(command:bigstring);
7  2  1:u  1
8  2  1:u  1  procedure initcommand;
9  2  1:u  1  procedure startmonitor;
10  2  1:u  1  procedure stopmonitor(saveit:boolean);
11  2  1:u  1  procedure getchainline(var command:bigstring);
12  2  1:u  1

End of compilation.
Using ERRORHAN

2 2 i:u 1
3 2 i:u 1
4 2 i:u 1 type
5 2 i:u 1 drive_range = 4..127;
6 2 i:u 1
7 2 i:u 1 procedure set_error_line (line : integer);
8 2 i:u 1 procedure set_user_message (drive : drive_range; mess : string);
9 2 i:u 1

end of Compilation.
Using SCREENOP

const
  sc_fill_len = 11;
  sc_eol = 13;

type
  sc_chset = set of char;
  sc_msdRec = packed record
    height, width : 0..255;
    can_break, slow, xy_crt, lc_crt,
    can_upscroll, can_downscroll : boolean;
  end;

  sc_date_rec = packed record
    month : 0..12;
    day : 0..31;
    year : 0..99;
  end;

  sc_info_type = packed record
    sc_version : string;
    sc_date : sc_date_rec;
    spec_char : sc_chset; {Characters not to echo}
    misc_info : sc_msdRec;
  end;

  sc_long_string = string[255];

  sc_scrn_command = (sc_home, sc_eras_s, sc_erase_eol, sc_clear_line,
    sc_clear_scrn, sc_up_cursor, sc_down_cursor,
    sc_left_cursor, sc_right_cursor);

  sc_key_command = (sc_backspace_key, sc_dct_key, sc_eof_key, sc_etx_key,
    sc_escape_key, sc_del_key, sc_up_key, sc_down_key,
    sc_left_key, sc_right_key, sc_not_legal, sc_insert_key,
    sc_delete_key);

  sc_choice = (sc_get, sc_give);

  sc_window = packed array [0..0] of char;

  sc_tx_port = record
    rov, col,             { screen relative}
    height, width,       { size of txport (zero based)}
    cur_x, cur_y : integer;
    (cursor positions relative to the txport )
  end;

var
  sc_port : sc_tx_port;

procedure sc_use_info(do_what: sc_choice; var t_info: sc_info_type);
procedure sc_use_port(do_what: sc_choice; var t_port: sc_tx_port);
procedure sc_erase_eol(x,y:integer);
procedure sc_left;
procedure sc_right;
procedure sc_up;
procedure sc_down;
procedure sc_getc_ch(var ch:char; return_on_match:sc_chset);
procedure sc_clr_screen;
procedure sc_clr_line (y:integer);
procedure sc_home;
procedure sc_eras_eos (x,line:integer);
procedure sc_goto_xy(x, line:integer);
procedure sc_clr_cur_line;
function sc_find_x:integer;
function sc_find_y:integer;
function sc_scrn_has(what:sc_scrn_command):boolean;
function sc_has_key(what:sc_key_command):boolean;
function sc_map_crt_command(var k_ch:char):sc_key_command;
function sc_prompt(line :sc_long_string; x_cursor,y_cursor,x_pos,
where:integer; return_on_match:sc_chset;
no_char_back:boolean; break_char:char):char;
function sc_check_char(var buf:sc_window; var buf_index,bytes_left:integer)
flush:boolean;
function sc_space_wait(flush:boolean):boolean;
procedure sc_init;
Using SYSINFO

Type SI_Date_Rec = Packed Record
  Month : 0..12;
  Day : 0..31;
  Year : 0..99;
End; (SI_Date_Rec)

Procedure SI_Code_Vid (Var SI_Vol : String);
  (Returns name of volume containing current workfile code)

Procedure SI_Code_Tid (Var SI_Title : String);
  (Returns title of current workfile code)

Procedure SI_Text_Vid (Var SI_Vol : String);
  (Returns name of volume containing current workfile text)

