ADDENDUM

TO THE

USERS' MANUAL SUPPLEMENT

VERSION IV

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

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, ADCBIM, SUBBIM, SBBBIM, CMPBIM, 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, DECMB, MULMB, IMULMB, DIVMB, IDIVMB, NOTMB, NEGMB, ROLMB, RORMB, 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 INB DX.

In the two-operand forms of IN and OUT, the order of the operands is not important; thus OUT ac,DX or OUT 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 MOVSB and MOVSW, CMPSB and CMPSW, SCASB and SCASW, 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, MOVSB/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:AVALUE
```

is equivalent to

```
  SEG   ES   MOV   AX,AVALUE
```

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

a. The mnemonics CALLL, 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     exprl,expr2
```

where exprl 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

___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, FADDQ, 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 parenenthesis then illegality
77: Expected absolute expression
78: Both operands connot 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: MOVBIM 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..7FFFH generate the absolute short address mode. Absolute short addresses in the range F8000H..FFFFFH cannot be generated.
>
> Absolute addresses in the range 8000H..FFFFH and external references generate the absolute long address mode. Absolute addresses above FFFFH 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, DBcc, 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, ROL (memory), ROR (memory), ROXL (memory) ROXR (memory), RTE, RSR, RTS, SBCD, Scc, 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   |                              |B    |                           |  U  |
|o T |    240 Block Virtual         |o T  |    240 Block Virtual      |  n  |
|o r |         Floppy               |o r  |         Floppy            |  u  |
|t a |     Disk Volume One          |t a  |     Disk Volume Two       |  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 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.

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 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 ther is no special disk for utilities: utility programs are shipped on the second image c 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
DISKCHANGE.CODE     8  5-Dec-80    132    512  Codefile
FINDPARAMS.CODE     9  3-Dec-80    140    512  Codefile
< UNUSED >         91              149
6/6 files<listed/in-dir>, 149 blocks used, 91 unused, 91 in largest
```

```
SYS2:
YALOE.CODE         12  2-Dec-80      6    512  Codefile
BOOTER.CODE         3  4-Dec-80     18    512  Codefile
DISKSIZE.CODE       3  3-Dec-80     21    512  Codefile
SAMPLEGOTO.TEXT     4 17-Nov-78     24    512  Textfile
DECODE.CODE        28  5-Mar-81     28    512  Codefile
COPYDUPDIR.CODE     3  2-Dec-80     56    512  Codefile
MARKDUPDIR.CODE     4  2-Dec-80     59    512  Codefile
XREF.CODE          29  3-Dec-80     63    512  Codefile
RECOVER.G.CODE      8  5-Dec-80     92    512  Codefile
REALOPS.4.CODE     11 21-Aug-81    100    512  Codefile
REALOPS.Z2.CODE     9 24-Nov-81    111    512  Codefile
KERNEL.CODE        65 11-Sep-81    120    512  Codefile
COMMANDIO.CODE      6 20-Jul-81    185    512  Codefile
SCREENOPS.CODE     13 29-Jun-81    191    512  Codefile
LIBRARY.CODE       13  6-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:

```
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
DISKCHANGE.CODE     8  5-Dec-80    132    512  Codefile
FINDPARAMS.CODE     9  3-Dec-80    140    512  Codefile
< UNUSED >         91              149
6/6 files<listed/in-dir>, 149 blocks used, 91 unused, 91 in largest

SYS2:
ASM8086.CODE       60 29-May-81      6    512  Codefile
8086.OPCODES        5  5-May-81     66     52  Datafile
8086.ERRORS        11  8-May-81     71    130  Datafile
8087.FOPS           3  4-Feb-81     82    512  Datafile
XALOE.CODE         12  2-Dec-80     85    512  Codefile
BOOTER.CODE         3  4-Dec-80     97    512  Codefile
DISKSIZE.CODE       3  3-Dec-80    100    512  Codefile
SAMPLEGOTO.TEXT     4 17-Nov-78    103    512  Textfile
DECODE.CODE        28  5-Mar-81    107    512  Codefile
COPYDUPDIR.CODE     3  2-Dec-80    135    512  Codefile
MARKDUPDIR.CODE     4  2-Dec-80    138    512  Codefile
XREF.CODE          29  3-Dec-80    142    512  Codefile
RECOVER.G.CODE      8  5-Dec-80    171    512  Codefile
SCREENTEST.CODE    13  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:

```
86SYS:
SYSTEM.INTERP      19 15-Dec-81      6    512  Datafile
SYSTEM.MISCINFO     1 24-Nov-81     25    194  Datafile
SYSTEM.PASCAL     108 24-Nov-81     26    512  Datafile
SYSTEM.FILER       32 28-Jul-81    134    512  Codefile
SETUP.CODE         28 15-Jul-81    166    512  Codefile
SYSTEM.LIBRARY      7  8-Feb-82    194    512  Codefile
SYSTEM.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

86INT:
COMPRESS.CODE      10  3-Dec-80      6    512  Codefile
INTERPX.CODE       20  6-Oct-81     16    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
TERTBOOT.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 20-Jul-81    206    512  Codefile
SCREENOPS.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

Table of Contents

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  |                           |B  |                           | U |
|o T|    240 Block Virtual      |o T|    240 Block Virtual      | n |
|o r|         Floppy            |o r|         Floppy            | u |
|t a|    Disk Volume One        |t a|    Disk Volume Two        | 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 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:

```
SYSZ800:
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:
YALOE.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
XREF.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.Z2.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
FP0.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     98    512  Codefile
INTER.CPM2.CODE     4  1-Nov-81    102    512  Codefile
INTER.CPM4.CODE     4  1-Nov-81    106    512  Codefile
TERTBOOT.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.ASSMBLER       51  2-Dec-80      6    512  Codefile
Z80.OPCODES         3 20-Dec-78     57     68  Datafile
Z80.ERRORS          8 23-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 >        101              139
7/7 files<listed/in-dir>, 139 blocks used, 101 unused, 101 in largest
```

For 8080 Systems, the release disks are the same, except that ADAPZ is replaced by ADAP8:

```
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
```

```
SYS80D:
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 keys-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   202    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   202    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:

Interpreter	INTERPX.CODE or INTERPX.2.CODE or INTERPX.4.CODE
Runtime Support Package	RSP.CODE
BIOS	BIOS.CODE or BIOS.C.CODE or BIOS.CR.CODE or BIOS.CRP.CODE
Tertiary Bootstrap	TERTBOOT.CODE

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 (intrasegment) jump to the bootstrap address (8000H -- CS-relative). (The CS register may also be set by making a long {intersegment} 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.

Routine	Vector offset	Inputs	Outputs
SYSINIT	00	AX = pointer to Interpreter jump table	
SYSHALT	02		
CONINIT	04		AH = ioresult

CONSTAT	06		AH = ioresult
			AL = char present

CONREAD	08		AH = ioresult
			AL = char

CONWRIT	0A	AL = char	AH = ioresult
SETDISK	0C	AL = current disk	
SETTRAK	0E	AL = current track	
SETSECT	10	AL = current sector	
SETBUFR	12	AX = buffer addr (ES-relative)	
DSKREAD	14		AH = ioresult
DSKWRIT	16		AH = ioresult
DSKINIT	18		AH = ioresult
DSKSTRT	1A		
DSKSTOP	1C		
PRNINIT	1E		AH = ioresult

PRNSTAT	20		AH = ioresult
			AL = char present

PRNREAD	22		AH = ioresult
			AL = char

PRNWRIT	24	AL = char	AH = ioresult
REMINIT	26		AH = ioresult

REMSTAT	28		AH = ioresult
			AL = char present

REMREAD	2A		AH = ioresult
			AL = char

REMWRIT	2C	AL = char	AH = ioresult
USRINIT	2E	AL = unit #	AH = ioresult
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	
USRWRIT	34	TOS(SP)->return addr block # byte count ^buffer (DS-rel) device # control word	

CLKREAD	36		AH = ioresult
			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 ioresult 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.

Routine	Vector offset	Inputs
POLLUNITS	00	
DSKCHNG	02	BX = ^disk descriptor block (#tracks #sectors sector size interleaving skew first track)
EVENT	04	DI = event #

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 SBIOS must be relative to this CS value.)

The following parameters must be on the stack:

```
TOS --> SBIOS tester parameter  (ignored by primary boot)
          address of Interpreter  (CS-relative)
          address of SBIOS           (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 SBIOS, and call the SBIOS 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 SBIOS       (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 SBIOS 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 (intrasegment) 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 <u>unitclear</u> 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 <u>unitclear</u> 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.

