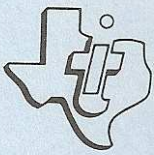


# UCSD p-System<sup>\*</sup>



- Compiler

# UCSD p-System<sup>\*</sup>



- Compiler

# UCSD p-System\*

- Compiler

This manual was developed by staff members of the Texas Instruments Education and Communications Center.

This software is copyrighted 1979, 1981 by the Regents of the University of California, SofTech Microsystems, Inc., Texas Instruments Incorporated, and other copyright holders as identified in the program code. No license to copy this software is conveyed with this product. Additional copies for use on additional machines are available through Texas Instruments Incorporated. No copies of the software other than those provided for in Title 17 of the United States Code are authorized by Texas Instruments Incorporated.

UCSD Pascal and UCSD p-System are trademarks of the Regents of the University of California. Item involved met its quality assurance standards applicable to Version IV.0.



## TABLE OF CONTENTS

<b>GENERAL INFORMATION</b>	9
1.1 Using this Manual	10
1.2 Set-up Instructions	12
1.3 Special Keys	15
<b>UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL</b>	17
2.1 Strings	18
2.2 I/O Intrinsic	21
2.2.1 End of File (EOF)	21
2.2.2 End of Line (EOLN)	21
2.2.3 Files	22
2.2.4 READ and READLN	27
2.2.5 RESET	28
2.2.6 REWRITE	29
2.2.7 WRITE and WRITELN	29
2.2.8 PAGE	30
2.3 Separate Compilation and Memory Management	31
2.3.1 Memory Allocation	31
2.3.2 SEGMENT Routines	31
2.3.3 UNITS	32
2.3.4 EXTERNAL Routines	32
2.4 Concurrent Processes	33
2.5 Texas Instruments Supplied Units	34
2.6 Miscellaneous Differences	35
2.6.1 CASE Statements	35
2.6.2 Comments	35
2.6.3 Extended Comparisons	36
2.6.4 GOTO and EXIT Statements	36
2.6.5 Long Integers	39
2.6.6 Packed Variables	41
2.6.7 Parametric PROCEDURES and FUNCTIONS	44
2.6.8 Program Headings	45
2.6.9 Sets	45
2.6.10 Transcendental Functions	46
2.6.11 Size Limitations	46
<b>PROCEDURES AND FUNCTIONS</b>	47
3.1 ATTACH	48
3.2 BLOCKREAD	49

## TABLE OF CONTENTS

3.3	BLOCKWRITE	50
3.4	CHAIN	51
3.5	CLOSE	52
3.6	CONCAT	53
3.7	COPY	54
3.8	DELETE	55
3.9	EXCEPTION	56
3.10	FILLCHAR	57
3.11	GOTOXY	58
3.12	HALT	59
3.13	INSERT	60
3.14	IORESULT	61
3.15	LENGTH	63
3.16	MARK	64
3.17	MEMAVAIL	65
3.18	MEMLOCK	66
3.19	MEMSWAP	67
3.20	MOVELEFT	68
3.21	MOVERIGHT	69
3.22	POS	70
3.23	PWROFTEN	71
3.24	REDIRECT	72
3.25	RELEASE	73
3.26	SCAN	74
3.27	SEEK	75
3.28	SEMINIT	76
3.29	SIGNAL	77
3.30	SIZEOF	78
3.31	START	79
3.32	STR	80
3.33	TIME	81
3.34	UNITBUSY	82
3.35	UNITCLEAR	83
3.36	UNITREAD	84
3.37	UNITSTATUS	86
3.38	UNITWAIT	87
3.39	UNITWRITE	88
3.40	VARAVAIL	89
3.41	VARDISPOSE	90
3.42	VARNEW	91
3.43	WAIT	92

## TABLE OF CONTENTS

<b>SEGMENTS AND LINKING</b>	<b>93</b>
4.1 Main Memory Management	94
4.2 Separate Compilation	95
4.3 Programming Tactics	96
4.4 SEGMENTS	98
4.5 UNITS	100
4.6 The Linker	107
4.7 The Utility Library	108
<b>CONCURRENT PROCESSES</b>	<b>109</b>
5.1 PROCESSES	110
5.2 Semaphores	112
5.2.1 Mutual Exclusion	113
5.2.2 Synchronization	114
5.3 Other Features	115
<b>TEXAS INSTRUMENTS UNITS</b>	<b>116</b>
6.1 Support Procedures and Functions	117
6.1.1 CHR_DEFAULT	117
6.1.2 SET_PATTERN	117
6.1.3 GET_PATTERN	121
6.1.4 SET_CHR_COLOR	122
6.1.5 SET_SCREEN	123
6.1.6 SET_SCR_COLOR	124
6.1.7 JOY	124
6.2 Random Numbers	125
6.2.1 SET_RND	125
6.2.2 RANDOMIZE	125
6.2.3 RND_INT	126
6.2.4 RND_REAL	126
6.3 Strings	127
6.3.1 BREAK	127
6.3.2 SPAN	128
6.3.3 UPPER_CASE	128
6.4 Sound Processing	129
6.4.1 MAKE_SND_LIST	131
6.4.2 DEL_SND_LIST	132
6.4.3 SND_NOTE	132
6.4.4 SND_TONE	133
6.4.5 WHITE_NOISE	133

## TABLE OF CONTENTS

6.4.6	PERIODIC_NOISE . . . . .	134
6.4.7	SND_VOLUME . . . . .	134
6.4.8	CALL_SND . . . . .	135
6.4.9	GOSUB_SND . . . . .	136
6.4.10	RETURN_SND . . . . .	137
6.4.11	JUMP_SND . . . . .	137
6.4.12	CHAIN_SND . . . . .	137
6.4.13	CHN_SND_CHAIN . . . . .	138
6.4.14	READ_SND_CHAIN . . . . .	138
6.4.15	WRITE_SND_LIST . . . . .	139
6.4.16	READ_SND_LIST . . . . .	139
6.4.17	END_SND . . . . .	140
6.4.18	SET_SND . . . . .	140
6.4.19	PLAY_SND . . . . .	140
6.4.20	PLAY_ALL_SND . . . . .	141
6.4.21	KILL_SND . . . . .	141
6.4.22	KILL_ALL_SND . . . . .	141
6.4.23	SET_SND_TEMPO . . . . .	141
6.4.24	SET_SND_FLAG . . . . .	142
6.4.25	READ_SND_FLAG . . . . .	142
6.4.26	SND_BEAT . . . . .	143
6.4.27	SND_LST_OFFSET . . . . .	143
6.5	Sprite Handling . . . . .	144
6.5.1	SET_SPRITE . . . . .	146
6.5.2	SET_SPR_ATTRIBUTE . . . . .	148
6.5.3	DEL_SPRITE . . . . .	149
6.5.4	SET_SPR_SIZE . . . . .	149
6.5.5	SPRITE_COINC . . . . .	152
6.5.6	PAST_SPRITE_COINC . . . . .	153
6.5.7	GET_SPRITE . . . . .	153
6.6	Speech Handling . . . . .	154
6.6.1	GET_SPEECH . . . . .	154
6.6.2	SAY . . . . .	154
	<b>USING THE COMPILER . . . . .</b>	<b>156</b>
7.1	Compile-time Options . . . . .	158
7.1.1	Compile-Time Option Descriptions . . . . .	159
7.2	Conditional Compilation . . . . .	164



## TABLE OF CONTENTS

<b>APPENDICES</b> . . . . .	166
8.1 Execution Errors . . . . .	167
8.2 I/O Results . . . . .	168
8.3 Device Numbers . . . . .	169
8.4 Pascal Syntax Errors . . . . .	170
8.5 Summary of Differences between UCSD Pascal and Standard Pascal . .	175
8.5.1 String Handling . . . . .	175
8.5.2 I/O Intrinsics . . . . .	175
8.5.3 Memory Management . . . . .	177
8.5.4 Concurrency . . . . .	178
8.5.5 Miscellaneous . . . . .	179
8.5.6 Writing a Transportable Program . . . . .	181
8.6 Summary of Differences between System Versions . . . . .	182
8.6.1 Version IV.0 . . . . .	183
8.7 Converting Programs for use under IV.0 . . . . .	186
8.7.1 Converting Pascal Programs . . . . .	186
8.7.2 Converting Assembly Language Programs . . . . .	194
8.8 Reserved Words . . . . .	196
8.8.1 Standard Pascal Reserved Words . . . . .	196
8.8.2 UCSD Pascal Reserved Words . . . . .	196
8.8.3 Standard Predeclared Identifiers . . . . .	196
8.8.4 UCSD Predeclared Identifiers . . . . .	197
8.9 Assembler Syntax Errors . . . . .	198
8.10 American Standard Code for Information Interchange (ASCII) . . . .	201
8.11 Musical Tone Frequencies . . . . .	202
8.12 Color Codes . . . . .	203
8.13 High-resolution Color Combinations . . . . .	204
8.14 Mathematical Functions . . . . .	206
8.15 List of Speech Words . . . . .	207
8.16 Program Development with Multi-Drive Systems . . . . .	211
8.16.1 Two-Drive System . . . . .	211
8.16.2 Three-Drive System . . . . .	211
 IN CASE OF DIFFICULTY . . . . .	 212
 WARRANTY . . . . .	 213



## SECTION 1: GENERAL INFORMATION

UCSD Pascal<sup>\*</sup> is a powerful, high-level, structured language designed for education, business, scientific, mathematic, and entertainment uses. With the TI Home Computer enhancements to release IV.0, you can use sound, graphics, color, speech, sprites (moving graphics), and wired remote controllers, as well as the standard features of UCSD Pascal.

With the Pascal Compiler, you can compile programs you have written and entered into the p-System using the UCSD p-System<sup>\*</sup> Editor (available separately), and then run the compiled program using the UCSD p-System.

The Pascal Compiler is designed to be used with the Pascal Editor and Filer and at least two disk drives, although it can provide limited use with one disk drive. The Pascal Compiler package contains a diskette labeled Compiler and this manual. The diskette contains the files SYSTEM.COMPIILER, which is the Compiler's p-code, SYSTEM.LIBRARY, which contains the Texas Instruments Units described in Section 6, and SCREENOPS.CODE and COMMANDIO.CODE, which are used by various intrinsics described in Section 3.

The simplest hardware configuration for developing Pascal programs requires the TI Home Computer, the TI Color Monitor (or a video modulator and a television set), the Memory Expansion unit, the p-Code peripheral, and a Disk Memory System with at least one Disk Memory Drive. To enhance your system, you can add Disk Memory Drives, the RS232 Interface, and other peripherals available from Texas Instruments.

The p-System Editor/Filer (described in the UCSD p-System Editor/Filer owner's manual) allows you to create, edit, print, and save files. After a program file has been created and saved on a diskette, you can compile it with the Compiler, and then load and run it as described in the UCSD p-System p-Code manual.

<sup>\*</sup> trademark of the Regents of the University of California.

## GENERAL INFORMATION

### 1.1 USING THIS MANUAL

The Pascal language was introduced around 1970 by its creators, Kathleen Jensen and Niklaus Wirth, both of the Institut fur Informatik, ETH Zurich. Their definitive description of the language is contained in the book Pascal User Manual and Report, 2nd Edition, which is available from Springer-Verlag publishers in New York. This manual assumes that you are familiar with Standard Pascal as described in that book. A good introduction to UCSD Pascal is The UCSD Pascal Handbook, available from Prentice-Hall publishers, Inglewood Cliffs, New Jersey.

This manual describes the differences between Standard Pascal and Release IV.0 of UCSD Pascal, as well as the enhancements which allow access to special Texas Instruments Home Computer abilities. Only the sections of interest to you need to be read in detail. However, you should read all of Section 2 and use that information to refer to Sections 3 and 6.

Section 2, UCSD Pascal Differences from Standard Pascal, describes the extensions to Standard Pascal (as described in the book by Jensen and Wirth) and mentions the few areas in which Standard Pascal has not been supported.

Section 3, Procedures and Functions, includes descriptions of the new procedures and functions that have been added to Standard Pascal in the UCSD version.

Section 4, Segments and Linking, gives details on how to use program segments, including how to link together segments which you have created, enabling the construction of larger programs than can be contained in memory at one time.

Section 5, Concurrent Processes, shows how to keep two sections of code active at one time and how these two sections can interact.

Section 6, Texas Instruments Units, describes the units that support certain features of the Texas Instruments Home Computer. These units allow easy use of character definition, screen control, sound, graphics, color, speech, sprites (moving graphics), and wired remote controllers.

Section 7, Using the Compiler, includes descriptions of the compile-time options and conditional compilation.



## GENERAL INFORMATION

Section 8, Appendices, provides quick reference to technical information. The Appendices include error codes, a summary of the differences between Standard Pascal and UCSD Pascal, and other useful information.

Also of use, and referred to in this manual, is the Internal Architecture Guide, available from

SofTech Microsystems, Inc.  
9494 Black Mountain Road  
San Diego, California 72126

## GENERAL INFORMATION

### 1.2 SET-UP INSTRUCTIONS

The steps involved in creating a program file and accessing the Compiler are included in this section. Please read this material completely before proceeding.

Use your Disk Manager or the Pascal Filer to make a backup copy of the diskette which contains the Pascal Compiler. Use this copy only for your own use. The original should be kept in a safe place.

**Note:** For the recommended placement of files on a multi-disk system, see the Appendix.

1. Be sure that the Memory Expansion unit, the p-Code peripheral, and the Disk Memory System are attached to the computer and turned on. Refer to the appropriate owner's manuals for product details.
2. To create a Pascal program, use the p-System Editor. Insert the Editor diskette into a disk drive.
3. Turn on the monitor and computer console. The p-System promptline appears. **Note:** If you turn on the computer before inserting a diskette in a disk drive, you must insert a diskette and then press **I** to initialize the System before you can proceed.
4. Press **E**, for E(dit, to load the Editor.
5. Refer to the the UCSD p-System Editor owner's manual for detailed directions on entering a program. When you have completed your program, press **Q** for Q(uit. Then press **W** for W(rite.
6. Remove the Editor diskette and insert the diskette on which you wish to save the program. If you have one disk drive, the program must be saved on the diskette that contains the Compiler. If the program is too long to fit on this diskette, then two disk drives are required.
7. Enter the filename for the program and press <return>.
8. Place the diskette that contains the Compiler and the program to be compiled in a disk drive. If you have two or three disk drives, place the diskette that contains the file to be compiled in one of the drives.

9. Press **C**, for C(ompile, to load the Compiler.
10. The screen displays the message

Compiling...

while the Compiler is loaded. If the workfile, SYSTEM.WRK.TEXT, exists, that file is compiled, and the p-code produced is saved as SYSTEM.WRK.CODE and you may proceed to step 11.

If SYSTEM.WRK.TEXT does not exist, the following prompt appears.

Compile?

Enter the location and name of the file which you wish to have compiled. For example, to compile the program TEST.TEXT, which is contained on the diskette in disk drive 2 (#5), enter

#5:TEST

Next the prompt

To what codefile?

appears. Enter the location and name of the file to which you wish the p-code to be saved. For example, if you wish the p-code to be saved as TEST.CODE on the diskette in disk drive 2 (#5), enter

#5:TEST

If you wish the p-code to be saved as SYSTEM.WRK.CODE on the diskette in disk drive 1 (#4), just press <return>.

## GENERAL INFORMATION

11. While the file is being compiled, an account of the progress and any error messages are displayed. The following is the display when a small program named TEST is compiled.

```
Pascal Compiler - Release 99/4 IV.0 Cla-4
<0      >.....
TEST
<5      >....
9       lines compiled
TEST    .
```

A description of the meaning of this display is given in Section 7.

12. When the compiling process is finished, the p-System promptline reappears. You may then compile another program, run the program you compiled, or insert a different diskette and perform some other task.

If you have only one disk drive, the size of the program which you may compile is limited to the memory available on the diskette which contains the Compiler. If you have two disk drives, then the program and p-code may occupy the memory on the second diskette. With three drives, the Compiler can be on one diskette, a large program on a second diskette, and the p-code on a third diskette.