Procedure SI_Text_Tid (Var SI_Title : String);
  (Returns title of current workfile text)

Function SI_Sys_Unit : Integer;
  (Returns number of bootload unit)

Procedure SI_Get_Sys_Vol (Var SI_Vol : String);
  (Returns system volume name)

Procedure SI_Get_Pref_Vol (Var SI_Vol : String);
  (Returns prefix volume name)

Procedure SI_Set_Pref_Vol (SI_Vol : String);
  (Sets prefix volume name)

Procedure SI_Get_Date (Var SI_Date : SI_Date_Rec);
  (Returns current system date)

Procedure SI_Set_Date (Var SI_Date : SI_Date_Rec);
  (Sets current system date)

(*-----------------------------------------------------------------------*)

End of Compilation.
Using FILEINFO

Type F_File_Type = file;
F_Date_Record = Packed Record
   Month : 0..12;
   Day : 0..31;
   Year : 0..100;
End; (F_Date_Record)

Function F_Open (var fid: F_File_Type): boolean;
(* returns true if the file is open and false if not open *)

Function F_Length (var Fid: F_File_Type): integer;
(Returns the length of the file attached to the Fid identifier. If the file is not opened result is returned as zero)

Function F_Unit_number (var Fid: F_File_Type): integer;
(Returns the unit containing the file attached to the Fid identifier. If there is no file opened to Fid, the function result is zero.)

Procedure F_Volume (var Fid: F_File_Type;
   Var File_Volume: String);
(Returns the name of the volume containing the file attached to the Fid identifier. If there is no file opened to Fid, the file_volume is set to a null string.)

Procedure F_File_Title (var Fid: F_File_Type;
   Var File_Title: String);
(Returns the title (with suffix) of the file attached to the Fid identifier. If there is no file opened to Fid, the File_title is set to the null string.)

Function F_Start (var Fid: F_File_Type): integer;
(Returns the block number of the first block of the file attached to the Fid identifier. If there is no file opened to Fid, the function result is returned is zero.)

Function F_is Blocked (var Fid: F_File_Type): Boolean;
(Returns a boolean that is TRUE if the file attached to the Fid identifier is located on a block-structured unit. If there
57 2 1:u 1  is no file opened for the Fid or if the device is not block structured
58 2 1:u 1  , the function result is set to false.)
59 2 1:u 1
60 2 1:u 1
61 2 1:u 1  Procedure F_Date (Var Fid       : F_File_Type;
62 2 1:u 0  Var File_Date : F_Date_REC);
63 2 1:u 1
64 2 1:u 1  (Returns a record indicating the last access date for the file
65 2 1:u 1  attached to the Fid identifier. If there is no file opened to
66 2 1:u 1  Fid, the File_Date is unchanged.)
67 2 1:u 1
68 2 1:u 1

End of Compilation.
Using WILD

Type

D_PatRecP = *D_PatRec;

D_PatRec = Record

  CompPos,          ( starting position of pattern in subject string )
  CompLen,         ( length of pattern in subject string )
  WildPos,         ( starting position of pattern in wild string )
  WildLen : Integer; ( length of pattern in wildcard string )
  Next : D_PatRecP; ( next pattern )
  End;          ( D_PatRec )

Function D_Wild_Match(Wild, Comp : String; Var PPtr : D_PatRecP;

PPtr : Boolean) : Boolean;

( Compares two strings (one containing wildcards) and returns true if they
match. Includes information about pattern matching that occurred if re-
quested (by PInfo) )

end of Compilation.
Using DIRINFO

uses
  (**U WILD.CODE*) wild;

Type
  D_DateRec = Packed Record
    Month : 0..12;
    Day : 0..31;
    Year : 0..100;
  End;

D_NameType = (D_Vol, D_Code, D_Text, D_Data, D_SVol, D_Temp, D_Free);

D_Choice = Set of D_NameType;

D_ListP = 'D_List;