Routine	Vector offset	Inputs	Outputs

CONSRD	00		AH = ioresult
			AL = char

CONSWR	02	AL = char	AH = ioresult
CONSIN	04	TOS -> ^break routine ^syscom	AH = ioresult
CONSST	06	TOS -> ^status record control word	AH = ioresult

PRNRD	08		AH = ioresult
			AL = char

PRNWR	0A	AL = char	AH = ioresult
PRNIN	0C		AH = ioresult
PRNST	0E	TOS = ^status record control word	AH = ioresult
DSKRD	10	TOS = block # byte count ^buffer (ES-relative) drive # (0..5) control word	AH = ioresult
DSKWR	12	TOS = block # byte count ^buffer (ES-relative) drive # (0..5) control word	AH = ioresult
DSKIN	14	CL = drive # (0..5)	AH = ioresult

DSKST	16	CL = drive # (0..5)
		TOS = ^status record
		control word	

REMRD	18		AH = ioresult
			AL = char

REMWR	1A	AL = char	AH = ioresult
REMIN	1C		AH = ioresult
REMST	1E	TOS = ^status record control word	AH = ioresult
USERRD	20	TOS -> block # byte count ^buffer (DS-rel) device # control word	
USERWR	22	TOS -> block # byte count ^buffer (DS-rel) device # control word	"
USERIN	24	AL = unit #	AH = ioresult
USERST	26	TOS -> i/o toggle ^statrec device #	
SYSRD	28		
SYSWR	2A		
SYSIN	2C		

SYSST	2E	TOS -> ^statrec
		control word
			statrec[0]=hiram
			statrec[1]=clock(lo)
			statrec[2]=clock(hi)

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 >        108             132
4/4 files<listed/in-dir>, 132 blocks used, 108 unused, 108 in largest
```

```
SYS2:
ASM8086.CODE       60 29-May-81     6    512  Codefile
8086.OPCODES        5  5-May-81    66     52  Datafile
8086.ERRORS        11  8-May-81    71    130  Datafile
8087.FOPS           3  4-Feb-81    82    512  Datafile
YALOE.CODE         12  2-Dec-80    85    512  Codefile
BOOTER.CODE         3  4-Dec-80    97    512  Codefile
DISKCHANGE.CODE     8  5-Dec-80   100    512  Codefile
DISKSIZE.CODE       3  3-Dec-80   108    512  Codefile
FINDPARAMS.CODE     9  3-Dec-80   111    512  Codefile
SAMPLEGOTO.TEXT     4 17-Nov-78   120    512  Textfile
DECODE.CODE        28  5-Mar-81   124    512  Codefile
COPYDUPDIR.CODE     3  2-Dec-80   152    512  Codefile
MARKDUPDIR.CODE     4  2-Dec-80   155    512  Codefile
XREF.CODE          29  3-Dec-80   159    512  Codefile
RECOVER.G.CODE      8  5-Dec-80   188    512  Codefile
SCREENTEST.CODE    13  4-Jun-81   196    512  Codefile
LIBRARY.CODE       13  6-Nov-81   209    512  Codefile
ABSWRITE.CODE       4  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     108 24-Nov-81    26    512  Datafile
SYSTEM.FILER       32 28-Jul-81   134    512  Codefile
SETUP.CODE         28 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
INTERPX.CODE       20  6-Oct-81    16    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
TERTBOOT.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 20-Jul-81   206    512  Codefile
SCREENOPS.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
```

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(ist P(code, or 'LR' for L(ist R(egister). 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(lobal, L(ocal, I(ntermediate, E(xtended, or P(rocedure. If 'G' or 'L' is typed, the prompt 'Varname 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 'Varname 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 msstat=347C msdyn=F0A0 msipc=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 erec=FEE8 seg=9782 ipc=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 msstat=347C msdyn=F0A0 msipc=01DA msenv=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
                   ('Whats 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 O#6
 (cd)  S=NOTDEBUG  P#1 O#6  SLDO1
 (cd)  S=NOTDEBUG  P#1 O#7  SLDC1
 (cd)  S=NOTDEBUG  P#1 O#8  NEQUI
```