## 1.3 SPECIAL KEYS

In this manual, the keys that you press are indicated by surrounding them with <angle brackets>. The name <return> is used when the Pascal prompts on the screen refer to <return> or <cr> (carriage return). You should press the <ENTER> key. Pressing any key for more than approximately half a second causes that key to be repeated.

To obtain lower-case letters, press the key with the letter on it. To obtain all upper-case letters on the TI-99/4, use the alpha lock toggle to change to upper-case. On the TI-99/4A you may use the alpha lock toggle or press the <ALPHA LOCK> key. To obtain a single upper-case letter on the TI-99/4 when the computer is in lower-case mode, simultaneously press the small space key on the left side of the keyboard or the space bar and the key. On the TI-99/4A, press the key and <SHIFT>.

<u>Name</u>	<u>TI-99/4</u>	<u>TI-99/4A</u>	<u>Action</u>
<del>	SHIFT F	FCTN 1	Deletes a character.
<ins>	SHIFT G	FCTN 2	Inserts a character.
<flush>	SPACE 3	FCTN 3	Stops writing output to the screen.
<break>	SPACE 4	FCTN 4	Stops the program and initializes the System.
<stop>	SPACE 5	FCTN 5	Suspends the program until this key is pressed again.
<alpha lock>	SPACE 6	FCTN 6 or ALPHA LOCK	Acts as a toggle to convert upper-case letters to lower-case and back again.
<screen left>	SPACE 7	FCTN 7	Moves the text displayed on the screen to the left 20 columns at a time.
<screen right>	SPACE 8	FCTN 8	Moves the text displayed on the screen to the right 20 columns at a time.
<line del>	SHIFT Z	FCTN 9	Deletes the current line of information.
{	SPACE 1	FCTN F	Types the left brace.
}	SPACE 2	FCTN G	Types the right brace.
[	SPACE 9	FCTN R	Types the left bracket.
]	SPACE 0	FCTN T	Types the right bracket.
<etx/eof>	SHIFT C	CTRL C	Indicates the end of a file.
<esc>	SPACE .	CTRL .	Tells the program to ignore previous text.
<tab>	SHIFT A	CTRL I	Moves the cursor to the next tab.
<up-arrow>	SHIFT E	FCTN E	Moves the cursor up one line.
<left arrow> or <backspace>	SHIFT S	FCTN S	Moves the cursor to the left one character.

## GENERAL INFORMATION

<u>Name</u>	<u>TI-99/4</u>	<u>TI-99/4A</u>	<u>Action</u>
<right-arrow>	SHIFT D	FCTN D	Moves the cursor to the right one character.
<down-arrow>	SHIFT X	FCTN X	Moves the cursor down one line.
<return>	ENTER	ENTER	Tells the computer to accept the information you type.

## SECTION 2: UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

This section summarizes the areas in which UCSD Pascal differs from Standard Pascal and explains in detail those topics (such as packing) which are exclusive to UCSD Pascal. For a full description of the Pascal language, refer to other appropriate references such as The UCSD Pascal Handbook (Inglewood Cliffs, N.J.: Prentice-Hall). Standard Pascal is defined by the Pascal User Manual and Report (2nd Edition), by Kathleen Jensen and Niklaus Wirth (New York: Springer-Verlag, 1975).

This version of UCSD Pascal differs from other implementations in six general areas.

- **String Handling:** The type `STRING` has been added to the language, along with a number of intrinsic functions for manipulating strings.
- **I/O Ininsics:** A number of intrinsics have been added to facilitate handling of files and peripheral devices. The Standard Pascal I/O (input/output) intrinsics have been slightly modified to make them more appropriate to an interactive environment.
- **Separate Compilation and Memory Management:** The language has been extended by the addition of `SEGMENT` routines, which facilitate swapping of program code at execution time, and `UNITs`, which allow separate compilation of Pascal routines and data structures.
- **Concurrency:** Some syntax extensions have been made, and a few intrinsics added, to support concurrent processes.
- **Texas Instruments UNITs:** Special `UNITs` containing procedures and functions enable easy use of character definition, screen control, sound, graphics, color, speech, sprites (moving graphics), and wired remote controllers.
- **Miscellaneous:** There are a number of small deviations from and extensions to Pascal syntax, as well as limitations imposed by the microprocessor environment.

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

### 2.1 STRINGS

This version of Pascal provides a predeclared type STRING. A variable or constant of type STRING is a sequence of characters. In a STRING variable, the length of the sequence can vary during the execution of a program.

A STRING variable has a maximum length, called the static length, of 255 characters. The default maximum length of a STRING variable is 80 characters, but this default can be overridden by following STRING with the desired maximum length enclosed in square brackets ([ ]). The empty string, with a LENGTH of zero, represented by two single quotes (""), is allowed.

The following examples show STRING declarations.

```
TITLE: STRING;      { Defaults to a maximum length of 80
                     characters. }
NAME: STRING[20];   { Defines the STRING with a maximum of 20
                     characters. }
```

Strings can be manipulated by either Standard Pascal syntax or the special string-handling intrinsics in this Pascal. The intrinsic function LENGTH returns the dynamic length of a string. Values can be assigned to STRINGS using assignment statements, STRING intrinsics, or READLN statements.

The following examples illustrate the use of STRINGS.

```
TITLE := '    THIS IS A TITLE    ';
READLN(COVER);
NAME := COPY(LAST,1,20);
```

The individual characters within a STRING are indexed as a PACKED ARRAY OF CHAR is indexed, from one to the LENGTH of the STRING.

The following examples show two uses of STRING indexing, assuming the variables do not equal the null string.

```
LETTERS[1] := 'A';
TITLE[LENGTH(TITLE)] := 'M';
```



A variable of type STRING may not be indexed beyond its current dynamic LENGTH when range-checking is turned on. Beware especially of strings of length zero. The following sequence results in a "VALUE RANGE ERROR" runtime error.

```
TITLE := '1234';  
TITLE[5] := '5';
```

STRING variables are compatible for assignment and comparison with any other string constant or variable regardless of their static or dynamic length. String constants (but not variables) can be compared with a PACKED ARRAY OF CHAR. String comparisons return a result based on alphabetical ordering. See Section 8.10.

The following program illustrates the comparison of STRING variables.

```
PROGRAM COMPARESTRINGS;  
VAR S: STRING;  
    T: STRING[40];  
BEGIN  S := 'SOMETHING';  
      T := 'SOMETHING BIGGER';  
      IF S = T  
      THEN WRITELN('Strings do not work very well')  
      ELSE  
        IF S > T  
        THEN WRITELN(S, ' is greater than ', T)  
        ELSE  
          IF S < T  
          THEN WRITELN(S, ' is less than ', T);  
      IF S = 'SOMETHING'  
      THEN WRITELN(S, ' equals ', S);  
      IF S > 'SOMETHING'  
      THEN WRITELN(S, ' is greater than SOMETHING');  
      IF S = 'SOMETHING'  
      THEN WRITELN('BLANKS DON'T COUNT')  
      ELSE WRITELN('BLANKS APPEAR TO MAKE A DIFFERENCE');  
      S := 'XXX';  
      T := 'ABCDEF';  
      IF S > T  
      THEN WRITELN(S, ' is greater than ', T)  
      ELSE WRITELN(S, ' is less than or equal to ', T);  
END.
```

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

This program produces the following output.

```
SOMETHING is less than SOMETHING BIGGER
SOMETHING equals SOMETHING
SOMETHING is greater than SAMETHING
BLANKS APPEAR TO MAKE A DIFFERENCE
XXX is greater than ABCDEF
```

One of the most common uses of STRING variables in this Pascal is reading file names from the CONSOLE: as in the following program segment.

```
PROGRAM LISTER;
VAR BUFFER: PACKED ARRAY[0..511] OF CHAR;
    FILENAME: STRING;
    F: FILE;
BEGIN
    WRITELN('Enter the filename of the file');
    WRITE('to be listed --->');
    READLN(FILENAME);
    RESET(F,FILENAME);
    WHILE NOT EOF(F) DO
        BEGIN
            ... { Code to use the file. }
        END;
    END.
```

The Pascal intrinsics READ and READLN read characters one at a time into a STRING variable, up to but not including the end of line (<return>). Thus, only one STRING can be read for each line of the input file.

For example, the single statement READLN(S1,S2) is equivalent to the two-statement sequence READ(S1); READLN(S2). In both cases the STRING variable S2 is assigned the empty string. READ and READLN are described in Section 2.2.4.

The string-handling intrinsics are CONCAT, COPY, DELETE, INSERT, LENGTH, and POS. Descriptions of these intrinsics are in Section 3.

## **2.2 I/O INTRINSICS**

UCSD Pascal is designed for interactive use. Therefore some of the Standard I/O intrinsics have been altered and others added. In addition to the changes from Standard Pascal described in this section, refer to the I/O intrinsics which have been added to UCSD Pascal. These are BLOCKREAD, BLOCKWRITE, CLOSE, IORESULT, UNITCLEAR, UNITREAD, UNITSTATUS, and UNITWRITE, all described in Section 3.

### **2.2.1 End of File (EOF)**

To set EOF TRUE for a text file being entered from the CONSOLE:, press <etx>. Also refer to the section on SETUP in the UCSD p-System Utilities owner's manual.

If a file F is closed, EOF(F) returns the value TRUE. For a TEXT file, EOF(F) being TRUE implies that EOLN(F) is also TRUE. After a RESET(F), EOF(F) is FALSE. If EOF(F) becomes TRUE during a GET(F) or a READ(F,...), the data obtained is not valid.

When your program starts executing, the System performs a RESET on the predeclared files INPUT, OUTPUT, and KEYBOARD. The predeclared file KEYBOARD is described in Section 2.2.3.1.

EOF and EOLN refer to the file INPUT unless the name of another file is given as their first parameter.

### **2.2.2 End of Line (EOLN)**

EOLN(F) is defined only if the contents of F are of type CHAR. EOLN(F) becomes TRUE only after reading a <return> from file F. As with EOF, EOLN refers to the Standard file INPUT if the first parameter is not the name of a file.

The following example shows the importance of typing a <return> at the proper time. Entry data for this program consists of integers separated by spaces. To end entry and set EOLN(F) to TRUE, press <return> immediately after the last digit of the last integer on a line. If a space precedes <return>, EOLN remains FALSE and another READ takes place.

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

```
PROGRAM ADDLINES;
VAR K,SUM: INTEGER;
BEGIN
  WHILE NOT EOF(INPUT) DO
    BEGIN
      SUM := 0;
      READ(INPUT,K);
      WHILE NOT EOLN(INPUT) DO
        BEGIN
          SUM := SUM+K;
          READ(INPUT,K);
        END;
      SUM := SUM+K;
      WRITELN(OUTPUT);
      WRITELN(OUTPUT,'THE SUM FOR THIS LINE IS ',SUM);
    END;
  END.
```

Press <etx> to stop program execution.

### **2.2.3 Files**

The file type INTERACTIVE and files without a type have been added to Standard Pascal. Files cannot be declared inside structured variables.

#### **2.2.3.1 INTERACTIVE Files**

Like files of type TEXT, files of type INTERACTIVE are composed of characters. INTERACTIVE files differ from TEXT files in their behavior when they are used by the intrinsics READ, READLN, and RESET. Files that have types other than INTERACTIVE behave as in Standard Pascal.

The Standard predeclared files INPUT and OUTPUT are defined to be INTERACTIVE. The file KEYBOARD, which is predeclared in this Pascal, is also INTERACTIVE.

INPUT defaults to CONSOLE:. The statement READ(INPUT,CH) where CH is a character variable, echos the character typed from CONSOLE: back to CONSOLE:.

WRITE statements default to OUTPUT, causing the output to appear on CONSOLE:.

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

KEYBOARD is the non-echoing equivalent of INPUT. For example, the following two statements are equivalent to the single statement READ(INPUT,CH).

```
READ(KEYBOARD,CH);  
WRITE(OUTPUT,CH);
```

For an explanation of "redirecting" the Standard files INPUT and OUTPUT, see the UCSD p-System P-Code Peripheral owner's manual.

Suppose that you have made the following declarations.

```
VAR CH: CHAR;  
    F: TEXT; { Type TEXT is a FILE OF CHAR. }
```

Then the statement READ(F,CH) is defined in Standard Pascal to be equivalent to the following two-statement sequence.

```
CH := F^; { Standard }  
GET(F);   { method. }
```

In other words, the Standard definition of READ requires that opening a file must load the file window variable F with the first character of the file. In an interactive programming environment it is not convenient to type the first character of an input file at the time the file is opened, because then every program using files would wait until a character was typed whether or not the program performed any input operations.

The INTERACTIVE file type has been defined in UCSD Pascal to overcome this problem. Declaring a file F to be of type INTERACTIVE is equivalent to declaring F to be of type TEXT, except that READ(F,CH) on an INTERACTIVE file is the reverse of the sequence specified by the Standard definition for files of type TEXT.

```
GET(F); { UCSD Pascal }  
CH := F^; { method. }
```

This difference affects the way in which EOLN must be used when reading from a text file of type INTERACTIVE. As described above, EOLN becomes TRUE only after reading the end of line character (<return>). When <return> is read, EOLN is TRUE, and the character returned as a result of the READ is a blank.

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

On a Standard file, RESET(F) performs an immediate GET(F). This does not happen if the file is INTERACTIVE. Thus, on an INTERACTIVE file, the equivalent of a Standard RESET is the following two-statement sequence.

```
RESET(F);  { Makes INTERACTIVE }  
GET(F);    { look like TEXT.      }
```

Refer to Section 2.2.4 on READ and READLN and Section 2.2.5 on RESET for more details.

### **2.2.3.2 Files without a Type**

This version of Pascal allows files to be declared without a type. An untyped file F can be thought of as a file without a window variable F. With such a file, all I/O must be performed with the functions BLOCKREAD and BLOCKWRITE, described in Section 3. Any number of blocks can be transferred with either BLOCKREAD or BLOCKWRITE. They return the number of blocks transferred.

The following program reads a diskette file called "SOURCE.DATA" and copies the file into another diskette file called "DESTINATION" using untyped files and the intrinsics BLOCKREAD and BLOCKWRITE. See the notes in Sections 3.2 and 3.3 when using BLOCKREAD and BLOCKWRITE.

```
PROGRAM FILEDEMO;
VAR  BLOCKNUMBER,BLOCKSTRANSFERRED: INTEGER;
      BADIO: BOOLEAN;
      G,F: FILE;
      BUFFER: PACKED ARRAY[0..511] OF CHAR;
BEGIN
  BADIO := FALSE;
  RESET(G,'SOURCE.DATA');
  REWRITE(F,'DESTINATION');
  BLOCKNUMBER := 0;
  {$I-}      { This turns off I/O checking. }
  BLOCKSTRANSFERRED := BLOCKREAD(G,BUFFER,1,BLOCKNUMBER);
  WHILE ((IORESULT=0) AND (NOT BADIO) AND
(BLOCKSTRANSFERRED=1) DO
    BEGIN
      BLOCKSTRANSFERRED := BLOCKWRITE(F,BUFFER,1,BLOCKNUMBER);
      BADIO := ((BLOCKSTRANSFERRED<1) OR (IORESULT<>0));
      BLOCKNUMBER := BLOCKNUMBER+1;
      BLOCKSTRANSFERRED := BLOCKREAD(G,BUFFER,1,BLOCKNUMBER);
    END;
  BLOCKSTRANSFERRED := BLOCKWRITE(F,BUFFER,1,BLOCKNUMBER);
  CLOSE(F,LOCK);
END.
```

### **2.2.3.3 Random Access of Files**

Files can be randomly accessed with the UCSD Pascal intrinsic SEEK. The parameters for SEEK are the file identifier and an integer specifying the record number to which the window should be moved. The first record of a structured file is record number zero.