D_List = Record
  D_Unit : Integer;  (* Unit # of entry *)
  D_Volume : String[7];  (* volume name of unit *)
  D_VPatt : D_PattRecP;  (* volume pattern info *)
  D_NextEntry : D_ListP;  (* Next entry in list *)
  Case D_IsBlkD : Boolean Of
    True : (D_Start,
      (* Starting block of entry *)
      D_Length : Integer;  (* Length (in blocks) of entry *)
      Case D_Kind : D_NameType Of
        0 : 0;
        ( Everything but D_Free )
        D_Vol,
        D_Temp,
        D_Code,
        D_Text,
        D_Data,
        D_SVol : (D_Title : String[15];  (* File name *)
          D_FPatt : D_PattRecP;  (* name pattern info *)
          D_Date : D_DateRec;  (* file date *)
          Case D_NameType of
            0 : 0;
            ( # of files on vol )
            D_Vol : (D_NumFiles : Integer));
    End;
  End;

D_Result = (D_Okay,
  (* Mission accomplished *)
  D_Not_Found,  (* Couldn't find name and/or type *)
  D_Exists,  (* Name already exists; no name change made *)
  D_Name_Error,  (* Illegal string passed *)
  D_Off_Linel,  (* Volume not on line *)
  D_Other);  (* Miscellaneous error *)

Function D_Dir_List(D_Name : String; D_Select : D_Choice;
  Var D_Ptr : D_ListP; D_PInfo : Boolean) : D_Result;

(* Creates pointer to list of names of specified NameTypes
  (D_Select), matching specified D_Name (wildcard characters allowed). In- 
  cludes information about pattern matching that occurred if requested 
  (by D_Pinfo) *)
Function D_Scan_Title(D_Name : String; Var D_VolID, D_TitleID : String;
    Var D_Type : D_NameType; Var D_Segs : Integer) : D_Result;

{ Parses D_Name }

Function D_Change_Name(D_OldName, D_NewName : String; D_RemOld : Boolean) : D_Result;
{ Changes file name in D_OldName to name in D_NewName, removing already
existing files of name in D_NewName if D_RemOld is set }

Function D_Change_Date(D_Name : String; D_NewDate : D_DateRec;
    D_Select : D_Choice) : D_Result;
{ Changes date of directory or file name in D_Name to date specified by
D_NewDate. D_Name may contain wildcards }

Function D_Rem_Files (D_Name : String; D_Select : D_Choice) : D_Result;
{ Removes file of specified name (wildcards allowed) }

Procedure D_Lock;

Procedure D_Release;
{ Provide means to limit use of DirInfo routines to one task at a time
in multi-tasking environments }

Function D_Krunch (D_Unit,
    D_Block : Integer) : D_Result;
{ Collects all unused space on a volume around D_Block. This unit must
not be in use when this operation is performed. }

Function D_Mount (D_File_Name : String) : D_Result;
Function D_DisMount (D_Vol_Name : String) : D_Result;
{ Provides a means of mounting and dismounting subsidiary volumes.
Wild cards may be used. }

*****************************************

-end of Compilation.
program windowdisp;

Using WINDOWMA

(Window Manager for the UCSD p-System)
(Windows are displayed as rectangular areas on the screen, bordered by
a frame and optionally headed by a heading. Each window has its own
size, screen location, text area, cursor and status information.
Each window may be written into and will scroll independently, and
may be cleared, moved, changed in size, etc. by a user's program.

During any input operation the user may escape into 'Window Manager
Mode' (and subsequently return to 'Input Mode', to complete the input).
In Window Manager Mode the Window Manager uses a special cursor which
is independent of any window. This cursor is used to indicate screen
position parameters to the Hide, Show, Alter, Move and Kill commands.
Whether or not a particular command may be applied to a particular
window is controlled by the user's program.