We see that our first single-step did a short load global 1. (Note: This put 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, erec, seg, ipc, tib, rdyq
P(code	Current P-code mnemonic
M(arkstack	Mark stack display
A(ddress	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(lobal	Displays global memory
L(ocal	Displays local memory
I(nter	Displays intermediate memory
P(roc	Displays data segement 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:

```
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.}
```

```
        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:

CODE POOL BASE[First Word]
CODE POOL 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 Pooldes 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:

Routine Name	Vector Number	Description
POLLUNITS	0	polls character I/O devices
DSKCHNG	1	changes disk format values
EVENT	2	signals an event

(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 attach'ed a semaphore to an event number (as described in the Users' Manual), then a call to EVENT with that event number signal's the semaphore. The signal takes place before the next P-code is executed. If the event number has not been attach'ed, 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
   2     1:u    1
   2     1:u    1
   2     1:u    1   CONST
   2     1:u    1     VERSION = '[IV.1 B4h]';
   2     1:u    1     MMAXINT = 32767;    { MAXIMUM INTEGER VALUE }
   2     1:u    1     MAXDIR = 77;        { MAX NUMBER OF ENTRIES IN A DIRECTORY }
   2     1:u    1     VIDLENG = 7;        { NUMBER OF CHARS IN A VOLUME ID }
   2     1:u    1     TIDLENG = 15;       { NUMBER OF CHARS IN TITLE ID }
   2     1:u    1     MAXSEG = 15;        { MAX CODE SEGMENT NUMBER }
   2     1:u    1     FBLKSIZE = 512;     { STANDARD DISK BLOCK LENGTH }
   2     1:u    1     DIRBLK = 2;         { DISK ADDR OF DIRECTORY }
   2     1:u    1     AGELIMIT = 300;     { MAX AGE FOR GDIRP...IN TICKS }
   2     1:u    1     EOL = 13;           { END-OF-LINE...ASCII CR }
   2     1:u    1     DLE = 16;           { BLANK COMPRESSION CODE }
   2     1:u    1     NAME_LENG = 23;     { Number of characters in a full file name}
   2     1:u    1     SWAPPING = 0;       { Swapping segment status}
   2     1:u    1     P_LOCKED = -1;      { Position locked segment status}
   2     1:u    1     STACK_SLOP = 40;    { Number of words of temp for procedure stack}
   2     1:u    1     MEM_LINK_SIZE = 4;  { Number of words in heap record}
   2     1:u    1
   2     1:u    1     sys_error = 0;      { Unknown system error}
   2     1:u    1     proc_error= 3;      { Unknown procedure error}
   2     1:u    1     stk_error = 4;      { Stack overflow error}
   2     1:u    1     sys_io_error = 9;   { System I/O error}
   2     1:u    1     halt_error = 14;    { Programmed halt }
   2     1:u    1     heap_error= 15;     { Heap operation error}
   2     1:u    1     seg_fault = 128;    { Segment fault}
   2     1:u    1     stk_fault = 129;    { Stack fault}
   2     1:u    1     heap_fault= 130;    { Heap operation fault}
   2     1:u    1     pool_fault= 131;    { Used to consolidate pool after purge}
   2     1:u    1
   2     1:u    1   TYPE
   2     1:u    1
   2     1:u    1     IORSLTWD = (INOERROR,IBADBLOCK,IBADUNIT,IBADMODE,ITIMEOUT,
   2     1:u    1                 ILOSTUNIT,ILOSTFILE,IBADTITLE,INOROOM,INOUNIT,
   2     1:u    1                 INOFILE,IDUPFILE,INOTCLOSED,INOTOPEN,IBADFORMAT,
   2     1:u    1                 IBUFOVFL, i_write_prot, i_ill_block, i_ill_buf);
   2     1:u    1
   2     1:u    1                       { COMMAND STATES...SEE GETCMD }
   2     1:u    1
   2     1:u    1     CMDSTATE = (sys_boot, sys_init, sys_halt,UPROGNOU,UPROGUOK,SYSPROG,
   2     1:u    1                 COMPONLY,COMPANDGO,LINKANDGO,sys_debug);
   2     1:u    1
   2     1:u    1                                            { CODE FILES USED IN GETCMD }
   2     1:u    1
   2     1:u    1     SYSFILE = (ASSMBLER,COMPILER,EDITOR,FILER,LINKER);
   2     1:u    1
   2     1:u    1                                            { ARCHIVAL INFO...THE DATE }
   2     1:u    1
   2     1:u    1     DATEREC = PACKED RECORD
   2     1:u    1                 MONTH: 0..12;             { 0 IMPLIES DATE NOT MEANINGFUL }
   2     1:u    1                 DAY: 0..31;               { DAY OF MONTH }
   2     1:u    1                 YEAR: 0..100              { 100 IS TEMP DISK FLAG }
   2     1:u    1               END { DATEREC } ;
   2     1:u    1
```

```
 2   1:u   1                                       { VOLUME TABLES }
 2   1:u   1    UNITNUM = 0..127;
 2   1:u   1    VID = STRING[VIDLENG];
 2   1:u   1
 2   1:u   1                                       { DISK DIRECTORIES }
 2   1:u   1    DIRRANGE = 0..MAXDIR;
 2   1:u   1    TID = STRING[TIDLENG];
 2   1:u   1
 2   1:u   1    full_id = STRING[name_leng];
 2   1:u   1    file_table = ARRAY [sys_file] OF full_id;
 2   1:u   1
 2   1:u   1    FILEKIND = (UNTYPEDFILE,XDSKFILE,CODEFILE,TEXTFILE,INFOFILE,
 2   1:u   1                DATAFILE,GRAFFILE,FOTOFILE,SECUREDIR,SUBSVOL);
 2   1:u   1
 2   1:u   1    DIRENTRY = PACKED RECORD
 2   1:u   1                 DFIRSTBLK: INTEGER;   { FIRST PHYSICAL DISK ADDR }
 2   1:u   1                 DLASTBLK: INTEGER;    { POINTS AT BLOCK FOLLOWING }
 2   1:u   1                 CASE DFKIND: FILEKIND OF
 2   1:u   1                   SECUREDIR,
 2   1:u   1                   UNTYPEDFILE: { ONLY IN DIR[0]...VOLUME INFO }
 2   1:u   1                      (filler_1 : 0..2048;      {13 bits}
 2   1:u   1                       DVID: VID;               { NAME OF DISK VOLUME }
 2   1:u   1                       DEOVBLK: INTEGER;        { LASTBLK OF VOLUME }
 2   1:u   1                       DNUMFILES: DIRRANGE;     { NUM FILES IN DIR }
 2   1:u   1                       DLOADTIME: INTEGER;      { TIME OF LAST ACCESS }
 2   1:u   1                       DLASTBOOT: DATEREC);     { MOST RECENT DATE SETTING }
 2   1:u   1                   XDSKFILE,CODEFILE,TEXTFILE,INFOFILE,
 2   1:u   1                   DATAFILE,GRAFFILE,FOTOFILE,SUBSVOL:
 2   1:u   1                      (filler_2 : 0..1024;      {12 bits}
 2   1:u   1                       status : BOOLEAN;        {Filer kludge temporary}
 2   1:u   1                       DTID: TID;               { TITLE OF FILE }
 2   1:u   1                       DLASTBYTE: 1..FBLKSIZE;  { NUM BYTES IN LAST BLOCK }
 2   1:u   1                       DACCESS: DATEREC)        { LAST MODIFICATION DATE }
 2   1:u   1               END { DIRENTRY } ;
 2   1:u   1
 2   1:u   1    DIRP = ^DIRECTORY;
 2   1:u   1
 2   1:u   1    DIRECTORY = ARRAY [DIRRANGE] OF DIRENTRY;
 2   1:u   1
 2   1:u   1                                       { FILE INFORMATION }
 2   1:u   1
 2   1:u   1    CLOSETYPE = (CNORMAL,CLOCK,CPURGE,CCRUNCH);
 2   1:u   1    WINDOWP = ^WINDOW;
 2   1:u   1    WINDOW = PACKED ARRAY [0..0] OF CHAR;
 2   1:u   1    FIBP = ^FIB;
 2   1:u   1
 2   1:u   1    FIB = RECORD
 2   1:u   1            FWINDOW: WINDOWP;  { USER WINDOW...F^, USED BY GET-PUT }
 2   1:u   1            FEOF,FEOLN: BOOLEAN;
 2   1:u   1            FSTATE: (FJANDW,FNEEDCHAR,FGOTCHAR);
 2   1:u   1            FRECSIZE: INTEGER; { IN BYTES...0=>BLOCKFILE, 1=>CHARFILE }
 2   1:u   1            f_lock : SEMAPHORE;
 2   1:u   1            CASE FISOPEN: BOOLEAN OF
 2   1:u   1              TRUE: (FISBLKD: BOOLEAN; { FILE IS ON BLOCK DEVICE }
 2   1:u   1                     FUNIT: UNITNUM;    { PHYSICAL UNIT # }
 2   1:u   1                     FVID: VID;         { VOLUME NAME }
 2   1:u   1                     FREPTCNT,          {  # TIMES F^ VALID W/O GET }
 2   1:u   1                FNXTBLK,          { NEXT REL BLOCK TO IO }
 2   1:u   1                FMAXBLK: INTEGER; { MAX REL BLOCK ACCESSED }
 2   1:u   1                FMODIFIED:BOOLEAN;{ PLEASE SET NEW DATE IN CLOSE }
 2   1:u   1                FHEADER: DIRENTRY;{ COPY OF DISK DIR ENTRY }
 2   1:u   1                CASE FSOFTBUF: BOOLEAN OF { DISK GET-PUT STUFF }
 2   1:u   1                  TRUE: (FNXTBYTE,FMAXBYTE: INTEGER;
 2   1:u   1                         FBUFCHNGD: BOOLEAN;
 2   1:u   1                         FBUFFER: PACKED ARRAY [0..FBLKSIZE] OF CHAR))