Attempts to PUT records beyond the physical end of file set EOF TRUE. (The physical end of file is the point at which the next record in the file would overwrite another file on the diskette.) SEEK always sets EOF and EOLN to FALSE. A subsequent GET or PUT sets these conditions as appropriate. SEEK is described in Section 3.

The following sample program demonstrates the use of SEEK to access and update records in a file randomly.

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

```
PROGRAM RANDOMACCESS;
VAR  RECNUMBER: INTEGER;
     CH: CHAR;
     DISK: FILE OF RECORD
         NAME: STRING[20];
         DAY,MONTH,YEAR: INTEGER;
         ADDRESS: PACKED ARRAY[0..49] OF CHAR;
END;
BEGIN
  RESET(DISK, 'RECORDS.DATA');
  WHILE NOT EOF(INPUT) DO
    BEGIN
      WRITE(OUTPUT, 'Enter record number --->');
      READLN(INPUT, RECNUMBER);
      SEEK(DISK, RECNUMBER);
      GET(DISK);
      WITH DISK DO
        BEGIN
          IF NOT EOF(DISK)
            THEN WRITELN(OUTPUT, NAME, DAY, MONTH, YEAR, ADDRESS)
            ELSE WRITELN('New Record');
          WRITE(OUTPUT, 'Enter correct name --->');
          READLN(INPUT, NAME);
          ... { Code to use the information obtained. }
        END;
      { Must point the window back to the record since
        GET(DISK) advances the window to the next record after
        loading DISK. }
      SEEK(DISK, RECNUMBER);
      PUT(DISK);
    END;
  END.
```

### **2.2.3.4 Files as Elements of Records or Arrays**

This version of Pascal does not allow files to be declared inside structured variables such as arrays or records. Consequently, file variables cannot be stored on the Heap. This restriction is imposed so that the Compiler can easily produce hidden code to open and close an internal file at the proper limits of its scope.



#### 2.2.4 READ and READLN

Strings are read character-by-character until terminated by your pressing <return>. When integers are read, leading blanks and end-of-lines are ignored until a non-blank character is read. An integer is terminated by a space (" "), a character that is not a digit, or a <return>. Before a string has been completely read, it can be corrected by backspacing over it and retyping.

Real values are read in the same way as integers. Neither Boolean values nor any structured type can be read.

The behavior of READ and READLN conforms to the definition in Standard Pascal except when handling files that are INTERACTIVE. The Standard file INPUT is defined to be INTERACTIVE in UCSD Pascal. The action of READ on an INTERACTIVE file is described below.

In the following example, the left fragment is taken from Standard Pascal with only the RESET and REWRITE statements altered. This program correctly copies the text file X to text file Y. The program fragment on the right performs a similar task, except that the source file being copied is INTERACTIVE, thus forcing a slight change in the program in order to produce the desired result.

```
PROGRAM STANDARD;
VAR X,Y:TEXT;
    CH:CHAR;
BEGIN
    RESET(X,'SOURCE.TEXT');
    REWRITE(Y,'SOMETHING.TEXT');

    WHILE NOT EOF(X) DO
        BEGIN
            WHILE NOT EOLN(X) DO
                BEGIN
                    READ(X,CH);
                    WRITE(Y,CH);
                END;
            READLN(X);
            WRITELN(Y);
        END;
    CLOSE(Y,LOCK);
END.
```

```
PROGRAM UCSD_VERSION;
VAR X,Y:INTERACTIVE;
    CH:CHAR;
BEGIN
    RESET(X,'CONSOLE:');
    REWRITE(Y,'SOMETHING.TEXT');
    READ(X,CH);
    WHILE NOT EOF(X) DO
        BEGIN
            WHILE NOT EOLN(X) DO
                BEGIN
                    WRITE(Y,CH);
                    READ(X,CH);
                END;
            READLN(X);
            WRITELN(Y);
        END;
    CLOSE(Y,LOCK);
END.
```

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

Note that text files X and Y in both programs had to be opened with the extended form of the Standard procedures RESET and REWRITE.

The CLOSE intrinsic (a new intrinsic; see Section 3) was applied to file Y in both versions of the program to make it a permanent file in the diskette directory called "SOMETHING.TEXT". Text file X could have been a diskette file instead of coming from CONSOLE: in the right-hand version of the program.

### 2.2.5 RESET

The Standard procedure RESET(F) resets the file window to the beginning of file F. The next GET(F) or PUT(F) affects record number zero of that file. An immediate GET(F) is also performed within RESET (thus getting the first record of the file), unless file F is INTERACTIVE.

Thus, for INTERACTIVE files, the equivalent of the Standard definition of RESET(F) is the following two-statement sequence.

```
RESET(F); { Makes an INTERACTIVE file }
GET(F);   { look like a TEXT file. }
```

Except for this stipulation about INTERACTIVE files, the behavior of RESET is as in Standard Pascal.

UCSD Pascal also allows RESET to have a second parameter, which is the name of an existing diskette file or device, contained in a string constant or string variable. The diskette file (or device) is referred to as an "external" file, while a file that is a data object in a Pascal program is called an "internal" file.

The following statements associate the file pointer F with the external (diskette) text file "ODD" or the diskette file named in the string variable FNAME.

```
RESET(F, 'ODD.TEXT')
RESET(F, FNAME)
```

Trying to RESET a nonexistent external file or an internal file that is already open causes an I/O error. Trying to RESET a write-only device, such as PRINTER:, causes an I/O error since the device is not an input device, and the GET that RESET implicitly performs attempts to read the device.

External files that are opened by a program with RESET or REWRITE can be closed with the intrinsic CLOSE (see Section 3).

### **2.2.6 REWRITE**

The intrinsic REWRITE "clears" a file by setting F to the empty file and EOF(F) to TRUE. A call to REWRITE can also be used to open a new file.

In this Pascal, the REWRITE intrinsic can be called with a second parameter which is the name of a diskette file (as in RESET) contained in a string constant or a string variable.

If the diskette file is named, it can be either an existing file or a new file. If it is new, a file of the appropriate type is created on the diskette. If it already exists, REWRITE creates a temporary file which can replace the old file, be saved under a new name, or be discarded. See the CLOSE intrinsic in Section 3.

If there is no second parameter, REWRITE(F) is equivalent to REWRITE(F,'F').

Trying to REWRITE an already open internal file causes an I/O error.

Aside from the provision for binding an internal file to an external file name, REWRITE behaves as defined in Standard Pascal.

### **2.2.7 WRITE and WRITELN**

In UCSD Pascal, WRITE and WRITELN can write values of type INTEGER, REAL, STRING, and PACKED ARRAY OF CHAR. BOOLEANs, other types of arrays, and other structured types cannot be output.

WRITE and WRITELN can write an entire PACKED ARRAY OF CHAR in a single WRITE statement, as illustrated by the following statements.

```
VAR BUFFER: PACKED ARRAY[0..10] OF CHAR;  
BEGIN  
  BUFFER := 'HELLO THERE'; {Contains exactly 11 characters.}  
  WRITELN(OUTPUT, BUFFER);  
END.
```

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

Field width specifications also apply to STRINGS. When a string variable or constant is written without specifying a field width, the actual number of characters written is equal to the dynamic length of the string. If the field width specified is longer than the dynamic length of the string, leading blanks are inserted and written. If the field width is smaller than the dynamic length of the string, the excess characters are truncated on the right. These characteristics are illustrated below.

```
PROGRAM WRITESTRINGS;  
VAR S:STRING;  
BEGIN  
  S := 'THE BIG BROWN FOX JUMPED...';  
  WRITELN(S);  
  WRITELN(S:30);  
  WRITELN(S:10);  
END.
```

This program produces the following output.

```
THE BIG BROWN FOX JUMPED...  
  THE BIG BROWN FOX JUMPED...  
THE BIG BR
```

### 2.2.8 PAGE

In UCSD Pascal, the intrinsic PAGE sends a formfeed character to a file or device specified in a parameter. For example, PAGE(F); sends a formfeed character to the file specified by F. PAGE(OUTPUT); sends a formfeed character to the screen and clears the display.

## **2.3 SEPARATE COMPILATION AND MEMORY MANAGEMENT**

UCSD Pascal allows separate compilation and memory management, which are discussed in detail in Section 4. This section shows only the syntax of particular extensions.

### **2.3.1 Memory Allocation**

The Standard procedures DISPOSE and NEW are implemented and the MARK/RELEASE mechanism used in earlier versions of UCSD Pascal is still supported. In addition, the following intrinsics are provided as aids to memory management: MEMAVAIL, VARAVAIL, VARDISPOSE, and VARNEW. These are described in Section 3. If you intend to make much use of direct control of memory resources, you should refer to the Internal Architecture Guide.

**Note:** If you use the NEW intrinsic to allocate space for a record with a particular variant record, you must DISPOSE of that record using the same variant. Otherwise, you risk damaging the Heap and crashing the System. Similarly, it is crucial that MARKs and RELEASEs be properly paired. The contents of a MARKed pointer must not be altered until the matching call to RELEASE has been performed, and RELEASEs must only be performed on variables that are MARKed but not yet RELEASEd.

### **2.3.2 SEGMENT Routines**

Routines (procedures, functions, or processes) normally occupy the same code segment as the compilation unit in which they appear, but a segment routine occupies a code segment of its own. Code is swapped into memory one segment at a time; the space a segment occupies in memory becomes available to other programs as soon as it is no longer in use. Thus, declaring routines such as a program's initialization and termination routines as segment routines may improve a program's memory use.

To define a segment routine, begin its declaration with the reserved word SEGMENT, as shown below.

```
SEGMENT PROCEDURE ONE;  
  BEGIN  
    PRINT('SEGMENT NUMBER ONE');  
  END;
```

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

More information about segment routines, including some restrictions on the way in which they must be declared, appears in Section 4.

### **2.3.3 UNITS**

UNITs are used to compile Pascal routines and data structures separately from the main program. This is helpful in preparing long programs that compile slowly, in coordinating the efforts of several programmers using common facilities, or for producing a set of standard routines that perform commonly required functions.

### **2.3.4 EXTERNAL Routines**

A Pascal host can use an assembly language routine that is assembled separately. The host must include a Pascal routine heading (with parameters, if there are any) and designate it as EXTERNAL, as shown below.

```
FUNCTION FAST (SPEED: INTEGER): BOOLEAN; EXTERNAL;  
PROCEDURE WRITE_OUT; EXTERNAL;
```

Assembled routines used by a Pascal host must strictly adhere to Pascal calling conventions and System constraints on resources such as memory and registers. (See the UCSD p-System Pascal Assembler and Linker owner's manuals for more details.) Before you run a host which uses external routines, the routines must be bound to the host's code by using the Linker, as described in the Pascal Assembler and Linker manuals.

## 2.4 CONCURRENT PROCESSES

In this Pascal, you can declare a PROCESS. A process declaration is similar to a procedure declaration, as shown in the following example.

```
PROCESS BARN (VAR COW: REAL);
```

A process is a routine whose execution appears to proceed at the same time as the main program. Processes are initiated by the intrinsic START (see Section 3). START has some optional parameters which allow you to specify the space allocation and priority of a process.

The predeclared type SEMAPHORE allows concurrent processes to communicate with each other. Semaphores are initialized by the intrinsic SEMINIT and managed by the intrinsics SIGNAL and WAIT.

These intrinsics are described in Section 3, and concurrent processes are discussed more fully in Section 5.

## 2.5 TEXAS INSTRUMENTS SUPPLIED UNITS

The TI Home Computer has many capabilities that are not easily available with Pascal statements. These include definition of characters, sound, sprites, and speech. These capabilities are available as functions and procedures in specially written UNITS contained in SYSTEM.LIBRARY.

These UNITS are SUPPORT, RANDOM, MISC, SOUND, BEEP, SPRITE, and SPEECH. To access the functions and procedures within the UNITS, include a statement in your program which consists of USES followed by the name of the UNIT used by the program.

SUPPORT allows you to set character colors and screen colors, define patterns, obtain character patterns, turn the screen off, and set the display mode (pattern, multi-color, or text).

RANDOM provides for generation of pseudo-random numbers.

MISC lets you determine the values in strings, and change strings to all upper-case letters.

SOUND can be used to create a broad spectrum of notes and noises and coordinate those sounds with your program.

BEEP is a subset of the UNIT SOUND. It allows the use of basic sounds and takes less memory than SOUND.

SPRITE permits you to create and delete sprites (moving graphics), adjust their size and speed, and determine when they are coincident.

SPEECH allows you to use speech when the Solid State Speech <sup>TM</sup> Synthesizer, sold separately, is attached to the console.

For sprites and sounds, the procedures allow you to set up a complex sequence of instructions that are performed concurrently with program execution.

The procedures and functions available in each of these UNITS are described in Section 6.



## 2.6 MISCELLANEOUS DIFFERENCES

Several miscellaneous additions to and alterations of Standard Pascal have been made. They affect the use of CASE statements, comments, comparisons, the GOTO statement, the use of INTEGERS, packed variables, parameters, program headings, sets, and transcendental functions.

### 2.6.1 CASE Statements

In UCSD Pascal, CASE statements "fall through" if there is no label equal to the case selector. When this happens, the statement following the CASE statement is executed next.

For example, the following program only outputs the line "THAT'S ALL FOLKS" since the case statement "falls through" to the WRITELN statement following the case statement, as shown below.

```
PROGRAM FALLTHROUGH;
VAR CH:CHAR;
BEGIN
  CH := 'A';
  CASE CH OF
    'B': WRITELN(OUTPUT, 'HI THERE');
    'C': WRITELN(OUTPUT, 'THE CHARACTER IS A 'C'');
  END;
  WRITELN(OUTPUT, 'THAT'S ALL FOLKS');
END.
```

### 2.6.2 Comments

The Compiler considers a comment to be any text appearing between either the symbols "(" and ")" or the symbols "{" and "}". Text appearing between these symbols is ignored by the Compiler unless the first character is a dollar sign, in which case the text is interpreted as a Compiler control comment, as described in Section 7.

If the beginning of the comment is marked with the "(" symbol, the end of the comment must be marked with the matching ")" symbol, rather than the "}" symbol. When the comment begins with the "{" symbol, the comment continues until the

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

matching "]" symbol appears. This feature allows you to "comment out" a section of a program which itself contains comments, as shown below.

```
{ XCP := XCP + 1; (* Adjust for special case. *) }
```

The Compiler does not keep track of nested comments. When a comment symbol is encountered, the text is scanned for the matching comment symbol. Therefore, the following text results in a syntax error.

```
{This is a comment { Nested comment. } End of first comment.}  
^ Error here.
```

### **2.6.3 Extended Comparisons**

UCSD Pascal allows equal (=) and not equal (<>) comparisons of any array or record structure.

### **2.6.4 GOTO and EXIT Statements**

A GOTO statement causes a "jump" in the flow of control of a program. The next statement executed is the statement with the label named in the GOTO statement, and execution proceeds from that point. In UCSD Pascal, the label and the GOTO statement must be within the same routine or within the same main program block. This is a more restricted form of the GOTO statement than in the Standard language.

EXIT is an extension which accepts as its single parameter the identifier of a routine to be exited, the identifier of a program, or the reserved word PROGRAM. EXIT causes the routine or program it names to be stopped immediately. The addition of the EXIT statement to Pascal was inspired by the occasional need for a straightforward means to stop a complicated and possibly deeply nested series of procedure calls if an error occurs.

Using an EXIT statement to leave a FUNCTION can result in the FUNCTION returning undefined values if no assignment has been made to the FUNCTION identifier before the execution of the EXIT statement.