CONST WVersion='10-Dec-82';
NoWindow=0;
MaxWindow=10;

TYPE Window=NoWindow..MaxWindow;
WindowOptions=(CanHide,CanMove,CanAlter,CanKill,
HasHeading,CanScroll,CanPan);
WindowAttributes=SET Of WindowOptions;

{Initialisation Routines}

PROCEDURE WStartup;
{Called by *SYSTEM.STARTUP to REALLY initialise}
{If window manager is placed in *SYSTEM.PASCAL then you MUST supply
a *SYSTEM.STARTUP that calls WStartup. Thereafter any program which
uses the window manager should initialise via WInit, the effect of which
is to repaint the screen as it was when the last using program terminated.
If Window Manager is not in *SYSTEM.PASCAL then use WStartup always)

PROCEDURE WInit;
{Initialise Window Manager System}
{just redisplay all windows}
{if Manager is a system unit, all windows survive program changes}
{Routines to create, alter, show, clear, hide and dispose of windows}

FUNCTION WNew(WatX,WatY,WSizeX,WSizeY:INTEGER;
WControls:WindowAttributes;
WHeading:STRING):Window;
{Get new window}

PROCEDURE WAlter(W:Window;
WatX,WatY,WSizeX,WSizeY:INTEGER;
WControls:WindowAttributes;
WHeading:STRING);
{Alter existing window}
56 2 1:u 1 {Wax, WY, SizeX, SizeY -ve means do not alter}
57 2 1:u 1 {WControls replaces existing window attributes}
58 2 1:u 1 {Window must not be in show when WALTER called}
59 2 1:u 1
60 2 1:u 1 PROCEDURE WShow(W:Window);
61 2 1:u 1 {Display window and set it as "current" one}
62 2 1:u 1
63 2 1:u 1 PROCEDURE WClearAndShow(W:Window);
64 2 1:u 1 {Clear window, then "Show" it}
65 2 1:u 1
66 2 1:u 1 PROCEDURE WHide(W:Window);
67 2 1:u 1 {Remove window from screen - it is not disposed of}
68 2 1:u 1
69 2 1:u 1 PROCEDURE WDispose(W:Window);
70 2 1:u 1 {Dispose of old window}
71 2 1:u 1 {Window must not be in show when WDispose called}
72 2 1:u 1
73 2 1:u 1 {The following procedures all apply to the "current" last shown window}
74 2 1:u 1
75 2 1:u 1 PROCEDURE WClear;
76 2 1:u 1 {Clear Window}
77 2 1:u 1
78 2 1:u 1 PROCEDURE WCLrEOL;
79 2 1:u 1 {Clear remainder of current Line}
80 2 1:u 1
81 2 1:u 1 PROCEDURE WCLrEOS;
82 2 1:u 1 {Clear remainder of window}
83 2 1:u 1
84 2 1:u 1 PROCEDURE WGoToXY(X,Y:INTEGER);
85 2 1:u 1 {Set Window cursor to X,Y}
86 2 1:u 1 {X,Y are relative to top left of window - base of 0, excluding heading}
87 2 1:u 1
88 2 1:u 1 PROCEDURE WWriteCh(Ch:CHAR);
89 2 1:u 1 {Write Ch at cursor position in window}
90 2 1:u 1 {Non printable chars map to bell}
91 2 1:u 1
92 2 1:u 1 PROCEDURE WWriteStr(Str:STRING);
93 2 1:u 1 {Write Str at cursor position in window}
94 2 1:u 1 {MUST NOT CONTAIN NON PRINTABLE CHARs}
95 2 1:u 1
96 2 1:u 1 PROCEDURE WWriteInt(Int,Width:INTEGER);
97 2 1:u 1 {Write Int at cursor posn in window}
98 2 1:u 1 {Equivalent to WRITE(Int,Width) in Pascal}
99 2 1:u 1 {Width may be 0 (or -ve) to mean as narrow as possible}
100 2 1:u 1
101 2 1:u 1 PROCEDURE WWriteLn;
102 2 1:u 1 {Write newline at cursor position in window}
103 2 1:u 1 {If cursor goes below base of window, window is cleared}
104 2 1:u 1
105 2 1:u 1 PROCEDURE WReadCh(VAR Ch:CHAR;Echo:BOOLEAN);
106 2 1:u 1 {Get character from keyboard}
107 2 1:u 1 {Window functions can only take place within WReadCh}
108 2 1:u 1 {Any non window function Ch is returned to user}
109 2 1:u 1 {Echo is controlled by user - non printable chars echo as bell}
110 2 1:u 1 {Other Window Reading Procedures - below - use WReadCh}
111 2 1:u 1
112 2 1:u 1 PROCEDURE WReadLnStr(VAR Str:STRING);
PROCEDURE WReadLnInt(VAR Int:INTEGER);
{Get an integer from keyboard - echoed}
PROCEDURE WReadLn(Echo:Boolean);
{Read up to next newline from keyboard}
PROCEDURE WReadLn(Echo:Boolean);
{Read up to next newline from keyboard}