 2   1:u   1          END { FIB } ;
 2   1:u   1
 2   1:u   1                                 { USER WORKFILE STUFF }
 2   1:u   1
 2   1:u   1       INFOREC = RECORD
 2   1:u   1                 SYMFIBP,CODEFIBP: FIBP;         { WORKFILES FOR SCRATCH }
 2   1:u   1                 ERRSYM,ERRBLK,ERRNUM: INTEGER; { ERROR STUFF IN EDIT }
 2   1:u   1                 SLOWTERM,STUPID: BOOLEAN;      { STUDENT PROGRAMMER ID!! }
 2   1:u   1                 ALTMODE: CHAR;                 { WASHOUT CHAR FOR COMPILER }
 2   1:u   1                 GOTSYM,GOTCODE: BOOLEAN;       { TITLES ARE MEANINGFUL }
 2   1:u   1                 WORKVID,SYMVID,CODEVID: VID;   { PERM&CUR WORKFILE VOLUMES }
 2   1:u   1                 WORKTID,SYMTID,CODETID: TID    { PERM&CUR WORKFILES TITLE }
 2   1:u   1               END { INFOREC } ;
 2   1:u   1
 2   1:u   1
 2   1:u   1       int_p = ^INTEGER;
 2   1:u   1       tib_p = ^tib;
 2   1:u   1       sib_p = ^sib;
 2   1:u   1       e_rec_p = ^e_rec;
 2   1:u   1       e_vec_p = ^e_vec;
 2   1:u   1       sem_p = ^sem;
 2   1:u   1       mscwp = ^mscw;
 2   1:u   1       p_mem_chunk = ^mem_chunk;
 2   1:u   1       vip = ^vinfo;
 2   1:u   1
 2   1:u   1       byte = 0..255;
 2   1:u   1       mem_chunk = ARRAY [0..0] OF INTEGER;       {Accessed $R-}
 2   1:u   1       alpha = PACKED ARRAY [0..7] OF CHAR;       {Identifier name}
 2   1:u   1
 2   1:u   1       mem_ptr = RECORD
 2   1:u   1                 CASE INTEGER OF
 2   1:u   1                   0 : (m : ^mem_link);
 2   1:u   1                   1 : (i : int_p);
 2   1:u   1                   2 : (c : p_mem_chunk);
 2   1:u   1                   3 : (t : INTEGER);
 2   1:u   1               END {of mem_ptr};
 2   1:u   1
 2   1:u   1       mem_link = RECORD
 2   1:u   1                  avail_list : mem_ptr;
 2   1:u   1                  n_words : INTEGER;
 2   1:u   1                  CASE BOOLEAN OF
 2   1:u   1                    TRUE : (last_avail,
 2   1:u   1                            prev_mark : mem_ptr);
 2   1:u   1                END {of mem_link};
 2   1:u   1
 2   1:u   1       vinfo = record
 2   1:u   1               segunit : integer;
 2   1:u   1               segvid  : vid;
 2   1:u   1             end {of vinfo};
```

```
2   1:u   1
2   1:u   1   poolptr = ^pooldes;
2   1:u   1   sib = RECORD
2   1:u   1           seg_pool : poolptr;
2   1:u   1           seg_base : mem_ptr;        {Base memory location}
2   1:u   1           seg_refs : INTEGER;        {# of active calls}
2   1:u   1           timestamp: INTEGER;        {Memory swap priority}
2   1:u   1           link_count:INTEGER;        {Number of links to SIB}
2   1:u   1           residency: p_locked..MAX_INT;      {memory residency status}
2   1:u   1           seg_name : alpha;          {Segment name}
2   1:u   1           seg_leng : INTEGER;        {# of words in segment}
2   1:u   1           seg_addr : INTEGER;        {Disk address of segment}
2   1:u   1           vol_info : vip;            {Disk unit and vol id of segment}
2   1:u   1           data_size: INTEGER;        {Number of words in data segment}
2   1:u   1           res_sibs : RECORD          {Code Pool management record}
2   1:u   1                   next_sib,              {Pointer to next sib}
2   1:u   1                   prev_sib : sib_p;      {Pointer to previous sib}
2   1:u   1                   CASE BOOLEAN OF        {Scratch area}
2   1:u   1                     TRUE  : (next_sort : sib_p);
2   1:u   1                     FALSE : (new_loc : mem_ptr)
2   1:u   1                 END {of res_sibs};
2   1:u   1           mtype:integer;
2   1:u   1         END {of sib};
2   1:u   1
2   1:u   1   e_vec = RECORD     {Environment vector}
2   1:u   1             vect_length : INTEGER;
2   1:u   1             map : ARRAY [1..1] OF e_rec_p;   {Accessed $R-}
2   1:u   1           END {of e_vec};
2   1:u   1
2   1:u   1   e_rec = RECORD     {Environment record}
2   1:u   1             env_data : mem_ptr;      {Pointer to base data segment}
2   1:u   1             env_vect : e_vec_p;      {Pointer to environment vector}
2   1:u   1             env_sib : sib_p;         {Pointer to associated segment}
2   1:u   1             CASE BOOLEAN OF          {Outer block information}
2   1:u   1               TRUE : (link_count : INTEGER;
2   1:u   1                       next_rec : e_rec_p);
2   1:u   1           END; {of e_rec}
2   1:u   1
2   1:u   1   mscw = PACKED RECORD { Mark stack control }
2   1:u   1           ms_stat : mscw_p;{ Lexical parent pointer }
2   1:u   1           ms_dynl : mscw_p;{ Ptr to caller's mscw }
2   1:u   1           ms_ipc : INTEGER;{ Byte inx in retrn code seg }
2   1:u   1           ms_env : e_rec_p;{ Environment of caller code }
2   1:u   1           ms_proc: integer;{ Proc # of caller }
2   1:u   1         END { of mscw } ;
2   1:u   1
2   1:u   1   tib = RECORD { Task information block }
2   1:u   1           regs : PACKED RECORD
2   1:u   1                   wait_q : tib_p;     { Queue link for semaphores }
2   1:u   1                   prior : byte;       { Task's cpu priority }
2   1:u   1                   flags : byte;       { State flags...not defined yet }
2   1:u   1                   sp_low : mem_ptr;   { Lower stack pointer limit }
2   1:u   1                   sp_upr : mem_ptr;   { Upper limit on stack }
2   1:u   1                   sp : mem_ptr;       { Actual top of stack pointer }
2   1:u   1                   mp : mscw_p;        { Active procedure MSCW ptr }
2   1:u   1                   reserved : integer;
2   1:u   1                   ipc : INTEGER;      { Byte ptr in current code seg }
```

```
 2   1:u   1                    env : e_rec_p;    { Ptr to current environment }
 2   1:u   1                    procnum:byte;
 2   1:u   1                    tibioresult:byte;
 2   1:u   1                    hang_p : sem_p;    { Which task is waiting on }
 2   1:u   1                    m_depend : INTEGER;{ Reserved for interpreter }
 2   1:u   1                                       { initted to 0 when process started }
 2   1:u   1                  END { of regs } ;
 2   1:u   1           main_task : BOOLEAN;
 2   1:u   1           start_mscw : mscw_p;
 2   1:u   1         END { of tib } ;
 2   1:u   1
 2   1:u   1   sem = RECORD { Semaphore format }
 2   1:u   1           sem_count : INTEGER;  { Number outstanding signals }
 2   1:u   1           sem_wait_q : tib_p    { List of tasks waiting on sem }
 2   1:u   1         END { of sem } ;
 2   1:u   1
 2   1:u   1   fault_message = RECORD
 2   1:u   1                     fault_tib : tib_p;
 2   1:u   1                     fault_e_rec : e_rec_p;
 2   1:u   1                     fault_words : INTEGER;
 2   1:u   1                     fault_type : seg_fault..pool_fault;
 2   1:u   1                   END {of fault_message};
 2   1:u   1
 2   1:u   1   fulladdress = array[0..1] of integer; {32 bits}
 2   1:u   1
 2   1:u   1   utablentry = record
 2   1:u   1                  uvid : vid;    { volume id for unit }
 2   1:u   1                  case uisblkd : boolean of
 2   1:u   1                    true : (ueovblk : integer;
 2   1:u   1                            uphysvol : unitnum;
 2   1:u   1                            ublkoff : integer;
 2   1:u   1                            upvid : vid)
 2   1:u   1                end {utable} ;
 2   1:u   1   utable = array [unitnum] of utablentry; { 0 not used }
 2   1:u   1
 2   1:u   1                                      { SYSTEM COMMUNICATION AREA }
 2   1:u   1                                      { SEE INTERPRETERS...NOTE   }
 2   1:u   1                                      { THAT WE ASSUME BACKWARD   }
 2   1:u   1                                      { FIELD ALLOCATION IS DONE  }
 2   1:u   1   syscomrec = record {modified for IV.1}
 2   1:u   1                 iorslt : iorsltwd;   { RESULT OF LAST IO CALL }
 2   1:u   1                 rsrvd1 : integer;
 2   1:u   1                 sysunit : unitnum;   { PHYSICAL UNIT OF BOOTLOAD }
 2   1:u   1                 rsrvd2 : integer;
 2   1:u   1                 gdirp : dirp;        { GLOBAL DIR POINTER,SEE VOLSEARCH }
 2   1:u   1                 fault_sem : RECORD
 2   1:u   1                        real_sem, message_sem : SEMAPHORE;