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

An example of the EXIT statement is shown below.

```
PROGRAM EXITDEMO;
VAR T: STRING;
    CN: INTEGER;
PROCEDURE Q; FORWARD;
PROCEDURE P;
BEGIN
    READLN(T);
    WRITELN(T);
    IF T[1]='#'
        THEN EXIT(Q);
    WRITELN('LEAVE P');
END;
PROCEDURE Q;
BEGIN
    P;
    WRITELN('LEAVE Q');
END;
PROCEDURE R;
BEGIN
    IF CN <= 10
        THEN Q;
    WRITELN('LEAVE R');
END;
BEGIN
    CN := 0;
    WHILE NOT EOF DO
        BEGIN
            CN := CN+1;
            R;
            WRITELN;
        END;
END.
```

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

Assume that the following are the three inputs for the program.

```
THIS IS THE FIRST STRING <return>
# <return>
LAST STRING <ETX>
```

Then the following output results.

```
THIS IS THE FIRST STRING
LEAVE P
LEAVE Q
LEAVE R

#
LEAVE R

LAST STRING
LEAVE P
LEAVE Q
LEAVE R
```

The EXIT(Q) statement causes PROCEDURE P and Q to stop. Processing continues following the call to Q inside PROCEDURE R. Thus, the only line of output following "#" is "LEAVE R" at the end of PROCEDURE R. In the two cases where the EXIT(Q) statement is not executed, processing proceeds normally through the ends of procedures P and Q.

If the procedure identifier passed to EXIT is a recursive procedure, the most recent invocation of that procedure is exited. If, in the above example, one or both of the procedures P and Q declared and opened some local files, an implicit CLOSE(F) is performed when the EXIT(Q) statement is executed, as though the procedures P and Q had stopped normally.

The EXIT statement can also be used to exit a Pascal program by EXIT(PROGRAM) or EXIT(programname).

### 2.6.5 Long Integers

In UCSD Pascal, an optional length attribute can be included with the predeclared type INTEGER. INTEGERS declared in this way are called LONG INTEGERS. They are intended for business, scientific, or other applications which need extended number lengths with complete accuracy. The usual limit for variables of type INTEGER is from -32,768 to 32,767.

This extension supports the four basic Standard INTEGER arithmetic operations (addition, subtraction, multiplication, and division), as well as routines that facilitate conversion to strings and Standard INTEGERS. Input/Output, in-line declaration of constants, and inclusion in structured types are all fully supported and function as they do with Standard INTEGERS.

LONG INTEGERS are declared with the Standard identifier INTEGER followed by a length attribute in square brackets. This length is an unsigned number, no greater than 36, which specifies the minimum number of decimal digits the LONG INTEGER must be able to represent.

The following example defines X as an integer with a minimum of eight digits.

```
VAR X: INTEGER[8];
```

Constants are defined in the normal manner, as shown below.

```
CONST LARGE = 79413034;
```

Because of its magnitude, LARGE is by default a LONG INTEGER and can be used anywhere a LONG INTEGER is acceptable.

Make sure that sufficient memory has been allocated by the declared length attribute to store the result of assignment or arithmetic expression statements. INTEGER expressions are implicitly converted to LONG INTEGERS as required by the space demands of an operation or assignment. The reverse is not true. Some examples of conversions are as follows.

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

```
VAR I: INTEGER;
    L: INTEGER[N]; { Where N is an integer constant <= 36. }
    S: REAL;

I := L; { Syntax error, see TRUNC(L). }
L := -L; { Correct. }
L := I; { Always correct. }
L := S; { Never accepted. }
S := L; { Never accepted. }
```

Arithmetic operations which can be used in conjunction with LONG INTEGERS are +, -, \*, DIV, and unary plus/minus. On assignment, the length of the LONG INTEGER is adjusted during execution to the declared length attribute of the destination variable. Overflow may result if the destination variable is not large enough to hold the source.

The comparisons =, <, >, <=, >=, and <> can be used in expressions that contain both LONG INTEGERS and INTEGERS.

The function TRUNC accepts both LONG INTEGERS and REALs as arguments. Thus TRUNC(L), where L is a LONG INTEGER, converts L to an INTEGER. Overflow results if L is greater than 32,767 or less than -32,768.

The procedure STR(L,S) converts the INTEGER or LONG INTEGER L into a string (complete with minus sign if needed), and places it in the STRING S. The following program fragment illustrates a suitable "dollar and cent" routine.

```
STR(L,S);
INSERT('.',S,LENGTH(S)-1);
WRITELN(S);
```

Pascal syntax requires that parameter types be specified by type identifiers. Therefore, attempting to use an "INTEGER[<length>]" style declaration in a parameter list results in a syntax error, which can be prevented by declaring an appropriate type identifier, as illustrated below.

```
TYPE LONG = INTEGER[18];
PROCEDURE BIGNUMBER(BANKACCT: LONG);
```

### **2.6.6 Packed Variables**

Arrays and records can be packed in UCSD Pascal, with some limitations. Packed records require less room in memory and on diskettes. However, extra execution time is required for the packing and unpacking.

The Standard intrinsics PACK and UNPACK are not supported because these functions are performed automatically on each element of a type or variable that is declared as packed.

#### **2.6.6.1 Packed Arrays**

This Pascal packs arrays and records if the ARRAY or RECORD declaration is preceded by the word PACKED. For example, consider the following declarations.

```
A: ARRAY[0..9] OF CHAR;  
B: PACKED ARRAY[0..9] OF CHAR;
```

The array A occupies ten 16-bit words of memory with each element of the array occupying one word. The PACKED ARRAY B, on the other hand, occupies only five words since each 16-bit word contains two eight-bit characters. Therefore, each element of the PACKED ARRAY B is eight bits long.

PACKED ARRAYs need not be restricted to arrays of type CHAR, as shown below.

```
C: PACKED ARRAY[0..1] OF 0..3;  
D: PACKED ARRAY[1..9] OF SET OF 0..15;  
E: PACKED ARRAY[0..239,0..319] OF BOOLEAN;
```

Each element of PACKED ARRAY C is only two bits long, since only two bits are needed to represent the values in the range 0..3. Therefore, C occupies only one 16-bit word of memory, and 12 of the bits in that word are unused. PACKED ARRAY D is a nine-word array, since each element of D is a SET which can be represented in a minimum of 16 bits. Each element of a PACKED ARRAY OF BOOLEAN, as in the case of E in the above example, occupies only one bit.

The word PACKED must occur before the last use of ARRAY for an array to be packed. Therefore, the following two declarations are not equivalent.

```
F: PACKED ARRAY[0..9] OF ARRAY[0..3] OF CHAR;  
G: PACKED ARRAY[0..9,0..3] OF CHAR;
```

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

The second occurrence of the reserved word ARRAY in the declaration of F causes the packing option in the Compiler to be turned off so that F becomes an unpacked array of 40 words. On the other hand, the PACKED ARRAY G occupies 20 total words, because the word ARRAY occurs only once in the declaration. If F had been declared as

```
F: PACKED ARRAY[0..9] OF PACKED ARRAY[0..3] OF CHAR;
```

or as

```
F: ARRAY[0..9] OF PACKED ARRAY[0..3] OF CHAR;
```

G and F would have had identical configurations.

PACKED only has true significance before the last appearance of the word ARRAY in a declaration of a PACKED ARRAY. When in doubt, place the word PACKED before every appearance of the word ARRAY to ensure that the resulting array is PACKED.

The resulting array is only packed if the final type of the array is a scalar, subrange, or set which can be represented by eight bits or less. The following declaration results in no packing because the final type of the array cannot be represented in a field of eight bits.

```
H: PACKED ARRAY[0..3] OF 0..1000;
```

Therefore, H is an array which occupies four 16-bit words.

Packing never occurs across word boundaries. Thus, if the type of the element to be packed requires a number of bits that do not divide evenly into 16, there are some unused bits at the high-order end of each of the words comprising the array.

For the purposes of assignment and comparison, a string constant is compatible with a PACKED ARRAY OF CHAR but not with an unpacked ARRAY OF CHAR. In a similar fashion, no packed array or record can be assigned to or compared with an unpacked version of the same type.

Initialization of a PACKED ARRAY OF CHAR can be accomplished very efficiently with the intrinsics FILLCHAR and SIZEOF.



```
PROGRAM FILLFAST;  
VAR A: PACKED ARRAY[0..10] OF CHAR;  
BEGIN  
    FILLCHAR(A[0],SIZEOF(A),'A');  
    WRITELN(A);  
    FILLCHAR(A[0],SIZEOF(A),'B');  
    WRITELN(A);  
END.
```

The above sample program fills the entire PACKED ARRAY A with blanks. Refer to the descriptions of FILLCHAR and SIZEOF in Section 3.

#### 2.6.6.2 Packed Records

The following RECORD specification declares a RECORD with four fields. The entire RECORD occupies one 16-bit word, because it is declared as a PACKED RECORD.

```
VAR R:  PACKED RECORD  
        I,J,K: 0..31;  
        B: BOOLEAN  
    END;
```

The variables I, J, and K each occupy five bits in the word. The Boolean variable B is allocated the sixteenth bit of the same word.

Just as PACKED ARRAYs can be multidimensional, PACKED RECORDs can contain fields which themselves are PACKED RECORDs or PACKED ARRAYs. Again, slight differences in the way in which declarations are made affect the degree of packing achieved. For example, the following declarations are not equivalent.

VAR A:	VAR B:
PACKED RECORD	PACKED RECORD
C: INTEGER;	C: INTEGER;
F: PACKED RECORD	F: RECORD
R: CHAR;	R: CHAR;
K: BOOLEAN	K: BOOLEAN
END;	END;
H: PACKED ARRAY[0..3] OF CHAR	H: PACKED ARRAY[0..3] OF CHAR
END;	END;

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

As with packed arrays, the word PACKED must appear with every occurrence of the reserved word RECORD in order for the PACKED RECORD to retain its packed qualities throughout all fields of the RECORD. In the above example, only RECORD A has all of its fields packed into one word. In B, the F field is not packed and therefore occupies two 16-bit words. A packed or unpacked ARRAY or RECORD which is a field of a PACKED RECORD always starts at the beginning of the next word boundary. Thus, in the case of A, even though the F field does not completely fill one word, the H field starts at the beginning of the next word boundary.

When a record, either packed or unpacked, contains a case variant, the field is allocated enough space to contain the largest variant. Consider the following example.

```
VAR K: PACKED RECORD
    B: BOOLEAN;
    CASE F: BOOLEAN OF
        TRUE: (Z: INTEGER);
        FALSE: (M: PACKED ARRAY[0..3] OF CHAR)
    END
END;
```

In this example, the B and F fields are stored in two bits of the first 16-bit word of the record. The remaining 14 bits are not used. Since the size of the case variant field is always the size of the largest variant, the case variant field in the example occupies two words. Thus, the entire PACKED RECORD occupies three words.

### **2.6.6.3 Restrictions on Packed Variables**

This Pascal does not support the Standard procedures PACK and UNPACK. If a type or variable is declared as packed, the packing and unpacking are automatic.

No element of a PACKED ARRAY or field of a PACKED RECORD can be passed as a variable (call-by-reference) parameter to a routine. Packed variables can, however, be passed as value parameters.

### **2.6.7 Parametric PROCEDURES and FUNCTIONS**

This Pascal does not allow PROCEDURES or FUNCTIONS to be declared as formal parameters in the parameter list of a PROCEDURE or FUNCTION.

### 2.6.8 Program Headings

Although the Pascal Compiler permits a list of file parameters to be present following the program identifier, these parameters are ignored by the Compiler and have no affect on the program being compiled. As a result the following two program headings are equivalent.

```
PROGRAM DEMO( INPUT,OUTPUT );
```

```
PROGRAM DEMO;
```

With either of these program headings, the program has three files predeclared and opened by the System. They are INPUT, OUTPUT, and KEYBOARD; all are type INTERACTIVE. To declare additional files, the file declarations must be declared with the program's other VAR declarations.

### 2.6.9 Sets

Sets are defined as in Standard Pascal. Sets of subranges of integers are limited to the positive integers. The lower bound of a set declaration must be no less than 0. The value of the upper bound of a set declaration must be no greater than 4079, regardless of the value of the lower bound.

Comparisons and operations on sets are allowed only between sets which either have the same base type or are subranges of the same underlying type. For example, in the program below the base type of set S is the subrange type 0..49, while the base type of set R is the subrange type 1..100. The underlying type of both sets is the type INTEGER, which, by the above definition of compatibility, implies that the comparisons and operations on sets S and R in the following program are legal.

```
PROGRAM SETCOMPARE;  
VAR S: SET OF 0..49;  
    R: SET OF 1..100;  
BEGIN  
  S := [0,5,10,15,20,25,30,35,40,45];  
  R := [10,20,30,40,50,60,70,80,90];  
  IF S = R  
    THEN WRITELN('... oops ...')  
    ELSE WRITELN('sets work');  
  S := S + R;  
END.
```

## UCSD PASCAL DIFFERENCES FROM STANDARD PASCAL

In the following example, the construct  $I = J$  is not legal, since the two sets are of two distinct underlying types.

```
PROGRAM ILLEGALSETS;  
TYPE SPELL=(ZERO,ONE,TWO);  
VAR I: SET OF SPELL;  
    J: SET OF 0..2;  
BEGIN  
    I := [ZERO];  
    J := [1,2];  
    IF I = J  
    THEN ... <<<< error here  
END.
```

### 2.6.10 Transcendental Functions

In UCSD Pascal, the arctangent function can be called by either ATAN or ARCTAN.

Since Pascal has a limited set of transcendentals, formulas for the more common transcendental functions are listed in the Appendix.

### 2.6.11 Size Limitations

The following is a list of size limitations that apply to this implementation of Pascal.

- Local variables in a PROCEDURE or FUNCTION can occupy a maximum of 16,383 words of memory.
- The maximum number of characters in a STRING variable is 255.
- The maximum number of words allocated to a SET is 255. Therefore, the maximum number of elements in a set is  $(255*16)=4080$ .
- The maximum number of routines within a segment is 256.
- The maximum integer is 32,767, and the minimum integer is -32,768.

### SECTION 3: PROCEDURES AND FUNCTIONS

This section describes the non-Standard intrinsic procedures and functions included in this version of Pascal. The intrinsics are listed in alphabetical order. Indications of the use of these intrinsics are given in Section 2.

Many of these intrinsics were created to provide access to internal System capabilities. Because they are designed for speed and knowledgeable use, they provide little in the way of parameter checking. Therefore, range and validity checks are your responsibility. The improper use of some of these intrinsics can cause the System to stop functioning, requiring that you turn the computer off for approximately 10 seconds and then back on. Intrinsics which require particular care are noted in their descriptions.

Required parameters are listed with the function/procedure identifier, while optional parameters are listed in brackets. The default values for optional parameters are described in the text.

## PROCEDURES AND FUNCTIONS

### 3.1 ATTACH

ATTACH is a procedure with the form

```
ATTACH (SEM:SEMAPHORE; VECTOR:INTEGER);
```

ATTACH has no effect in this version of Pascal. It is included here only for completeness.

### 3.2 BLOCKREAD

BLOCKREAD is a function with the form

```
BLOCKREAD (FILEID,ARRAY,BLOCKS,[RELBLOCK]): INTEGER;
```

BLOCKREAD reads the number of blocks specified by BLOCKS from FILEID into ARRAY and returns the number of blocks read. If the value returned does not equal BLOCKS, the end of file was encountered or a read error occurred. If the end of file is encountered, EOF is TRUE.

FILEID is an untyped file (see Section 2.2.3.2). ARRAY can be any sort of array or any sort of variable, since BLOCKREAD does no checking. BLOCKS is an integer.

If the optional parameter RELBLOCK is present, it is the number of the block, relative to block zero, from which BLOCKREAD starts reading.

If RELBLOCK is not present, records are read sequentially from the current file location. Immediately after FILEID has been initialized with RESET or REWRITE, BLOCKREAD starts from block zero. Successive BLOCKREADs continue to read sequential records unless RELBLOCK is used or FILEID is re-initialized with RESET or REWRITE.

If the parameter ARRAY contains a subscript (for example, BIG\_TABLE[1024]) and ARRAY is a PACKED ARRAY OF CHAR, BLOCKREAD fills ARRAY starting with that element. If ARRAY is a record that is not packed, it can contain a field specification and filling starts from there.

**Note:** This is a dangerous intrinsic because the bounds of ARRAY are not checked. You are responsible for seeing that no important memory is altered.