{the following functions and procedures are utilities on windows}

FUNCTION WInWindow(X,Y:INTEGER):Window;
{Returns window in which position X,Y occurs - NoWindow if none}

FUNCTION WCharXY(X,Y:INTEGER; W:Window):CHAR;
{Return Ch under screen position X,Y in W}

FUNCTION WCharXY(X,Y:INTEGER; W:Window);
{Space returned if X,Y not in Window, or NoWindow}

PROCEDURE WXY(VAR X,Y:INTEGER);
{Get Coordinates of window manager cursor - in window coordinates}

PROCEDURE WCxyXY(X,Y:INTEGER);
{Get coordinates of window manager cursor - in window coordinates}

FUNCTION WCCurrentWindow:Window;
{Return Current Window - one Last shown - may be NoWindow}

{HISTORY}

Copyright: Austin Tate, ERCC. All rights reserved.
This program may be used for non-commercial purposes
by users of the UCSD p-System provided that this
copyright notice appears in the source. Enquiries for
other uses should be directed to the copyright owner.

The Window Manager was originally written in March 1981 by
Austin Tate, ERCC as a demonstration for a course on Office
Systems and Advanced Personal Computers.

It was subsequently modified by Chris Lee while at INMOS up
to 10-Feb-82. The major changes he made were:

I/O optimisation - all I/O is delayed as long as possible, and
is done in as large units as possible, via UNITWRITE (asynch!!!),
and isn't done at all if what we want is on the screen already.
See routines flushoutput and repaint. Repaint certainly pays
its way (try moving and altering windows with the original and
this version), flushoutput almost certainly doesn't - it makes
lines zap out in one swell foop, but stops the dreaded dots.

Window Functions - on the operator interface are all handled
inside the window manager. During input use ESC to toggle between
window manager and input modes (and notice the cursor change when
you do). See WindowFunction. Notice that screen position parameters
are signalled by moving the cursor and typing ESC or <space> (known
as a 'mark' in Window Manager Mode). Eg, to move a window type
ESC
-- to enter window mode

<move cursor into window to be moved>

M or m
-- to request a move

<move cursor to new top left corner position>

<space> or ESC
-- the window moves at this point

<space> or ESC
-- to return to input mode

NB at point ----> WindowFunction calls itself recursively. You
can nest window functions (and get into a real mess) if you want).
The interface to WNew has changed to allow the programmer to
control what WindowFunctions the user may apply to a window.

Frames = are drawn by characters. There are CONSTs for the four
corners and four sides so if you have forms drawing chars you
should be able to make things look pretty

Scrolling and Panning - misc minor changes. Scroll by one third
of window depth rather than one line. Only re-pan at input time.
Both these mods are designed to eliminate unnecessary I/O and
repaing.

To improve efficiency WMWriteStr assumes that only printable
chars are in the string.

This version was produced during experimentation into window
management. Hence the code hasn't been beautified and bugs
may be present.

The Window Manager as modified by Chris Lee was then altered in
some minor respects by Austin Tate prior to release to the

uses (#u #5:wiler.code)windowmanager;
210 2 :O begin end.

End of Compilation.