 2   1:u   1                        message : fault_message;
 2   1:u   1                      END {of fault_sem};
 2   1:u   1                   { starting unit number for subsidiary volumes}
 2   1:u   1                 subsidstart : unitnum;
 2   1:u   1                 rsrvd3 : integer;
 2   1:u   1                 spool_avail : boolean;
 2   1:u   1                 poolinfo : record
 2   1:u   1                              pooloutside : boolean;
 2   1:u   1                              poolsize    : integer;
 2   1:u   1                              poolbase    : fulladdress;
```

```
2   1:u   1                          resolution  : integer;
2   1:u   1                        end;
2   1:u   1             timestamp : integer;
2   1:u   1             unitable : ^utable;
2   1:u   1             unitdivision : packed record
2   1:u   1                          serialmax : byte;  {number of user serial units}
2   1:u   1                          subsidmax : byte;  {max number of subsid vols}
2   1:u   1                        end;
2   1:u   1             expaninfo: packed record
2   1:u   1                          insertchar,deletchar:char;
2   1:u   1                          expan1,expan2:integer;
2   1:u   1                        end;
2   1:u   1             pmachver : (pre_iv_i, iv_i, post_iv_1);
2   1:u   1             realsize : integer;
2   1:u   1             MISCINFO: PACKED RECORD
2   1:u   1                         NOBREAK,STUPID,SLOWTERM,
2   1:u   1                         HASXYCRT,HASLCCRT,HAS8510A,HASCLOCK: BOOLEAN;
2   1:u   1                         USERKIND:(NORMAL, AQUIZ, BOOKER, PQUIZ)
2   1:u   1                       END;
2   1:u   1             CRTTYPE: INTEGER;
2   1:u   1             CRTCTRL: PACKED RECORD
2   1:u   1                        RLF,NDFS,ERASEEOL,ERASEEOS,HOME,ESCAPE: CHAR;
2   1:u   1                        BACKSPACE: CHAR;
2   1:u   1                        FILLCOUNT: 0..255;
2   1:u   1                        CLEARSCREEN, CLEARLINE: CHAR;
2   1:u   1                        PREFIXED: PACKED ARRAY [0..8] OF BOOLEAN
2   1:u   1                      END;
2   1:u   1             CRTINFO: PACKED RECORD
2   1:u   1                        WIDTH,HEIGHT: INTEGER;
2   1:u   1                        RIGHT,LEFT,DOWN,UP: CHAR;
2   1:u   1                        BADCH,CHARDEL,STOP,BREAK,FLUSH,EOF: CHAR;
2   1:u   1                        ALTMODE,LINEDEL: CHAR;
2   1:u   1                        alphalok,char_mask,ETX,PREFIX: CHAR;
2   1:u   1                        PREFIXED: PACKED ARRAY [0..15] OF BOOLEAN;
2   1:u   1                      END
2   1:u   1           END { SYSCOM };
2   1:u   1
2   1:u   1   MISCINFOREC = RECORD
2   1:u   1                   MSYSCOM: SYSCOMREC
2   1:u   1                 END;
2   1:u   1
2   1:u   1   pooldes = record
2   1:u   1               poolbase   : fulladdress;
2   1:u   1               poolsize   : integer;
2   1:u   1               minoffset  : memptr;
2   1:u   1               maxoffset  : memptr;
2   1:u   1               resolution : integer; {in bytes}
2   1:u   1               poolhead   : sibp;
2   1:u   1               permsib    : sibp;
2   1:u   1               extended   : boolean;
2   1:u   1             end;
2   1:u   1   bytearray = packed array [0..0] of byte;
2   1:u   1
2   1:u   1
2   1:u   1 VAR
2   1:u   1   SYSCOM: ^SYSCOMREC;                  { MAGIC PARAM...SET UP IN BOOT }
2   1:u   2   GFILES: ARRAY [0..5] OF FIBP;        { GLOBAL FILES, 0=INPUT, 1=OUTPUT }
```

```
 2    1:u    8    USERINFO: INFOREC;                    { WORK STUFF FOR COMPILER ETC }
 2    1:u   54    EMPTYHEAP: ^integer;                  { HEAP MARK FOR MEM MANAGING }
 2    1:u   55    maintask : tib_p;                     { taskinfo block of op sys prog }
 2    1:u   56    Has_PM : BOOLEAN;                     { performance monitor in use }
 2    1:u   57    INPUTFIB,OUTPUTFIB,SYSTERM : FIBP;    { CONSOLE FILES...GFILES ARE COPIES }
 2    1:u   60    SYVID,DKVID: VID;                     { SYSUNIT VOLID & DEFAULT VOLID }
 2    1:u   68    THEDATE: DATEREC;                     { TODAY...SET IN FILER OR SIGN ON }
 2    1:u   69    STATE: CMDSTATE;                      { FOR GETCOMMAND }
 2    1:u   70    heap_info : RECORD { Stuff for heap management }
 2    1:u   70                   Lock : SEMAPHORE;
 2    1:u   70                   top_mark,
 2    1:u   70                   heap_top : mem_ptr;
 2    1:u   70                 END { of heap_info } ;
 2    1:u   74    task_info : RECORD { Stuff for task management }
 2    1:u   74                   Lock,
 2    1:u   74                   task_done : SEMAPHORE;
 2    1:u   74                   n_tasks : INTEGER;
 2    1:u   74                 END { of task_info } ;
 2    1:u   79    IPOT: ARRAY [0..4] OF INTEGER;        { INTEGER POWERS OF TEN }
 2    1:u   84    FILLER: STRING[11];                   { NULLS FOR CARRIAGE DELAY }
 2    1:u   90    DIGITS: SET OF '0'..'9';
 2    1:u   94    PL: STRING;
 2    1:u  135    maxunit : unitnum;
 2    1:u  136    FILENAME: file_table;
 2    1:u  196    junk : processid;
 2    1:u  197    fault_sem : RECORD
 2    1:u  197                   real_sem, message_sem : SEMAPHORE;
 2    1:u  197                   message : fault_message;
 2    1:u  197                 END {of fault_sem};
 2    1:u  205    unit_List : e_rec_p;
 2    1:u  206    user_env_vec,
 2    1:u  206    sys_env_vec : e_vec_p;
 2    1:u  208    dir_Lock  : SEMAPHORE;  { for volume and directory exclusion }
 2    1:u  210    inexerr : boolean;      { set when processing execution errors
 2    1:u  211                              so that file system Locks are not enforced }
 2    1:u  211    dfliptog:boolean;       { true when gdirp contains a flipped directory }
 2    1:u  212    debugging:boolean;
 2    1:u  213    permList:memptr;        { List of "permanent" new's }
 2    1:u  214    suspectset : set of unitnum;
 2    1:u  222    atsysprompt:boolean;    { true if at system prompt }
 2    1:u  223    userLib:fullid;         { Library text file }
 2    1:u  235    sysList:e_rec_p;        { Last unit placed on unit_List that is a
{commented ';'}                               system unit; all subsequent units added
 2    1:u  236                              to the unit_List are user units }
 2    1:u  236    nofit : boolean;        {set to true if dumpsegs can't get any more room}
 2    1:u  237    codepool : poolptr;
 2    1:u  238    resolving_break : boolean; {true if in debugger and a code segment
 2    1:u  239                                must be read in without breakpoints}
 2    1:u  239
 2    1:u  239    procedure exec_error(bad_e_rec_p:e_rec_p; n_words,err:integer);
 2    1:u    1    procedure Loadseg(segerec:erecp);
 2    1:u    1    procedure rlocseg(segerec:erecp);
 2    1:u    1    function ptr_add (p : mem_ptr; n_words : integer) : int_p;
 2    1:u    1    function ptr_sub (p_0, p_1 : mem_ptr) : integer;
 2    1:u    1    function ptr_less (p_1, p_2 : mem_ptr) : boolean;
 2    1:u    1    function ptr_gtr (p_1, p_2 : mem_ptr) : boolean;
 2    1:u    1    function ptr_geq (p_1, p_2 : mem_ptr) : boolean;
 399   2    1:u    1     procedure print(s:string);
 400   2    1:u    1     procedure printint(i:integer);
 401   2    1:u    1     procedure writestr(s:string);
 402   2    1:u    1     procedure checkunit(lvid:vid; lunit:unitnum);
 403   2    1:u    1     procedure moveseg(segsib:sibp; srcpool:poolptr; srcoffset:memptr);
 404   2    1:u    1

nd of Compilation.
```

```
                                Using COMMANDI
   2   2    1:u    1
   3   2    1:u    1  type bigstring=string[255];
   4   2    1:u    1  var havechain,inredirect,outredirect,monitoropen,inmonitor:boolean;
   5   2    1:u    6    function redirect(command:bigstring):boolean;
   6   2    1:u    1    procedure exception(stopchaining:boolean);
   7   2    1:u    1    procedure chain(command:bigstring);
   8   2    1:u    1
   9   2    1:u    1    procedure initcommand;
  10   2    1:u    1    procedure startmonitor;
  11   2    1:u    1    procedure stopmonitor(saveit:boolean);
  12   2    1:u    1    procedure getchainline(var command:bigstring);
  13   2    1:u    1

End of [illegible]mpilation.
```