## PROCEDURES AND FUNCTIONS

### 3.3 BLOCKWRITE

BLOCKWRITE is a function with the form

BLOCKWRITE (FILEID,ARRAY,BLOCKS,[RELBLOCK]): INTEGER;

BLOCKWRITE writes the number of blocks specified by BLOCKS from ARRAY into FILEID and returns the number of blocks written. If the value returned does not equal BLOCKS, either the end of file was encountered or a write error occurred. If the end of file is encountered, EOF is TRUE. FILEID is an untyped file (see Section 2.2.3.2), and ARRAY can be any sort of array, or anything else, as mentioned in BLOCKREAD. BLOCKS is an integer.

If the optional parameter RELBLOCK is present, it is an integer indicating the block, relative to block zero, at which writing starts.

If RELBLOCK is not present, blocks are written to FILEID sequentially from the current file location. After FILEID is initialized with RESET or REWRITE, BLOCKWRITE starts with block zero. Successive calls to BLOCKWRITE continue writing sequentially unless RELBLOCK is used or FILEID is re-initialized with RESET or REWRITE.

As with BLOCKREAD, a subscript of the parameter ARRAY, with ARRAY a PACKED ARRAY OF CHAR, causes the transfer to start with that element of ARRAY. Also as with BLOCKREAD, a record can have a field specification.

**Note:** This is a dangerous intrinsic because the bounds of ARRAY are not checked. You are responsible for seeing that no important memory is altered.



### 3.4 CHAIN

CHAIN is a procedure with the form

```
CHAIN (EXEC_OPTIONS:STRING);
```

A call to CHAIN causes the System to eX(ecute EXEC\_OPTIONS after the calling program (the "chaining program") has stopped. The effect is the same as typing X for X(ecute and then entering the characters in EXEC\_OPTIONS. Neither the System promptline nor the X(ecute prompt are displayed; the System immediately performs the actions indicated by EXEC\_OPTIONS.

EXEC\_OPTIONS is an execution option string, as defined in the UCSD p-System P-Code manual.

If a program or sequence of programs contains more than one call to CHAIN, the EXEC\_OPTIONS are saved and performed on a first-in, first-out basis before control of the System is returned to you. A call to CHAIN with an empty string--CHAIN("");--clears the queue.

An execution error or an error in an EXEC\_OPTIONS string clears the queue and returns control of the System to you. A call to EXCEPTION can also clear the queue. See the intrinsic EXCEPTION in Section 3.9.

CHAIN is a procedure in the Operating System's COMMANDIO unit. To use it, a program or unit must declare "USES COMMANDIO;".

## PROCEDURES AND FUNCTIONS

### 3.5 CLOSE

CLOSE is a procedure with the form

```
CLOSE (FILEID[,OPTION]);
```

Depending on the OPTION specified and the type of file, CLOSE can close a file, make a file permanent, or delete or truncate a file.

FILEID is the name of an internal file. Typically, it is a diskette file that was opened with a previous RESET or REWRITE and associated with an external file in the System. See Sections 2.2.5 and 2.2.6 on RESET and REWRITE.

OPTION need not be present. If it is present, it can be NORMAL, LOCK, PURGE, or CRUNCH.

If OPTION is not present or is NORMAL, CLOSE closes the file. If the file was opened using REWRITE and is a diskette file, it is deleted from the diskette directory.

If the file associated with FILEID is on a block-structured device such as a diskette and was opened with a REWRITE, the LOCK option makes it permanent in the directory. With any other type of device, a NORMAL close is performed.

If the file associated with FILEID is on a block-structured device, the PURGE option deletes it from the directory. If the file associated with FILEID was a device and not a block-structured volume, the device goes off-line. If no physical file or device was associated with FILEID, a NORMAL close is done.

The CRUNCH option LOCKs the file and truncates it at the point of last access so that the position at the last GET or PUT becomes the end of file.

All CLOSEs, regardless of the option, mark the file as closed and make the implicit variable FILEID undefined. CLOSE on a CLOSED file does nothing.

### 3.6 CONCAT

CONCAT is a function with the form

```
CONCAT (SOURCE_LIST): STRING;
```

This function returns a string which is the concatenation of the strings passed to it.

SOURCE\_LIST is a list of string variables, constants, or literal values. Any number of strings, separated by commas, can appear in SOURCE\_LIST. The CONCAT function is illustrated in the following program segment.

```
SHORTSTRING := 'THIS IS A STRING';  
LONGSTRING := 'THIS IS A VERY LONG STRING.';  
LONGSTRING := CONCAT('START ', SHORTSTRING, '- ', LONGSTRING);  
WRITELN(LONGSTRING);
```

This segment prints the following.

```
START THIS IS A STRING-THIS IS A VERY LONG STRING.
```

## PROCEDURES AND FUNCTIONS

### 3.7 COPY

COPY is a function with the form

```
COPY (SOURCE, INDEX, SIZE): STRING;
```

This function returns a string containing the number of characters (specified by SIZE) copied from SOURCE, starting at the position in SOURCE indicated by INDEX. SOURCE is a string, and INDEX and SIZE are integers.

The following example illustrates the use of the COPY function.

```
TL := 'KEEP SOMETHING HERE';  
KEPT := COPY(TL, POS('S', TL), 9);  
WRITELN(KEPT);
```

This example prints the following.

```
SOMETHING
```

### 3.8 DELETE

DELETE is a procedure with the form

```
DELETE (DESTINATION, INDEX, SIZE);
```

This procedure removes the number of characters specified by SIZE from DESTINATION, starting at the INDEX specified. DESTINATION is a string and INDEX and SIZE are integers.

The following example illustrates the use of the DELETE function.

```
OVERSTUFFED := 'THIS STRING HAS TOO MANY CHARACTERS IN IT.';  
DELETE(OVERSTUFFED, POS('HAS', OVERSTUFFED)+3, 4);  
WRITELN(OVERSTUFFED);
```

This example prints the following.

```
THIS STRING HAS MANY CHARACTERS IN IT.
```

## PROCEDURES AND FUNCTIONS

### 3.9 EXCEPTION

EXCEPTION is a procedure with the form

```
EXCEPTION (STOPCHAINING:BOOLEAN);
```

EXCEPTION turns off all redirection. If STOPCHAINING is TRUE, the queue of EXEC\_OPTIONS created by CHAIN is cleared (see the intrinsic CHAIN, Section 3.4).

When an execution error occurs, an EXCEPTION(TRUE) call is made; leaving redirection on after an error occurs would leave the System in an indeterminate state.

See the UCSD p-System P-Code manual for more information on redirection.

EXCEPTION is a procedure in the Operating System's COMMANDIO unit. Before you can use it, your program or unit must declare "USES COMMANDIO;".

### 3.10 FILLCHAR

FILLCHAR is a procedure with the form

```
FILLCHAR (DESTINATION,LENGTH,CHARACTER);
```

FILLCHAR fills DESTINATION with LENGTH instances of CHARACTER. DESTINATION can be any sort of array, including subscripted arrays. It can also be any other sort of variable, but caution must be exercised. If it is a record, it can have a field specification. LENGTH is an integer. CHARACTER is a single character.

This result of FILLCHAR could also be obtained by the following program segment.

```
A[0] := <character>;  
MOVELEFT(A[0],A[1],LENGTH-1);
```

However, FILLCHAR is twice as fast because no memory reference is needed for a source.

If DESTINATION is subscripted, FILLCHAR begins filling from the subscripted element. The same applies if DESTINATION is a record with a field specification.

**Note:** FILLCHAR is a dangerous intrinsic and does no checking. Use it with caution.

## PROCEDURES AND FUNCTIONS

### **3.11 GOTOXY**

GOTOXY is a procedure with the form

```
GOTOXY (XCOORD, YCOORD: INTEGER);
```

This procedure sends the CONSOLE's cursor to the coordinates specified by XCOORD and YCOORD. The upper left corner of the screen is (0,0).

XCOORD can be from 0 to 79 and YCOORD can be from 0 to 23 in all modes (pattern, text, and multicolor; see Section 6).



### 3.12 HALT

HALT is a procedure with the form

HALT;

This procedure generates a HALT, causing a runtime error to occur. The effect is similar to pressing the <break> key while a program is running. If the error message file (SYSTEM.SYNTAX) is present, the console displays an error message saying that the program has stopped itself as shown below.

Programmed HALT

Segment <segment name> Proc# <number> Offset# <number>

Type <space> to continue

If the error message file is not present, error #14 is given.

## PROCEDURES AND FUNCTIONS

### 3.13 INSERT

INSERT is a procedure with the form

```
INSERT (SOURCE,DESTINATION,INDEX);
```

This procedure inserts the string SOURCE into the string DESTINATION at the position in DESTINATION indicated by INDEX.

The following program segment illustrates the use of INSERT.

```
ID := 'INSERTIONS';  
MORE := ' DEMONSTRATE';  
DELETE(MORE,LENGTH(MORE),1);    { Deletes the final E in  
                                DEMONSTRATE. }  
INSERT(MORE, ID, POS(' IO', ID)); { Inserts " DEMONSTRAT" between  
                                "INSERT" and "IONS". }  
WRITELN(ID);
```

This example prints the following.

```
INSERT DEMONSTRATIONS
```

### 3.14 IORESULT

IORESULT is a function with the form

```
IORESULT: INTEGER;
```

After any I/O operation, IORESULT returns an INTEGER value corresponding to the values listed below.

The Compiler normally generates test code to be performed after each I/O operation to see if the operation has failed. If it has, the Compiler stops the program. Rather than allowing a program to stop, you can turn off I/O checking (see Section 7) and use IORESULT to see if an I/O operation has failed. If it has, your program can take corrective action, such as re-displaying a prompt.

Since any I/O operation, including WRITE and WRITELN, affects IORESULT (unless checking is turned off), WRITELN(IORESULT) is not informative. The following code achieves the desired effect.

```
CHECK_RESULT := IORESULT;  
WRITELN(CHECK_RESULT);
```

I/O checks are not generated for the procedures UNITREAD or UNITWRITE.

## PROCEDURES AND FUNCTIONS

The table of IORESULT values is in the Appendix and below.

- 0 = No error
- 1 = Bad block, parity error (CRC)
- 2 = Illegal device number
- 3 = Illegal I/O request
- 4 = I/O operation cancelled by user (REMIN:, REMOUT:, or PRINTER:)
- 5 = Volume went off-line
- 6 = File lost in directory
- 7 = Bad file name
- 8 = No room on volume
- 9 = Volume not found
- 10 = File not found
- 11 = Duplicate directory entry
- 12 = File already open
- 13 = File not open
- 14 = Bad input information
- 15 = Ring buffer overflow (caused by pressing <etx> when data is expected)
- 16 = Write protect
- 17 = Illegal block
- 18 = Illegal buffer

The IORESULT value is stored in a single System-wide variable. Therefore, concurrent processes (see Section 5) which use IORESULT may not receive a correct value because I/O performed by one process could change the information expected by another. This information change is quite likely with processes that are synchronized by attached semaphores. In other multiprocess situations, switching occurs at explicit SIGNAL and WAIT points, and problems with IORESULT are easily avoided. I/O done by the System itself does not affect a program's IORESULT.

### 3.15 LENGTH

LENGTH is a function with the form

```
LENGTH (SOURCE:STRING): INTEGER;
```

LENGTH returns the integer value of the dynamic length of SOURCE.

The following program segment illustrates the use of LENGTH.

```
ASTRING := '1234567';  
WRITELN(LENGTH(ASTRING), ' ', LENGTH(''));
```

This example prints the following.

```
7 0
```

## PROCEDURES AND FUNCTIONS

### **3.16 MARK**

MARK is a procedure with the form

```
MARK (VAR HEAPPTR: INTEGER);
```

MARK allocates a Heap Mark Record (HMR) on top of the Heap.

HEAPPTR must be a pointer. It is conventional to make it a ^INTEGER. The HMR contains valuable System information, so HEAPPTR must not be used as a pointer to available data space. To allocate memory, use the Standard procedure NEW or the intrinsic VARNEW. MARK is included for compatibility with prior versions of UCSD Pascal.

See the Internal Architecture Guide for more details.

### 3.17 MEMAVAIL

MEMAVAIL is a function with the form

MEMAVAIL: INTEGER;

MEMAVAIL returns the number of unallocated words in memory. This is the number of words between the Code Pool and the Stack plus the number of words available in the Heap.

MEMAVAIL does not return the maximum available memory space since there may be segments in main memory that could be overwritten if necessary. The intrinsic VARAVAIL should be used to determine space availability.

See the Internal Architecture Guide for more details.

## PROCEDURES AND FUNCTIONS

### 3.18 MEMLOCK

MEMLOCK is a procedure with the form

```
MEMLOCK (SEGLIST:STRING);
```

MEMLOCK loads the designated segments and "locks" them into main memory. SEGLIST must contain a list of segment names separated by commas.

See the Internal Architecture Guide for more details.



### 3.19 MEMSWAP

MEMSWAP is a procedure with the form

MEMSWAP (SEGLIST:STRING);

MEMSWAP returns the designated (locked) segments to diskette. SEGLIST must contain a list of segment names separated by commas.

See the Internal Architecture Guide for more details.

## PROCEDURES AND FUNCTIONS

### 3.20 MOVELEFT

MOVELEFT is a procedure with the form

```
MOVELEFT (SOURCE, DESTINATION, LENGTH);
```

MOVELEFT moves the number of bytes specified by LENGTH from SOURCE into DESTINATION, starting at the left. SOURCE and DESTINATION are any sort of array. Or, as with BLOCKREAD, BLOCKWRITE, and FILLCHAR, they can be of any other type as well. If either is an array, it can be subscripted; if either is a record, it can have a field specification. LENGTH is an integer.

MOVELEFT is a fast intrinsic and does no range checking. Exercise care when you use this intrinsic.

The following example shows the use of MOVELEFT.

```
VAR ARRAY: PACKED ARRAY [1..30] OF CHAR;
```

```
    {123456789a123456789b123456789c}
```

```
ARRAY: THIS IS THE TEXT IN THIS ARRAY
```

```
MOVELEFT(ARRAY[10], ARRAY[1], 10);
```

```
ARRAY: HE TEXT INE TEXT IN THIS ARRAY
```

```
MOVELEFT(ARRAY[1], ARRAY[3], 10)
```

```
ARRAY: HEHEHEHEHEHETEXT IN THIS ARRAY
```

```
MOVELEFT(ARRAY[23], ARRAY[2], 8);
```

```
ARRAY: HIS ARRAYEHETEXT IN THIS ARRAY
```

The same effect as MOVELEFT(A[1], A[10], 6); is achieved with

```
FOR I:=0 TO 5
```

```
DO
```

```
    A[10+I] := A[1+I];
```

except that MOVELEFT is much faster.

### 3.21 MOVERIGHT

MOVERIGHT is a procedure with the form

```
MOVERIGHT (SOURCE, DESTINATION, LENGTH);
```

MOVERIGHT moves the number of bytes specified by LENGTH from SOURCE into the DESTINATION, starting at the right. SOURCE and DESTINATION are any sort of arrays. As with MOVELEFT, they can be any other type as well. Either can have a subscript or, if declared as a record, a field specification. LENGTH is an integer.

This procedure is the counterpart to MOVELEFT. **Note:** MOVERIGHT does no range checking. Exercise care when you use this intrinsic.

The following example shows the use of MOVERIGHT.

```
VAR ARRAY: PACKED ARRAY [1..30] OF CHAR;

      {123456789a123456789b123456789c}
ARRAY: THIS IS THE TEXT IN THIS ARRAY

MOVERIGHT(ARRAY[10], ARRAY[1], 9);
ARRAY: HE TEXT IHE TEXT IN THIS ARRAY

MOVERIGHT (ARRAY[3], ARRAY[1], 9);
ARRAY: EHEHEHEHE TEXT IN THIS ARRAY
```

The same effect as MOVERIGHT(A[1], A[10], 6); is achieved with

```
FOR I:=5 DOWNT0 0
DO
  A[10+I] := A[1+I];
```

except that MOVERIGHT is much faster.