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

nd of Compilation.
```

```
                  Using SCREENOP
  2   1:u    1
  2   1:u    1
  2   1:u    1  const
  2   1:u    1      sc_fill_len = 11;
  2   1:u    1      sc_eol = 13;
  2   1:u    1
  2   1:u    1  type
  2   1:u    1      sc_chset        = set of char;
  2   1:u    1      sc_misc_rec     = packed record
  2   1:u    1                          height, width : 0..255;
  2   1:u    1                          can_break, slow, xy_crt, lc_crt,
  2   1:u    1                          can_upscroll, can_downscroll : boolean;
  2   1:u    1                        end;
  2   1:u    1      sc_date_rec     = packed record
  2   1:u    1                          month : 0..12;
  2   1:u    1                          day :   0..31;
  2   1:u    1                          year :  0..99;
  2   1:u    1                        end;
  2   1:u    1      sc_info_type    = packed record
  2   1:u    1                          sc_version : string;
  2   1:u    1                          sc_date : sc_date_rec;
  2   1:u    1                          spec_char : sc_chset; {Characters not to echo}
  2   1:u    1                          misc_info : sc_misc_rec;
  2   1:u    1                        end;
  2   1:u    1      sc_long_string  = string[255];
  2   1:u    1      sc_scrn_command = (sc_whome, sc_eras_s, sc_erase_eol, sc_clear_lne,
  2   1:u    1                         sc_clear_scn, sc_up_cursor, sc_down_cursor,
  2   1:u    1                         sc_left_cursor, sc_right_cursor);
  2   1:u    1      sc_key_command  = (sc_backspace_key, sc_dc1_key, sc_eof_key, sc_etx_key,
  2   1:u    1                         sc_escape_key, sc_del_key, sc_up_key, sc_down_key,
  2   1:u    1                         sc_left_key, sc_right_key, sc_not_legal, sc_insert_key,
  2   1:u    1                         sc_delete_key);
  2   1:u    1      sc_choice       = (sc_get, sc_give);
  2   1:u    1      sc_window       = packed array [0..0] of char;
  2   1:u    1      sc_tx_port      = record
  2   1:u    1                         row, col,             { screen relative}
  2   1:u    1                         height, width,        { size of txport (zero based)}
  2   1:u    1                         cur_x, cur_y  : integer;
  2   1:u    1                                  {cursor positions relative to the txport }
  2   1:u    1                       end;
  2   1:u    1
  2   1:u    1       {entries 4..syscom^.subsidstart-1 are valid}
  2   1:u    1      sc_err_msg_array = array [4..4] of ^string;  {accessed $R-}
  2   1:u    1
  2   1:u    1   var
  2   1:u    1     sc_port : sc_tx_port;
  2   1:u    7     sc_printable_chars : sc_chset;
  2   1:u   23     sc_errorline : integer;
  2   1:u   24     sc_errormessage : ^sc_err_msg_array;
  2   1:u   25
  2   1:u   25  procedure sc_use_info(do_what:sc_choice; var t_info:sc_info_type);
  2   1:u    1  procedure sc_use_port(do_what:sc_choice; var t_port:sc_tx_port);
  2   1:u    1  procedure sc_erase_to_eol(x,line:integer);
  2   1:u    1  procedure sc_left;
  2   1:u    1  procedure sc_right;
  57   2    1:u    1    procedure sc_up;
  58   2    1:u    1    procedure sc_down;
  59   2    1:u    1    procedure sc_getc_ch(var ch:char; return_on_match:sc_chset);
  60   2    1:u    1    procedure sc_clr_screen;
  61   2    1:u    1    procedure sc_clr_line (y:integer);
  62   2    1:u    1    procedure sc_home;
  63   2    1:u    1    procedure sc_eras_eos (x,line:integer);
  64   2    1:u    1    procedure sc_goto_xy(x, line:integer);
  65   2    1:u    1    procedure sc_clr_cur_line;
  66   2    1:u    1    function  sc_find_x:integer;
  67   2    1:u    1    function  sc_find_y:integer;
  68   2    1:u    1    function  sc_scrn_has(what:sc_scrn_command):boolean;
  69   2    1:u    1    function  sc_has_key(what:sc_key_command):boolean;
  70   2    1:u    1    function  sc_map_crt_command(var k_ch:char):sc_key_command;
  71   2    1:u    1    function  sc_prompt(line :sc_long_string; x_cursor,y_cursor,x_pos,
  72   2    1:u    1                        where:integer; return_on_match:sc_chset;
  73   2    1:u   21                        no_char_back:boolean; break_char:char):char;
  74   2    1:u    1    function  sc_check_char(var buf:sc_window; var buf_index,bytes_left:integer)
  75   2    1:u                               :boolean;
  76   2    1:u    1    function  sc_space_wait(flush:boolean):boolean;
  77   2    1:u    1    procedure sc_init;
  78   2    1:u    1
```

nd of Compilation.

```
                       Using SYSINFO
  2   1:u   1
  2   1:u   1
  2   1:u   1   Type SI_Date_Rec = Packed Record
  2   1:u   1                        Month : 0..12;
  2   1:u   1                        Day : 0..31;
  2   1:u   1                        Year : 0..99;
  2   1:u   1                      End; { SI_Date_Rec }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Code_Vid (Var SI_Vol : String);
  2   1:u   1     { Returns name of volume containing current workfile code }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Code_Tid (Var SI_Title : String);
  2   1:u   1     { Returns title of current workfile code }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Text_Vid (Var SI_Vol : String);
  2   1:u   1     { Returns name of volume containing current workfile text }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Text_Tid (Var SI_Title : String);
  2   1:u   1     { Returns title of current workfile text }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Function SI_Sys_Unit : Integer;
  2   1:u   1     { Returns number of bootload unit }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Get_Sys_Vol (Var SI_Vol : String);
  2   1:u   1     { Returns system volume name }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Get_Pref_Vol (Var SI_Vol : String);
  2   1:u   1     { Returns prefix volume name }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Set_Pref_Vol (SI_Vol : String);
  2   1:u   1     { Sets prefix volume name }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Get_Date (Var SI_Date : SI_Date_Rec);
  2   1:u   1     { Returns current system date }
  2   1:u   1
  2   1:u   1
  2   1:u   1   Procedure SI_Set_Date (Var SI_Date : SI_Date_Rec);
  2   1:u   1     { Sets current system date }
  2   1:u   1
  2   1:u   1
  2   1:u   1
  2   1:u   1 {***********************************************************************}
  2   1:u   1
  2   1:u   1

nd of Compilation.
```

```
                        Using FILEINFO
 2   1:u   1
 2   1:u   1
 2   1:u   1
 2   1:u   1     Type F_File_Type = file;
 2   1:u   1          F_Date_Rec  = Packed Record
 2   1:u   1                         Month : 0..12;
 2   1:u   1                         Day   : 0..31;
 2   1:u   1                         Year  : 0..100;
 2   1:u   1                       End; { F_Date_Rec }
 2   1:u   1
 2   1:u   1
 2   1:u   1     Function F_Open (var fid:  F_File_Type):boolean;
 2   1:u   1
 2   1:u   1     (* returns true if the file is open and false if not open  *)
 2   1:u   1
 2   1:u   1     Function F_Length (Var Fid         : F_File_Type)    : Integer;
 2   1:u   1
 2   1:u   1       {Returns the length of the file attached to the Fid identifier.
 2   1:u   1        If the file is not opened  result is returned as zero}
 2   1:u   1
 2   1:u   1
 2   1:u   1     Function F_Unit_number (Var Fid       : F_File_Type)    : integer;
 2   1:u   1
 2   1:u   1       {Returns the unit containing the file attached to the Fid
 2   1:u   1        identifier.  If there is no file opened to Fid, the function
 2   1:u   1        result is Zero.}
 2   1:u   1
 2   1:u   1
 2   1:u   1     Procedure F_Volume (Var Fid         : F_File_Type;
 2   1:u   0                         Var File_Volume : String);
 2   1:u   1
 2   1:u   1       {Returns the name of the volume containing the file attached
 2   1:u   1        to the Fid identifier.  If there is no file opened to Fid,
 2   1:u   1        the file_volume is set to a null string.}
 2   1:u   1
 2   1:u   1
 2   1:u   1     Procedure F_File_Title (Var Fid        : F_File_Type;
 2   1:u   0                             Var File_Title : String);
 2   1:u   1
 2   1:u   1       {Returns the title (with suffix) of the file attached to the
 2   1:u   1        Fid identifier.  If there is no file opened to Fid,
 2   1:u   1        the File_title is set to the null string.}
 2   1:u   1
 2   1:u   1
 2   1:u   1     Function F_Start (Var Fid        : F_File_Type)    : integer;
 2   1:u   1
 2   1:u   1       {Returns the block number of the first block of the file
 2   1:u   1        attached to the Fid identifier.  If there is no file opened
 2   1:u   1        to Fid, the function result is returned is zero.}
 2   1:u   1
 2   1:u   1
 2   1:u   1     Function F_is_Blocked (Var Fid        : F_File_Type) : Boolean;
 2   1:u   1
 2   1:u   1       {Returns a boolean that is TRUE if the file attached to the
 2   1:u   1        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
   2  2    1:u    1
   3  2    1:u    1
   4  2    1:u    1  Type
   5  2    1:u    1
   6  2    1:u    1     D_PatRecP = ^D_PatRec;
   7  2    1:u    1     D_PatRec = Record
   8  2    1:u    1                 CompPos,          { starting position of pattern in subject string }
   9  2    1:u    1                 CompLen,          { length of pattern in subject string }
  10  2    1:u    1                 WildPos,          { starting position of pattern in wild string }
  11  2    1:u    1                 WildLen : Integer; { length of pattern in wildcard string }
  12  2    1:u    1                 Next : D_PatRecP;  { next pattern }
  13  2    1:u    1              End; { D_PatRec }
  14  2    1:u    1
  15  2    1:u    1
  16  2    1:u    1   Function D_Wild_Match(Wild, Comp : String; Var PPtr : D_PatRecP;
  17  2    1:u    6                                             PInfo : Boolean) : Boolean;
  18  2    1:u    1   { Compares two strings (one containing wildcards) and returns true if they
  19  2    1:u    1     match. Includes information about pattern matching that occurred if re-
  20  2    1:u    1     quested (by PInfo) }
  21  2    1:u    1
  22  2    1:u    1  {*************************************************************************}
  23  2    1:u    1

nd of Compilation.
```

```
                     Using DIRINFO
 2    1:u    1
 2    1:u    1         uses
 2    1:u    1           (*$U WILD.CODE*) wild;
 2    1:u    1
 2    1:u    1         Type
 2    1:u    1           D_DateRec = Packed Record
 2    1:u    1                         Month : 0..12;
 2    1:u    1                         Day   : 0..31;
 2    1:u    1                         Year  : 0..100;
 2    1:u    1                       End;
 2    1:u    1
 2    1:u    1
 2    1:u    1           D_NameType = (D_Vol, D_Code, D_Text, D_Data, D_SVol, D_Temp, D_Free);
 2    1:u    1
 2    1:u    1           D_Choice = Set of D_NameType;
 2    1:u    1
 2    1:u    1           D_ListP = ^D_List;
 2    1:u    1           D_List = Record
 2    1:u    1                      D_Unit : Integer;              { Unit # of entry }
 2    1:u    1                      D_Volume : String[7];          { volume name of unit }
 2    1:u    1                      D_VPat : D_PatRecP;            { volume pattern info }
 2    1:u    1                      D_NextEntry : D_ListP;         { Next entry in list }
 2    1:u    1                      Case D_IsBlkd : Boolean Of
 2    1:u    1                        True : (D_Start,             { Starting block of entry }
 2    1:u    1                                D_Length : Integer;  { Length (in blocks) of entry }
 2    1:u    1                                Case D_Kind : D_NameType Of
 2    1:u    1                                  D_Vol,             { Everything but D_Free }
 2    1:u    1                                  D_Temp,
 2    1:u    1                                  D_Code,
 2    1:u    1                                  D_Text,
 2    1:u    1                                  D_Data,
 2    1:u    1                                  D_SVol : (D_Title : String[15];{ File name }
 2    1:u    1                                            D_FPat  : D_PatRecP; { name pattern info }
 2    1:u    1                                            D_Date  : D_DateRec; { File date }
 2    1:u    1                                            Case D_NameType of  { # of files on vol }
 2    1:u    1                                              D_Vol : (D_NumFiles : Integer)));
 2    1:u    1                  End;
 2    1:u    1
 2    1:u    1           D_Result = (D_Okay,           { Mission accomplished }
 2    1:u    1                       D_Not_Found,      { Couldn't find name and/or type }
 2    1:u    1                       D_Exists,         { Name already exists; no name change made }
 2    1:u    1                       D_Name_Error,     { Illegal string passed }
 2    1:u    1                       D_Off_Line,       { Volume not on line }
 2    1:u    1                       D_Other);         { Miscellaneous error }
 2    1:u    1
 2    1:u    1
 2    1:u    1       Function D_Dir_List(D_Name : String; D_Select : D_Choice;
 2    1:u    4                           Var D_Ptr : D_ListP; D_PInfo : Boolean) : D_Result;
 2    1:u    1       { Creates pointer to list of names of specified NameTypes
 2    1:u    1         (D_Select), matching specified D_Name (wildcard characters allowed). In-
 2    1:u    1         cludes information about pattern matching that occurred if requested
 2    1:u    1         (by D_PInfo) }
 2    1:u    1
 2    1:u    1
 2    1:u    1
 2   1:u   1   Function D_Scan_Title(D_Name : String; Var D_VolID, D_TitleID : String;
 2   1:u                            Var D_Type : D_NameType; Var D_Segs : Integer) : D_Result;
 2   1:u   1   { Parses D_Name }
 2   1:u   1
 2   1:u   1   Function D_Change_Name(D_OldName, D_NewName : String; D_RemOld : Boolean) : D_Result;
 2   1:u   1   { Changes file name in D_OldName to name in D_NewName, removing already
 2   1:u   1     existing files of name in D_NewName if D_RemOld is set }
 2   1:u   1
 2   1:u   1   Function D_Change_Date(D_Name : String; D_NewDate : D_DateRec;
 2   1:u                                                D_Select : D_Choice) : D_Result;
 2   1:u   1   { Changes date of directory or file name in D_Name to date specified by
 2   1:u   1     D_NewDate.  D_Name may contain wildcards }
 2   1:u   1
 2   1:u   1   Function D_Rem_Files (D_Name : String; D_Select : D_Choice) : D_Result;
 2   1:u   1   { Removes file of specified name (wildcards allowed) }
 2   1:u   1
 2   1:u   1   Procedure D_Lock;
 2   1:u   1   Procedure D_Release;
 2   1:u   1   { Provide means to limit use of DirInfo routines to one task at a time
 2   1:u   1     in multi-tasking environments }
 2   1:u   1
 2   1:u   1   Function D_Krunch (D_Unit,
 2   1:u   1                      D_Block : Integer) : D_Result;
 2   1:u   1   { Collects all unused space on a volume around D_Block. This unit must
 2   1:u   1     not be in use when this operation is performed.  }
 2   1:u   1
 2   1:u   1   Function D_Mount (D_File_Name : String) : D_Result;
 2   1:u   1   Function D_DisMount (D_Vol_Name : String) : D_Result;
 2   1:u   1   { Provides a means of mounting and dismounting subsidiary volumes.
 2   1:u   1     Wild cards may be used.  }
 2   1:u   1
 2   1:u   1
 2   1:u   1
 2   1:u   1  {************************************************************************}
     1:u   1

nd of Compilation.
```

```
 2   1:d   1  program windowdisp;