## PROCEDURES AND FUNCTIONS

### 3.22 POS

POS is a function with the form

POS (STRING, SOURCE): INTEGER;

POS attempts to match STRING to a substring of SOURCE. If STRING is matched, POS returns the location of the first character of the matched pattern. If STRING is not matched, POS returns zero. STRING and SOURCE are string variables or constants.

The following program segment illustrates the use of POS.

```
      { 123456789a123456789b123456789c12 }  
STUFF := 'TAKE THE BOTTLE WITH A METAL CAP';  
PATTERN := 'TAL';  
WRITELN(POS(PATTERN,STUFF));  
PATTERN := 'CZECHOSLOVAKIA';  
WRITELN(POS(PATTERN,STUFF));
```

This example prints the following.

```
26  
0
```

### 3.23 PWROFTEN

PWROFTEN is a function with the form

```
PWROFTEN (EXPONENT:INTEGER): REAL;
```

This function returns the value of ten to the EXPONENT power.

The legal range of EXPONENT is 0 to 127.

For example, the following program prints +1.00000000000000E+112.

```
VAR NUM_IN:INTEGER;  
    NUM_OUT:REAL;  
BEGIN  
    NUM_IN:=112;  
    NUM_OUT:=PWROFTEN(NUM_IN);  
    WRITELN(NUM_OUT);  
END.
```

## PROCEDURES AND FUNCTIONS

### 3.24 REDIRECT

REDIRECT is a function with the form

```
REDIRECT (EXEC_OPTIONS:STRING): BOOLEAN;
```

REDIRECT causes redirection by performing all the options specified in EXEC\_OPTIONS. If all goes well, the function returns TRUE. If an error occurs, it returns FALSE. EXEC\_OPTIONS is an execution option string as defined in the UCSD p-System P-Code owner's manual. The string should contain only option specifications, not the name of a file to execute. To execute a program from another program, see the CHAIN intrinsic, Section 3.4.

If an error occurs during a call to REDIRECT, the state of redirection is indeterminate, resulting in a dangerous condition. If REDIRECT returns FALSE, your program should follow it with a call to EXCEPTION, in order to turn off all redirection. If you do not, the results are unpredictable. See the intrinsic EXCEPTION, Section 3.9.

REDIRECT is a procedure in the Operating System's COMMANDIO unit. Before you can use it, your program or unit must contain the declaration "USES COMMANDIO;".

More information about redirection can be found in the P-Code manual.

### 3.25 RELEASE

RELEASE is a procedure with the form

```
RELEASE (VAR HEAPPTR: INTEGER);
```

The procedure RELEASE cuts back the Heap from the current Heap Mark Record (HMR) to the HMR designated by HEAPPTR. HEAPPTR must have been initialized by the MARK procedure. RELEASE is included for compatibility with prior versions of UCSD Pascal.

MARKs and RELEASEs must be matched properly. For additional information, see the important statement at the end of the discussion of memory allocation (Section 2.3.1).

See the Internal Architecture Guide for more details.

## PROCEDURES AND FUNCTIONS

### 3.26 SCAN

SCAN is a function with the form

SCAN (LENGTH,PARTIAL\_EXPRESSION,ARRAY): INTEGER;

SCAN scans ARRAY for the number of characters indicated by LENGTH or until it finds a character that satisfies the PARTIAL\_EXPRESSION. The function returns the offset from the starting position in ARRAY to the point at which it stopped. LENGTH is an integer, ARRAY is usually a PACKED ARRAY OF CHAR, and PARTIAL\_EXPRESSION is a "<>" or an "=" followed by a single character in quotes or a character of type CHAR.

If the position in ARRAY at which SCAN starts satisfies the PARTIAL\_EXPRESSION, SCAN returns zero. If the PARTIAL\_EXPRESSION is not satisfied, SCAN returns LENGTH. If the PARTIAL\_EXPRESSION is satisfied at some intermediate location, SCAN returns the offset from the starting position to that location.

If LENGTH is negative, the SCAN is from right to left and returns a negative value.

ARRAY can be subscripted. If so, SCAN starts scanning at that location. ARRAY can in fact be of any type, but you should exercise caution to be sure that the index returned is valid.

The following program segment illustrates the use of SCAN.

```
VAR DEM: PACKED ARRAY[0..52] OF CHAR;
BEGIN
    {123456789a123456789b123456789c123456789d123456789e1}
    DEM := '.....THE TERAk IS A MEMBER OF THE PTERODACTYL
           FAMILY.';
    WRITELN(SCAN(-26,=':',DEM[30]));
    WRITELN(SCAN(100,<>' ',DEM));
    WRITELN(SCAN(15,=' ',DEM[0]));
END.
```

This example prints the following.

-26

5

8



### 3.27 SEEK

SEEK is a procedure with the form

```
SEEK (FILEID, INDEX);
```

SEEK changes the file window variable F to point to the record in FILEID specified by INDEX. The first record in FILEID is zero. EOF and EOLN are set to FALSE. FILEID is a file of any structured type; i.e., it is not a text file (TEXT, INTERACTIVE, or FILE OF CHAR) or an untyped file. INDEX is an integer.

A GET or PUT should immediately follow a SEEK. Otherwise, the window contents are unpredictable.

## PROCEDURES AND FUNCTIONS

### 3.28 SEMINIT

SEMINIT is a procedure with the form

```
SEMINIT (VAR SEM:SEMAPHORE; SEM_COUNT:INTEGER);
```

SEMINIT initializes the semaphore SEM to the value SEM\_COUNT and establishes an empty queue. See Section 5.

**Note:** Failure to initialize a semaphore before using it in a SIGNAL or WAIT puts the System in an indeterminate state.

### 3.29 SIGNAL

SIGNAL is a procedure with the form

```
SIGNAL (VAR SEM: SEMAPHORE);
```

If no processes are waiting for the semaphore SEM, SIGNAL increments the count associated with SEM. If one or more processes are waiting for SEM, SEM is not incremented and the process at the head of SEM's queue (the process with the highest priority) is added to the ready queue, where it competes with other ready processes for processor time. See Section 5.

## PROCEDURES AND FUNCTIONS

### **3.30   SIZEOF**

SIZEOF is a function with the form

SIZEOF (VARIABLE\_OR\_TYPE\_IDENTIFIER): INTEGER;

SIZEOF returns the number of bytes allocated to the variable or type. This function is often useful as a parameter to FILLCHAR, MOVELEFT, or MOVERIGHT.

### 3.31 START

START is a procedure with the form

```
START (PROCESS_CALL[ ,PROCESSID[ ,STACKSIZE[ ,PRIORITY]]]);
```

START initiates a process. The PROCESS\_CALL parameter identifies the PROCESS to be started and may optionally pass parameters to the PROCESS.

PROCESSID is a variable of type PROCESSID, STACKSIZE is an integer, and PRIORITY is in the range [0..255]. These three parameters are optional.

Every process invocation (i.e., every call to START) is assigned a PROCESSID. This parameter, if present, is set to the PROCESSID value. PROCESSIDs are intended for the System's use.

STACKSIZE, if present, allocates stack space to the process. STACKSIZE defaults to 200 words. A process needs four words plus the number of words occupied by local variables plus room for the activation records of procedures started by the process plus space for the evaluation stack. If a process is allocated less memory than it needs, the program ends with a stack overflow. To determine the best value for STACKSIZE (i.e., the one that uses the least memory), start with a large value such as 3000. If the program is too large, use a smaller value. Gradually reduce the value of STACKSIZE until the program no longer runs. The minimum value at which the program runs is the best value for STACKSIZE.

PRIORITY, if present, specifies the priority of the process. Priorities determine the ordering of a queue waiting for a semaphore, and the ordering of the queue of all processes that are ready to run. The highest (most urgent) priority is 255. PRIORITY defaults to the priority of the STARTing process. If no PRIORITY parameter appears, the process's priority is the same as the priority of the process that calls START. The default priority is 127.

The following examples show the use of START.

```
start(PLOP);  
start(RED(I,J), PID);  
start(SHAWNEE(10), ID, 500);  
start(RED(6,14), PID, SSIZE, 46);
```

See Section 5 for more information on concurrent processes.

## PROCEDURES AND FUNCTIONS

### 3.32 STR

STR is a procedure with the form

```
STR (LONG,DESTINATION);
```

STR converts LONG into a string and places it in DESTINATION. This intrinsic is chiefly used to format long integers for output. LONG is either an integer or a long integer. DESTINATION is a string.

See Section 2.6.5 for more about long integers.

The following program segment shows the use of STR.

```
INTLONG := 102039503;  
STR( INTLONG, INTSTRING);  
INSERT( '.', INTSTRING, PRED(LENGTH( INTSTRING)));  
WRITELN( '$', INTSTRING);
```

This example prints the following.

```
$1020395.03
```

### 3.33 TIME

TIME is a procedure with the form

```
TIME (VAR HIWORD,LOWORD:INTEGER);
```

The TIME procedure returns the value of the System's clock in 60ths of a second. The value is stored in HIWORD, LOWORD as one 32-bit unsigned integer. No conventions exist to allow you to treat the value returned by TIME as the time of day. TIME is usually used for incremental time measurements, such as calculating benchmarks for a program.

## PROCEDURES AND FUNCTIONS

### 3.34 UNITBUSY

UNITBUSY is a function with the form

UNITBUSY (UNITNUMBER): BOOLEAN;

UNITBUSY always returns a value of false. It is included here only for completeness.



### 3.35 UNITCLEAR

UNITCLEAR is a procedure with the form

UNITCLEAR (UNITNUMBER);

UNITCLEAR cancels all I/Os to the specified unit and resets the hardware to its power-up state. UNITNUMBER is an integer that is the number of a device (see the Appendix, Section 8.3).

The function IORESULT can determine if an error occurred (see Section 3.14). IORESULTs are listed in Section 3.14 and the Appendix, Section 8.2.

## PROCEDURES AND FUNCTIONS

### 3.36 UNITREAD

UNITREAD is a procedure with the form

```
UNITREAD (UNITNUMBER, ARRAY, LENGTH[, BLOCKNUMBER][, [INTEGER]]);
```

UNITREAD reads the number of bytes specified by LENGTH from the device UNITNUMBER into ARRAY. ARRAY can be of any type, but is usually a PACKED ARRAY OF CHAR. UNITNUMBER is an integer that is the number of a device (see the Appendix). UNITREAD is a low-level intrinsic and should be used with extreme caution. It performs no I/O checking of any sort and receives all characters sent by the device, including protocols, blank-compressions, and the like.

ARRAY can be subscripted, in which case it is filled starting from that element.

BLOCKNUMBER is only meaningful if UNITNUMBER is a block-structured device. Then it is the number of the block (zero-based) from which the read starts. If BLOCKNUMBER is not given, the default is zero.

If INTEGER is equal to 16,384, the destination is VDP memory rather than CPU memory. ARRAY is then a pointer, with the value INTEGER<sup>^</sup> being the VDP address that is to be written to.

For example, the following program reads the first two blocks from unit #4 into VDP memory starting at address >0000 (the start of the Pattern Descriptor Table) and defines characters in the character set.

```
PROCEDURE VDP_WRITE;  
  TYPE MEM =RECORD CASE BOOLEAN OF  
    TRUE: (INT:INTEGER);  
    FALSE: (PTR: ^INTEGER);  
  END;  
  VAR VDP:MEM;  
  BEGIN  
    VDP.INT:=0; {Starting VDP address.}  
    UNITREAD(4,VDP.PTR,1024,0,16384);  
  END;
```

The following example illustrates the use of UNITREAD to read from a non-block-structured device.

```
UNITREAD(7,FILLME,80,,1)
```

This example reads 80 characters from REMIN: into the array FILLME. FILLME must be at least 80 characters long or other data is destroyed.

**Note:** Because it refers directly to a device, input from UNITREAD cannot be redirected.

## PROCEDURES AND FUNCTIONS

### 3.37 UNITSTATUS

UNITSTATUS is a procedure with the form

```
UNITSTATUS (UNITNUMBER, STATUS_REC, CONTROL);
```

UNITSTATUS returns information in STATUS\_REC. If CONTROL is zero, the information refers to output. If CONTROL is one, the information refers to input.

UNITNUMBER is an integer that is the number of a device (see the Appendix, Section 8.3). STATUS\_REC can be of any type; it should be an area of 30 words. CONTROL is an integer equal to either 0 or 1.

On a character-oriented device, such as PRINTER:, REMOUT:, or CONSOLE:, UNITSTATUS changes only the first word of STATUS\_REC and sets it equal to the number of characters waiting to be read or written. If no characters are waiting or UNITSTATUS cannot determine the device's state, it returns a zero.

If the device is a block-structured device (such as a diskette), UNITSTATUS changes the first four words of STATUS\_REC as follows.

Word one:      The number of characters waiting (as with a serial device).

Word two:      The number of bytes per sector on the device.

Word three:    The number of sectors per track.

Word four:     The number of tracks.

Although the remainder of STATUS\_REC is not affected, these locations are reserved for possible future use.

### 3.38 UNITWAIT

UNITWAIT is a procedure with the form

UNITWAIT (UNITNUMBER);

UNITWAIT always returns immediately. It is included here only for completeness.

## PROCEDURES AND FUNCTIONS

### 3.39 UNITWRITE

UNITWRITE is a procedure with the form

```
UNITWRITE (UNITNUMBER, ARRAY, LENGTH[, BLOCKNUMBER][, INTEGER]);
```

UNITWRITE writes the number of characters specified by LENGTH from ARRAY to the device UNITNUMBER. UNITNUMBER is an integer that is the number of a device (see the Appendix, Section 8.3).

ARRAY can have a subscript, in which case the transfer begins with that element.

BLOCKNUMBER applies only to block-structured devices and, if present, indicates the number of the block (zero-based) where the write starts. BLOCKNUMBER defaults to zero.

INTEGER, if present, may have the same values as in UNITREAD (see Section 3.36).

Because it refers directly to a device, output from UNITWRITE cannot be redirected.

**Note:** As with UNITREAD, no I/O checking is done, nor are any of the transmission's characters added or modified. UNITWRITE is a low-level intrinsic. Therefore, it is fast but dangerous.

### 3.40 VARAVAIL

VARAVAIL is a function with the form

VARAVAIL (SEGLIST): INTEGER;

VARAVAIL returns the number of words in main memory available for allocation, after subtracting the words used if the listed segments and all memory-locked segments are in memory. The value returned is not necessarily the current amount of memory available. SEGLIST is a string containing a list of segment names separated by commas. If a segment name is not recognized by the System, it is ignored.

VARAVAIL may not be meaningful if a PROCESS is running concurrently. See START (Section 3.31) for more information.

See the Internal Architecture Guide for more details.

## PROCEDURES AND FUNCTIONS

### 3.41 VARDISPOSE

VARDISPOSE is a procedure with the form

```
VARDISPOSE (POINTER,COUNT);
```

VARDISPOSE deallocates the number of words specified by COUNT. If COUNT is incorrect, VARDISPOSE destroys the Heap's integrity, so use extreme caution. POINTER is an arbitrary pointer type which must have been initialized by a call to VARNEW (Section 3.42). COUNT should have the same value as obtained with VARNEW.

See the Internal Architecture Guide for more details.



### 3.42 VARNEW

VARNEW is a function with the form

VARNEW (POINTER,COUNT): INTEGER;

VARNEW allocates the number of words specified by COUNT. POINTER is an arbitrary type.

Count is an INTEGER. If COUNT words are available, VARNEW returns COUNT. If COUNT words are not available, VARNEW returns a zero, and no words are allocated. You should maintain POINTER and COUNT and use them with VARDISPOSE (Section 3.41) to return the memory to System use.

See the Internal Architecture Guide for more details.

## PROCEDURES AND FUNCTIONS

### 3.43 WAIT

WAIT is a procedure with the form

```
WAIT (VAR SEM: SEMAPHORE);
```

The WAIT procedure is used in concurrent processing. If SEM is greater than zero, it is decremented and the process that called WAIT continues. If the count of SEM is zero, the process waits until SEM is again available.

See Section 5 for examples.

## **SECTION 4: SEGMENTS AND LINKING**

Segments and linking are two major facilities which can help management of program files and main memory. These facilities permit the development of very large programs in a microsystem environment and, in fact, have been used extensively in the development of the System itself.

The techniques offered by the System fall broadly into two categories: run time main memory management and separate compilation. This section discusses both categories.

## SEGMENTS AND LINKING

### 4.1 MAIN MEMORY MANAGEMENT

Not all of a program needs to be in main memory at run time; usually just one portion of code is required over a given period of time. For most (if not all) of a program's execution time, the code is a subset of the program. Portions of a program which are not currently needed can reside on diskette, freeing main memory for other uses.

When the System executes a code file, it reads code into main memory and runs it. When the code has finished running or the space it occupies is needed for some action of higher priority, the space it occupies can be overwritten with new code or new data. Code is moved into memory one segment at a time.

In its simplest form, a code segment includes a main program and all of its routines. A routine can occupy a segment of its own if it is a SEGMENT routine. SEGMENT routines can be swapped independently of the main program, so declaring a routine to be a SEGMENT is an efficient means of managing memory.

Routines which are not part of a program's main code are prime candidates for occupying their own segments. Such routines include initialization and wrap-up procedures and routines that are used once or rarely while a program is executing.

Reading a procedure from diskette into main memory before it is executed takes time, so carefully select which procedures to make diskette-resident.

The TI Home Computer has two separate areas for code. The main code pool is approximately 12K bytes and can only contain p-code. The alternate code pool is approximately 20K bytes and can hold data, assembly language code, and p-code. The main code pool is used to hold p-code until it is filled. Remaining p-code is then placed in the alternate code pool. However, any segment that contains assembly language code is put in the alternate code pool.

If you are not using assembly language, you can manage memory best by keeping each code segment less than 12K bytes so that the alternate code pool area is used only for data. If you are using assembly language, it is usually best to put all assembly language units into one segment, which is then loaded into the alternate code pool.

## 4.2 SEPARATE COMPILATION

Separate compilation, also referred to as "external compilation," is a technique in which portions of a program are compiled separately and are subsequently executed as a coordinated whole.

Many programs are too large to compile within the memory confines of a microcomputer. Compiling pieces of a program separately overcomes this memory problem. (The Operating System was compiled in this way.)

Separate compilation also has the advantage of allowing small portions of a program to be changed without affecting the rest of the code. This saves much time and is less error prone. Libraries of correct routines can be built up and used in the development of other programs.

These considerations also apply to assembly language programs. Large assembly language programs can often be more effectively maintained in several separate pieces. When all these pieces have been assembled, a "link editor" (the System's Linker) combines them by installing the linkages that allow the various pieces to refer to each other and function as a unified whole.

It may also be desirable for a higher-level language program to refer to an assembly language routine for performance reasons or to provide low-level machine- or device-dependent handling. The System allows assembly language routines to be linked with other assembly routines or into higher-level hosts (programs or units). Refer to the UCSD p-System Assembler and Linker manuals.

In this version of Pascal, separate compilation is achieved by the UNIT construct. A UNIT is a group of routines and data structures. The contents of a UNIT usually relate to some common application, such as screen control or data file handling. A program or another UNIT (called a "client module" or "host") can use the routines and data structures of a UNIT by simply naming it in a "USES" declaration.

The code for a UNIT that is used by a program may reside in \*SYSTEM.LIBRARY or in another code file. If it is in another code file, you can inform the Compiler of this with the \$U compile-time option (see Section 7), and inform the Operating System by including the code-file's name in a "library text file." The default library text file is \*USERLIB.TEXT, but it can be changed by an execution option. See Section 4.5.

## SEGMENTS AND LINKING

### 4.3 PROGRAMMING TACTICS

This section offers some advice on the use of SEGMENTS and UNITS. It presents a plan for the design of a large program, with some strategies that might be employed. UNITS and SEGMENTS are useful means of decomposing large programs into sections that perform independent tasks.

On microprocessor systems, the main bottlenecks in the development of large programs are (1) the large number of variable declarations that consume space while a program is compiling and (2) the large pieces of code taking up memory space while the program is executing. UNITS address the first problem by allowing separate compilation and minimizing the number of variables that are needed to communicate between separate tasks. SEGMENTS address the second problem by keeping unused code on diskette and only allowing code that is in use to be present in main memory.

A program can be written with run time memory management and separate compilations already planned, or it can be written as a whole and then tuned to fit a particular system. The latter approach is feasible when you are unsure about the necessity of using SEGMENTS or are quite sure that they will be used only rarely. The former approach is preferred and is usually easier to accomplish.

A typical plan for the construction of a relatively large program is shown below.

1. Design the program (user and machine interfaces).
2. Determine needed additions to the library of utilities, including both general and applied tools.
3. Write and debug utilities and add them to libraries.
4. Write and debug the program.
5. Modify the program for better performance.

During the design, you should try as much as possible to use existing procedures in order to decrease coding time and increase reliability. This strategy can be assisted by UNITS. To determine segmentation, consider the expected execution sequence and attempt to group routines inside SEGMENTS so that SEGMENT routines are called as infrequently as possible.

It is important that SEGMENT routines be independent and not call routines in different segments, including non-SEGMENT routines. If they do, both segments must be in memory at the same time, eliminating the advantage of segmentation.

While designing the program, also consider the logical (functional) grouping of procedures into UNITS. Beside making the compilation of a large program possible, this grouping can aid the program's conceptual design and therefore its testing. UNITS can contain SEGMENT routines, so the two techniques can be combined.

Note that a UNIT occupies a segment of its own except possibly for any SEGMENT routines it may contain. The UNIT's segment, like other code segments, remains resident on diskette except when its routines are being called.

Steps 2 and 3 mentioned earlier let you save the new routines in a form which allows them to be used in future programs. At this point the design should be reviewed (and perhaps modified) in order to identify routines which might be useful in the future. You may now want to make routines somewhat more general before putting them into libraries.

It is usually good practice to program and test these utilities before programming the remainder of the program. Doing so helps to ensure that the procedures added to the library have greater potential usefulness, since it helps you to avoid the tendency to tailor them to the particular program being developed.

The INTERFACE part of a UNIT should be completed before the IMPLEMENTATION part, especially if several programmers are working on the same project.

Tuning a program usually means performance tuning. Since SEGMENTS offer greater memory space at reduced speed, performance may be improved by turning routines into SEGMENT routines or by turning SEGMENT routines back into normal routines.

## SEGMENTS AND LINKING

### 4.4 SEGMENTS

The declaration of a segment routine is no different from that of other routine declarations (procedures, functions, and processes), except that it is preceded by the reserved word `SEGMENT`.

The following is an example of a segment.

```
SEGMENT PROCEDURE INITIALIZE;  
BEGIN  
  { Pascal code here. }  
END;
```

Declaring a routine as a segment routine does not change the meaning of the Pascal program, but does affect the time and space requirements of the program's execution. The segment routine and all of its nested routines (except a nested routine that is itself a segment routine) are grouped together in a code segment.

A program and its routines are all compiled as a single code segment except for routines declared as `SEGMENTS`. Since a code segment is diskette resident until it is used and since the space it occupies in memory may be overwritten when it stops, declaring once- or little-used routines as `SEGMENTS` may improve a program's use of main memory.

Up to 255 segments can be contained within a program. The "bodies" (that is, the `BEGIN-END` blocks) of all segment routines must be declared before the bodies of all non-segment routines within a given code segment. This applies to both segment routines and main programs. If a segment routine calls a non-segment routine, the non-segment routine must be forward-declared because its body cannot precede the body of any segment routine (including its caller).

Any routine can be declared a `SEGMENT`, with the following restrictions.

- `SEGMENT` routines must be declared in the `IMPLEMENTATION` section.
- An `EXTERNAL` routine cannot be a `SEGMENT` routine.



The following program segment illustrates the use of segments.

```
PROGRAM GOLE;  
SEGMENT PROCEDURE STRENGAL;  
  ...  
  BEGIN  
    ...  
  END;  
PROCEDURE MYNDAL (FLAK: INTEGER); FORWARD; {MYNDAL is not a  
                                             SEGMENT routine,  
                                             and therefore must  
                                             be declared  
                                             FORWARD.}  
SEGMENT FUNCTION MOAD (PART,WHOLE: REAL): INTEGER;  
  ...  
  BEGIN  
    ...  
  END;  
PROCEDURE MYNDAL;  
  ...  
  PROCEDURE EARLY (I: UNREAL);  
    ...  
    SEGMENT PROCEDURE LATE (J: IMAGINARY);  
      ...  
      BEGIN { LATE CODE }  
        { Note that this can be a segment because it precedes  
          all code bodies within the enclosing code segment  
          (i.e., GOLE). }  
      END { LATE };  
      BEGIN { EARLY CODE }  
        ...  
        END { EARLY };  
      BEGIN { MYNDAL CODE }  
        ...  
        END { MYNDAL };  
      BEGIN  
        ...  
      END { GOLE }.
```

## SEGMENTS AND LINKING

### 4.5 UNITS

A UNIT is a group of interdependent procedures, functions, processes, and associated data structures which are usually related to a common area of application. Whenever a UNIT is needed within a program, the program declares it in a USES statement. A UNIT consists of two main parts: an INTERFACE part which declares constants, types, variables, procedures, functions, and processes that are public and can be used by the host program or other UNIT, and an IMPLEMENTATION part which declares labels, constants, types, variables, procedures, functions, and processes that are private, i.e., not available to the host and used only within the UNIT. The INTERFACE part declares how the program communicates with the user of the UNIT, while the IMPLEMENTATION part defines how the UNIT accomplishes its task.

The Texas Instruments UNITS for the TI Home Computer are SUPPORT, RANDOM, MISC, SOUND, BEEP, SPRITE, and SPEECH. Their use is described in Section 6.

The syntax of a UNIT can be outlined as follows.

```
UNIT <unit identifier>;
INTERFACE
  USES <unit identifier list>;
  <constant definitions>;
  <type definitions>;
  <variable declarations>;
  <routine headings>;
IMPLEMENTATION
  USES <unit identifier list>;
  <label declarations>;
  <constant definitions>;
  <type definitions>;
  <variable declarations>;
  <routine declarations>;
[ BEGIN
  <initialization statements>
  ***;
  <termination statements> ]
END.
```

The INTERFACE part can only contain routine headings (no bodies). The bodies of routines declared in the INTERFACE part are defined in the IMPLEMENTATION part, much as FORWARD procedures are defined apart from their original declaration.

An INTERFACE part is terminated by the reserved word IMPLEMENTATION. An INTERFACE part cannot contain \$Include files (see Section 7). However, an INTERFACE part can be contained within a \$Include file, provided that all of the INTERFACE is in the \$Include file; i.e., an INTERFACE part cannot cross a \$Include file boundary. IMPLEMENTATION terminates an INTERFACE part, so if an INTERFACE part is contained in a \$Include file, the \$Include file must contain both the reserved words INTERFACE and IMPLEMENTATION.

The following are not legal forms of a UNIT.

UNIT GOLE1;	UNIT GOLE2;
INTERFACE	{\$I INTER_PART}
{\$I INTER_DECS}	IMPLEMENTATION
IMPLEMENTATION	...
...	END;
END;	

The following outline is a legal form of a UNIT.

```
UNIT GOLE3;  
  {$I WHOLE_UNIT}  
;
```

The initialization statements and termination statements are optional sections of code. Initialization statements, if present, are performed before any of the code in a host that USES the UNIT is executed; and termination statements, if present, are performed after the host's code has terminated.

Initialization statements are separated from termination statements by the line "\*\*\*;". The section of initialization statements, the section of termination statements, or both, can be empty.

## SEGMENTS AND LINKING

The following are all legal code bodies of a UNIT:

```
END { There is no initialization or termination code. };
```

```
BEGIN
```

```
    { This is initialization code. }
```

```
    INIT_ARRAYS;
```

```
    FLAG := FALSE;
```

```
    COUNT := 23;
```

```
    ***;
```

```
    { This is termination code. }
```

```
    SEMINIT ( LIGHT, 0 );
```

```
END { UNIT };
```

```
BEGIN
```

```
    ***;
```

```
    { This is all termination code. }
```

```
    INIT_ARRAYS;
```

```
    FLAG := FALSE;
```

```
    COUNT := 23;
```

```
    SEMINIT ( LIGHT, 0 )
```

```
END { UNIT };
```

```
BEGIN
```

```
    { This is all initialization code. }
```

```
    INIT_ARRAYS;
```

```
    FLAG := FALSE;
```

```
    COUNT := 23;
```

```
    SEMINIT ( LIGHT, 0 )
```

```
END { UNIT };
```

The statement part of a UNIT should not contain GOTO statements which branch around the "\*\*\*;" separator. The effect of executing such statements is not fully predictable.

A UNIT's statement part can contain statements of the form EXIT(PROGRAM) but EXIT(<unitname>) is not allowed. An EXIT(PROGRAM) in the initialization code has the effect of skipping the remainder of the initialization code (if any) and the host's code; execution then proceeds with the UNIT's termination section. An EXIT(PROGRAM) in the termination code skips the remainder of the termination code. There may be termination code from other hosts still waiting to execute; the EXIT does not stop the execution of these other termination sections.

To use one or more UNITs, a program must name them in a USES declaration immediately following the program heading. Upon encountering a USES declaration, the Compiler refers to the INTERFACE part of the UNIT as though it were part of the host text itself. Therefore, all identifiers declared in the INTERFACE part are global. Name conflicts may arise if the host defines an identifier already defined in the UNIT.

A UNIT can refer to (USE) another UNIT. Then the USES declaration may appear at the beginning of either the INTERFACE part or the IMPLEMENTATION part. Since references to a UNIT can be nested, if they appear in the INTERFACE part, the ordering of the reference is important. For example, if UNITA refers to UNITB, the declaration USES UNITB must appear before the declaration USES UNITA.

The three programs on the next page illustrate the use of UNITs, assuming \*USERLIB.TEXT contains A and B.

## SEGMENTS AND LINKING

```
PROGRAM HOST;                                { Host Program File. }
USES {$U B.CODE} UNITB,
    {$U A.CODE} UNITA;
BEGIN
    PROCA;
END.

UNIT UNITA;                                { UNITA Program File. }
INTERFACE
    USES {$U B.CODE} UNITB;
    PROCEDURE PROCA;
IMPLEMENTATION
    PROCEDURE PROCA;
    BEGIN
        WRITELN('PROC A');
        WRITE('CALLING PROC B - ');
        PROCB;
    END;
END.

UNIT UNITB;                                { UNITB Program File. }
INTERFACE
    PROCEDURE PROCB;
IMPLEMENTATION
    PROCEDURE PROCB;
    BEGIN
        WRITELN('PROC B');
    END;
BEGIN
    ***;
    WRITELN('TERMINATION CODE');
END.
```

Routines declared in the INTERFACE part must not be SEGMENT routines, but SEGMENT routines can be declared in the IMPLEMENTATION part. Declaring SEGMENTS within UNITS is subject to the same ordering as within a main program; see Section 4.2.

For purposes of listing a program, the Compiler treats an INTERFACE section as an include level. Thus, \$Include file nesting is restricted within the scope of a USES declaration.

The System compiles a Pascal program, a single UNIT, or a string of UNITs separated by semicolons. Your program can define a UNIT in-line, but an in-line UNIT definition must appear between the program heading and the <block>. If a UNIT and program are in the same source file and you make changes to either the program or the UNIT, then the source file must be recompiled. If the program and UNIT are in different files and you change the INTERFACE part of the UNIT, then both files must be recompiled.