              Using WINDOWMA
 2   1:u   1
 2   1:u   1
 2   1:u   1    {Window Manager for the UCSD p-System}
 2   1:u   1    {Windows are displayed as rectangular areas on the screen, bordered by
 2   1:u   1     a frame and optionally headed by a heading.  Each window has its own
 2   1:u   1     size, screen location, text area, cursor and status information.
 2   1:u   1     Each window may be written into and will scroll independently, and
 2   1:u   1     may be cleared, moved, changed in size, etc. by a user's program.
 2   1:u   1
 2   1:u   1     During any input operation the user may escape into 'Window Manager
 2   1:u   1     Mode' (and subsequently return to 'Input Mode', to complete the input).
 2   1:u   1     In Window Manager Mode the Window Manager uses a special cursor which
 2   1:u   1     is independent of any window. This cursor is used to indicate screen
 2   1:u   1     position parameters to the Hide, Show, Alter, Move and Kill commands.
 2   1:u   1     Whether or not a particular command may be applied to a particular
 2   1:u   1     window is controlled by the user's program.
 2   1:u   1    }
 2   1:u   1
 2   1:u   1   CONST WVersion='10-Dec-82';
 2   1:u   1         NoWindow=0;
 2   1:u   1         MaxWindow=10;
 2   1:u   1
 2   1:u   1   TYPE  Window=NoWindow..MaxWindow;
 2   1:u   1         WindowOptions=(CanHide,CanMove,CanAlter,CanKill,
 2   1:u   1                        HasHeading,CanScroll,CanPan);
 2   1:u   1         WindowAttributes=SET OF WindowOptions;
 2   1:u   1
 2   1:u   1  {Initialisation Routines}
 2   1:u   1
 2   1:u   1    PROCEDURE WStartup;
 2   1:u   1    {Called by *SYSTEM.STARTUP to REALLY initialise}
 2   1:u   1    {If window manager is placed in *SYSTEM.PASCAL then you MUST supply
 2   1:u   1     a *SYSTEM.STARTUP that calls WStartup. Thereafter any program which
 2   1:u   1     uses the window manager should initialise via WInit, the effect of which
 2   1:u   1     is to repaint the screen as it was when the last using program terminated.
 2   1:u   1     If Window Manager is not in *SYSTEM.PASCAL then use WStartup always}
 2   1:u   1
 2   1:u   1    PROCEDURE WInit;
 2   1:u   1    {Initialise Window Manager System}
 2   1:u   1    {just redisplays all windows}
 2   1:u   1    {if Manager is a system unit, all windows survive program changes}
 2   1:u   1
 2   1:u   1  {Routines to create, alter, show, clear, hide and dispose of windows}
 2   1:u   1
 2   1:u   1    FUNCTION  WNew(WatX,WatY,WSizeX,WSizeY:INTEGER;
 2   1:u   5                   WControls:WindowAttributes;
 2   1:u   6                   WHeading:STRING):Window;
 2   1:u   1    {Get new window}
 2   1:u   1
 2   1:u   1    PROCEDURE WAlter(W:Window;
 2   1:u                         WatX,WatY,WSizeX,WSizeY:INTEGER;
 2   1:u                         WControls:WindowAttributes;
 2   1:u                         WHeading:STRING);
 2   1:u   1    {Alter existing window}
 2   1:u   1    {WatX,WatY,SizeX,SizeY -ve means do not alter}
 2   1:u   1    {WControls replaces existing window attributes}
 2   1:u   1    {Window must not be in show when WAlter called}
 2   1:u   1
 2   1:u   1    PROCEDURE WShow(W:Window);
 2   1:u   1    {Display window and set it as "current" one}
 2   1:u   1
 2   1:u   1    PROCEDURE WClearAndShow(W:Window);
 2   1:u   1    {Clear window, then "Show" it}
 2   1:u   1
 2   1:u   1    PROCEDURE WHide(W:Window);
 2   1:u   1    {Remove window from screen - it is not disposed of}
 2   1:u   1
 2   1:u   1    PROCEDURE WDispose(W:Window);
 2   1:u   1    {Dispose of old window}
 2   1:u   1    {Window must not be in show when WDispose called}
 2   1:u   1
 2   1:u   1  {The following procedures all apply to the "current" last shown window}
 2   1:u   1
 2   1:u   1    PROCEDURE WClear;
 2   1:u   1    {Clear Window}
 2   1:u   1
 2   1:u   1    PROCEDURE WClrEOL;
 2   1:u   1    {Clear remainder of current line}
 2   1:u   1
 2   1:u   1    PROCEDURE WClrEOS;
 2   1:u   1    {Clear remainder of window}
 2   1:u   1
 2   1:u   1    PROCEDURE WGotoXY(X,Y:INTEGER);
 2   1:u   1    {Set Window cursor to X,Y}
 2   1:u   1    {X,Y are relative to top left of window - base of 0, excluding heading}
 2   1:u   1
 2   1:u   1    PROCEDURE WWriteCh(Ch:CHAR);
 2   1:u   1    {Write Ch at cursor position in window}
 2   1:u   1    {Non printable chs map to bell}
 2   1:u   1
 2   1:u   1    PROCEDURE WWriteStr(Str:STRING);
 2   1:u   1    {Write Str at cursor position in window}
 2   1:u   1    {MUST NOT CONTAIN NON PRINTABLE CHARS}
 2   1:u   1
 2   1:u   1    PROCEDURE WWriteInt(Int,Width:INTEGER);
 2   1:u   1    {Write Int at cursor posn in window}
 2   1:u   1    {Equivalent to WRITE(Int:Width) in Pascal}
 2   1:u   1    {Width may be 0 (or -ve) to mean as narrow as possible}
 2   1:u   1
 2   1:u   1    PROCEDURE WWriteLn;
 2   1:u   1    {Write newline at cursor position in window}
 2   1:u   1    {If cursor goes below base of window, window is cleared}
 2   1:u   1
 2   1:u   1    PROCEDURE WReadCh(VAR Ch:CHAR;Echo:BOOLEAN);
 2   1:u   1    {Get character from keyboard}
 2   1:u   1    {Window functions can only take place within WReadCh}
 2   1:u   1    {Any non window function ch is returned to user     }
 2   1:u   1    {Echo is controlled by user - non printable chs echo as bell}
 2   1:u   1    {Other Window Reading Procedures - below - use WReadCh}
 2   1:u   1
 2   1:u   1    PROCEDURE WReadLnStr(VAR Str:STRING);
 2   1:u   1    {Get a string from keyboard - echoed}
 2   1:u   1    {String is ended by newLine. Only edit ch allowed is backspace}
 2   1:u   1    {Non printable chs are not returned - but echo as bell}
 2   1:u   1
 2   1:u   1    PROCEDURE WReadLnInt(VAR Int:INTEGER);
 2   1:u   1    {Get an integer from keyboard - echoed}
 2   1:u   1    {Integer is ended by newLine. Only edit ch allowed is backspace}
 2   1:u   1    {Non printable chs are not returned - but echo as bell}
 2   1:u   1
 2   1:u   1    PROCEDURE WReadLn(Echo:BOOLEAN);
 2   1:u   1    {Read up to next newLine from keyboard}
 2   1:u   1    {Non printable chs echo as bell}
 2   1:u   1
 2   1:u   1  {the following functions and procedures are utilities on windows}
 2   1:u   1
 2   1:u   1    FUNCTION  WInWindow(X,Y:INTEGER):Window;
 2   1:u   1    {Returns window in which position X,Y occurs - NoWindow if none}
 2   1:u   1    {X,Y in screen coordinates}
 2   1:u   1
 2   1:u   1    FUNCTION  WChAtXY(X,Y:INTEGER; W:Window):CHAR;
 2   1:u   1    {Return Ch under screen position X,Y in W}
 2   1:u   1    {Space returned if X,Y not in Window, or NoWindow}
 2   1:u   1    {Ch need not be in view at time of call}
 2   1:u   1
 2   1:u   1    PROCEDURE WXY(VAR X,Y:INTEGER);
 2   1:u   1    {Get Coordinates of window manager cursor - in window coordinates}
 2   1:u   1
 2   1:u   1    PROCEDURE WCursorXY(X,Y:INTEGER);
 2   1:u   1    {Set coordinates of window manager cursor - in window coordinates}
 2   1:u   1
 2   1:u   1    FUNCTION WCurrentWindow:Window;
 2   1:u   1    {Return Current Window - one last shown - may be NoWindow}
 2   1:u   1
 2   1:u   1    {HISTORY
 2   1:u   1
 2   1:u   1     Copyright: Austin Tate, ERCC.  All rights reserved.
 2   1:u   1                This program may be used for non-commercial purposes
 2   1:u   1                by users of the UCSD p-System provided that this
 2   1:u   1                copyright notice appears in the source.  Enquiries for
 2   1:u   1                other uses should be directed to the copyright owner.
 2   1:u   1
 2   1:u   1     The Window Manager was originally written in March 1981 by
 2   1:u   1     Austin Tate, ERCC as a demonstration for a course on Office
 2   1:u   1     Systems and Advanced Personal Computers.
 2   1:u   1     -------------------------
 2   1:u   1     It was subsequently modified by Chris Lee while at INMOS up
 2   1:u   1     to 10-Feb-82. The major changes he made were:
 2   1:u   1
 2   1:u   1     I/O optimisation - all I/O is delayed as long as possible, and
 2   1:u   1     is done in as large units as possible, via UNITWRITE (asynch!!!),
 2   1:u   1     and isn't done at all if what we want is on the screen already.
 2   1:u   1     See routines flushoutput and repaint. Repaint certainly pays
 2   1:u   1     its way (try moving and altering windows with the original and
 2   1:u   1     this version), flushoutput almost certainly doesn't - it makes
 2   1:u   1     lines zap out in one swell foop, but stops the dreaded dots.
 2   1:u   1
 2   1:u   1     Window Functions - on the operator interface are all handled
 2   1:u   1     inside the window manager. During input use ESC to toggle between
 2   1:u   1     window manager and input modes (and notice the cursor change when
 2   1:u   1     you do). See WindowFunction. Notice that screen position parameters
 2   1:u   1     are signalled by moving the cursor and typing ESC or <space> (known
 2   1:u   1     as a 'mark' in Window Manager Mode). Eg, to move a window type
 2   1:u   1            ESC                             -- to enter window mode
 2   1:u   1            <move cursor into window to be moved>
 2   1:u   1            M or m                          -- to request a move
 2   1:u   1     ---->  <move cursor to new top left corner position>
 2   1:u   1            <space> or ESC                  -- the window moves at this point
 2   1:u   1            <space> or ESC                  -- to return to input mode
 2   1:u   1
 2   1:u   1     NB at point ----> WindowFunction calls itself recursively. You
 2   1:u   1        can nest window functions (and get into a real mess) if you want).
 2   1:u   1        The interface to WNew has changed to allow the programmer to
 2   1:u   1        control what WindowFunctions the user may apply to a window.
 2   1:u   1
 2   1:u   1     Frames - are drawn by characters. There are CONSTs for the four
 2   1:u   1     corners and four sides so if you have forms drawing chars you
 2   1:u   1     should be able to make things look pretty
 2   1:u   1
 2   1:u   1     Scrolling and Panning - misc minor changes. Scroll by one third
 2   1:u   1     of window depth rather than one line. Only re-pan at input time.
 2   1:u   1     Both these mods are designed to eliminate unnecessary I/O and
 2   1:u   1     repainting.
 2   1:u   1
 2   1:u   1     To improve efficiency WWriteStr assumes that only printable
 2   1:u   1     chars are in the string.
 2   1:u   1
 2   1:u   1     This version was produced during experimentation into window
 2   1:u   1     management. Hence the code hasn't been beautified and bugs
 2   1:u   1     may be present.
 2   1:u   1     -------------------------
 2   1:u   1     The Window Manager as modified by Chris Lee was then altered in
 2   1:u   1     some minor respects by Austin Tate prior to release to the
 2   1:u   1     UCSD p-System Users' Society (USUS) Software Library in February 1982.
 2   1:u   1   }
 2   1:u   1
 2   1:u   1
 2   1:d   1  uses {$u #5:wfiler.code}windowmanager;
 2    :0   0  begin end.

End of Compilation.
```