UNITs need not be explicitly linked together. At compile time a USED UNIT's INTERFACE part must be referenced by the Compiler. If the UNIT's source is in the host program's source, or if the UNIT's code is in \*SYSTEM.LIBRARY, nothing more needs to be specified. If the UNIT's code resides in a different file (a "user library"), the \$U Compiler directive must be used to specify which file (see Section 7).

At run time, the code (all code, in fact) must be in either the user program, \*SYSTEM.LIBRARY, a user library, or the Operating System. If a unit is in a user library, the name of the library file must appear in a "library text file." To find a UNIT's code, the System searches first the files named in a library text file, in order, and then searches \*SYSTEM.LIBRARY. If no library text file is present, the System only searches \*SYSTEM.LIBRARY. The default library text file is called \*USERLIB.TEXT. This default can be changed by an execution option (see Section 7).

The following might be the contents of a library text file.

```
FUN:ADVENT.LIB
  curve
  tg: graphics
  PLAY
```

For each UNIT encountered in the host, the System searches first ADVENT.LIB (which must reside on the volume FUN:), then CURVE.CODE (which must reside on the default volume), and so forth. Failing to find a UNIT in these four files, the System searches \*SYSTEM.LIBRARY.

## SEGMENTS AND LINKING

As indicated in the example, specifying the .CODE suffix to a file name is optional in the library text file's list.

The name \*SYSTEM.LIBRARY can be included in a library text file. If this is the case, it is searched in order, as it appears.

Changes in a host program require that you recompile the program. Changes in the IMPLEMENTATION part of a UNIT require you to recompile the UNIT. Changes in the INTERFACE part of a UNIT require that you recompile both the UNIT and all hosts that USE that UNIT.

External linkages involving assembled routines are discussed in the UCSD p-System Linker manual and in Section 4.6.



#### 4.6 THE LINKER

The Linker is a System program (accessed by the L(ink command at the System level) which allows EXTERNAL code to be linked with a Pascal program. EXTERNAL routines are procedures, functions, or processes that are written in TMS9900 assembly language and conform to the System's calling and parameter-passing protocols. They are declared EXTERNAL in the host program and must be linked before the program is run. The Linker can also be used to link together separately assembled pieces of a single assembly program. See the UCSD p-System Linker manual.

## SEGMENTS AND LINKING

### 4.7 THE UTILITY LIBRARY

LIBRARY.CODE is a utility program that allows you to group separate compilations (UNITs or programs) and separately assembled routines into a single file. It is discussed in the UCSD p-System Utilities manual.

## SECTION 5: CONCURRENT PROCESSES

This version of Pascal allows you to declare and initiate concurrent processes. A concurrent process is a procedure whose execution appears to proceed at the same time as the main program. Processes are declared like procedures and are set into action by the intrinsic START. More than one process can run at once, and the same process can be STARTed several times.

On the TI Home Computer, the System shares the processor among various Pascal processes. This switching may lead to an overall increase in program execution time. Processes are nonetheless useful in a variety of applications.

This implementation of UCSD Pascal does not permit interrupts to cause processes to be initialized. All events which cause the start or termination of a process must be caused by the program.

## CONCURRENT PROCESSES

### 5.1 PROCESSES

A process is declared exactly as a procedure is, with the reserved word `PROCESS` replacing the reserved word `PROCEDURE`.

The following program segments illustrate the use of `PROCESSes`.

```
PROCESS ZIP;
  BEGIN
    ...
  END;

PROCESS DINNER (var SPLIT, BLACKKEYED: peas);
  begin
    ...
  end;
```

A process is started by the intrinsic `START`. The principal parameter passed to `START` is a call to a process, for example, `START(ZIP)` or `START(DINNER(7,234))`.

In the following example, program `DUFFER` starts process `RED` four times and process `BLUE` once. Each of the five processes runs to completion, as does the main program, and the processor shares time among them. Note that the four invocations of `RED` result in four different versions of `RED` being started, each with different parameter values.

```
PROGRAM DUFFER;
  var PID: processid;
      I, J: integer;
  PROCESS BLUE;
    begin
      ...
    end;
  PROCESS RED (X, Y: integer);
    begin
      ...
    end;
  begin
    start(BLUE);
    I := 200;
    J := 300;
```

```
    start(RED(I, J));  
    start(RED(3, 4), PID);  
    start(RED(5, 5), PID, 300);  
    start(RED(J, 1), PID, I+J, 10);  
    ...  
end.
```

In addition to the principal parameter, START may have three optional parameters. Each invocation of a process is assigned an internal PROCESSID, which is a predeclared type. You can learn what PROCESSID has been assigned a given process invocation by using the second parameter. Thus, in START(RED(3,4), PID); the variable PID is set to a new PROCESSID value. PROCESSIDs are chiefly for the use of the System and system programmers.

The third parameter to START, if present, can be the stacksize parameter. It determines how much memory space is allocated to the process invocation. The default is 200 words.

The fourth parameter to START, if present, can be a priority value. This determines the proportion of processor time that the process receives before it is completed. The priorities assigned to processes are used by the System to decide which active process gets to use the available processor. Higher priority processes are given the processor more often than lower priority processes. If no priority value is given in START, the new process inherits the priority value of its caller. Priorities range from 0 through 255, with 255 being the highest (most urgent) priority. The default priority is 127.

See START in Section 3.31 for more details.

### 5.2 SEMAPHORES

The name "semaphore" was coined by E.W. Dijkstra as an analogy to a railroad traffic signal. The railroad semaphore controls whether or not a train can enter the next section of track. A train passing the semaphore when it is green automatically switches it to red, preventing further trains from entering that section of track until the first train has left, at which time the semaphore is switched to green again.

Semaphores can be used for mutual exclusion problems, i.e, controlling access to "critical sections" of code and synchronizing "cooperating processes." A common application employing both of these capabilities is resource allocation, discussed below.

Semaphores can be divided into two classes: Boolean and counting semaphores. A semaphore which has only two states (for example, stop and go) is a Boolean semaphore. If more than two states are allowed, a semaphore is a counting semaphore. In this version of Pascal, counting semaphores can span the range 0 through 32,767. The zero is analogous to the stop value. It is possible to use counting semaphores as Boolean semaphores if they are restricted to the values 0 and 1.

Given a set of concurrent processes and a single semaphore variable which they test, we can imagine that each process (or "train") is running on a private processor ("track") with separate indicators of the semaphore value under some central control. For example, there might be a section of track which must be shared by all the trains, but only a single train is to be allowed in that section at a time. When the value of the semaphore is zero, the central control causes any trains that approach the semaphore to stop and wait until they are individually signalled to proceed. When the central control determines that it is safe for a train to continue (i.e., no train is on the common section of track), it selects one of the trains waiting and signals it to go on.

The intrinsics which manipulate semaphores in this version of Pascal are SEMINIT, SIGNAL, and WAIT, described in Sections 3.28, 3.29, and 3.43.

SEMINIT initializes a semaphore by assigning it a count and an empty queue. All semaphores must be initialized by SEMINIT, or their values and the results of the program are unpredictable.

WAIT causes a process to wait for a given semaphore, and SIGNAL informs the System that a semaphore is again available.

The use of these intrinsics is demonstrated in examples in the rest of this section.

### 5.2.1 Mutual Exclusion

When concurrent processes must share resources, it may be essential for only one process to access a particular resource at a given time. This is known as "mutual exclusion" and can be achieved by allowing the resource to be accessed only in "critical sections" of code to which the mutual exclusion criteria are applied.

Suppose, for example, that two processes must both display information on the screen and request input from the operator, but only one process may be allowed to do so at a time. These two processes must practice mutual exclusion with respect to the screen.

Critical sections can be implemented using Boolean semaphores by enclosing the critical section between WAIT(sem) and SIGNAL(sem). The semaphore should be initialized to 1.

The following program illustrates the use of semaphores to perform mutual exclusion.

```
Initialize: SEMINIT(bridge_empty, 1);
Critical Section:
Procedure CROSSBRIDGE;
begin
    WAIT(bridge_empty);
    ... { Critical section of code. }
    SIGNAL(bridge_empty);
end { CROSSBRIDGE };
```

In this example, processes ("trains") seeking to use the critical section (to cross a bridge that holds only one train at a time) call CROSSBRIDGE, which takes care of mutual exclusion internally via the global semaphore bridge\_empty.

## CONCURRENT PROCESSES

### 5.2.2 Synchronization

When concurrent processes are cooperating, you may want one process to wait at a certain point in its execution until another process has caused some event to occur, such as filling a buffer. A counting semaphore can be used as an "eventname" in this case. In the following example, two distinct "events" (the filling and emptying of a buffer) are used to synchronize two concurrent processes.

The following program illustrates synchronization.

```
PROGRAM BUFF;  
  const N = { Number of available buffers. };  
  var buff_full, buff_avail: semaphore;  
PROCESS FILL_BUFFER;  
  begin  
    repeat  
      wait(buff_avail);  
      ... { Select and fill a buffer. }  
      signal(buff_full)  
    until false;  
  end;  
PROCESS SEND_BUFFER;  
  begin  
    repeat  
      wait(buff_full);  
      ... { Select and send a buffer. }  
      signal(buff_avail)  
    until false;  
  end;  
begin { BUFF }  
  seminit(buff_full, 0);  
  seminit(buff_avail, N);  
  start(FILL_BUFFER);  
  start(SEND_BUFFER);  
  ...  
end.
```



### 5.3 OTHER FEATURES

As noted above, there is a predefined type PROCESSID. A value of type PROCESSID can be returned upon the invocation of a process. In the present implementation, PROCESSIDs are not considered a user-oriented feature, but are used for Operating System work. Variables of type PROCESSID can be used in expressions in the same way as pointer variables (that is, only the operators <>, =, and := are legal).

All processes must be declared at the outer (global) block of a program. They cannot be declared within a procedure or another process. Process initiation must occur in the principal task of a program. That is, a process cannot be started from any of a program's subsidiary processes.

Users interested in using processes at a fairly low level, especially using them in conjunction with the System's facilities for memory management and Heap control, should refer to the Internal Architecture Guide for further details.

## SECTION 6: TEXAS INSTRUMENTS UNITS

The Texas Instruments Home Computer has many capabilities that are not available with standard Pascal statements. However, UNITS have been written to give you access to these capabilities through Pascal. The UNITS are contained in SYSTEM.LIBRARY.

For sprites (moving graphics) and sounds, the UNITS allow you to set up a complex sequence of instructions that are performed concurrently with program execution. This concurrency lets you present complex visual and auditory displays at the same time that the computer is accepting input and processing information, which is useful especially for educational and recreational applications.

The UNITS are SUPPORT, RANDOM, MISC, SOUND, BEEP, SPRITE, and SPEECH.

- SUPPORT allows you to set character colors, screen colors, and patterns; obtain character patterns; turn the screen off; read the Wired Remote Controller's position; and set the screen display mode (pattern, multi-color, or text).
- RANDOM provides for generation of pseudo-random numbers.
- MISC lets you determine the values in strings and change strings to all upper-case letters.
- SOUND can be used to create a broad spectrum of notes and noises and coordinate those sounds with the rest of your program.
- BEEP is a subset of the procedures in the UNIT SOUND. It allows you to use sounds without using as much memory as the UNIT SOUND does.
- SPRITE permits you to create and delete sprites (moving graphics), adjust their size and speed, and determine when they are coincident.
- SPEECH allows you to use speech when a Speech Synthesizer, sold separately, is attached to the console.

To access the functions and procedures within the UNITS, include a statement in your program which consists of USES followed by the name of the UNIT used by the program. This section contains descriptions of the UNITS.

## 6.1 SUPPORT PROCEDURES AND FUNCTIONS

The UNIT SUPPORT allows you to set character colors, screen colors, and patterns; obtain character patterns; turn the screen off; read the Wired Remote Controller's position; and set the screen display mode (pattern, multi-color, or text). To access these procedures and functions, include USES SUPPORT; in your program.

The procedures and functions included in SUPPORT are listed below.

<u>Section</u>	<u>Name</u>	<u>Description</u>
6.1.1	CHR_DEFAULT	Sets the characters to their default definitions.
6.1.2	SET_PATTERN	Sets the pattern of a character.
6.1.3	GET_PATTERN	Returns the pattern of a character.
6.1.4	SET_CHR_COLOR	Sets the foreground and background colors of a character.
6.1.5	SET_SCREEN	Sets the screen mode.
6.1.6	SET_SCR_COLOR	Sets the screen color.
6.1.7	JOY	Returns the location of a Wired Remote Controller's lever.

### 6.1.1 CHR\_DEFAULT

CHR\_DEFAULT is a procedure with the form

```
CHR_DEFAULT;
```

CHR\_DEFAULT resets the character definitions for characters 0 through 255 to their standard representation. For example, if a character has been defined as a sprite pattern, then calling the procedure CHR\_DEFAULT changes the sprite's appearance to the standard definition of that character. Characters 32 through 126 are the ASCII character set associated with the keyboard. See the Appendix, Section 8.10.

### 6.1.2 SET\_PATTERN

SET\_PATTERN is a procedure with the form

```
SET_PATTERN (CHARACTER_NUMBER: INTEGER, PATTERN_STRING: STRING);
```

SET\_PATTERN allows you to define special graphics characters. You can redefine any of the standard group of characters (ASCII characters 0 through 127) or the other characters (ASCII characters 128 through 255).

## TEXAS INSTRUMENTS UNITS

CHARACTER\_NUMBER is an integer from 0 through 255 that indicates the character to be defined. PATTERN\_STRING is a string up to 16 characters long which specifies the pattern of the character you are defining. This string is a coded representation of the design which makes up a character displayed on the screen. The design is made of pixels or dots, which are the smallest units on the screen that can be turned on and off. The display screen is 256 pixels wide and 192 pixels high.

Characters are defined by turning some pixels "on" and leaving others "off." The space character (ASCII character 32) is a character with all the pixels turned "off." Turning all the pixels "on" produces a solid block. All the standard characters are set with the appropriate pixels on.

Each character is made up of 64 pixels comprising an 8-by-8 grid as shown below.

	LEFT	RIGHT
	BLOCKS	BLOCKS
ROW 1		
ROW 2		
ROW 3		
ROW 4		
ROW 5		
ROW 6		
ROW 7		
ROW 8		

Each row is divided into two blocks of four pixels each.

ANY ROW		
	LEFT	RIGHT
	BLOCKS	BLOCKS

Each character in PATTERN\_STRING describes the pattern in one block of one row. The rows are defined from left to right and from top to bottom. Therefore, the first two characters in PATTERN\_STRING describe the pattern for row one of the grid, the next two the second row, and so on.

To create a new character, specify which pixels to turn on and which to leave off. The code used in PATTERN\_STRING is the hexadecimal representation of a bit (binary digit) code. The following table shows all the possible on/off conditions for the four pixels in a given block and the binary and hexadecimal codes for each condition.

BLOCKS	Binary Code (0=Off; 1=On)	Hexadecimal Code
	0000	0
*	0001	1
*	0010	2
* *	0011	3
*	0100	4
*   *	0101	5
* *	0110	6
* * *	0111	7
*	1000	8
*     *	1001	9
*   *	1010	A
*   * *	1011	B
* *	1100	C
* *   *	1101	D
* * *	1110	E
* * * *	1111	F

If the PATTERN\_STRING is less than 16 characters, the computer assumes that the remaining characters are zeros.

## TEXAS INSTRUMENTS UNITS

For example, the PATTERN\_STRING "1898FF3D3C3CE404" describes the pattern shown below.

	LEFT BLOCKS	RIGHT BLOCKS	BLOCK CODES
ROW 1	*   *	18	
ROW 2	*       *   *	98	
ROW 3	*   *   *   *   *   *   *	FF	
ROW 4	*   *   *   *     *	3D	
ROW 5	*   *   *   *	3C	
ROW 6	*   *   *   *	3C	
ROW 7	*   *   *       *	E4	
ROW 8	*	04	

The following program uses this and one other character to make a figure "dance."

```

program dance;
uses support;
const a = '1898FF3D3C3CE404';
      b = '1819FFBC3C3C2320';
var c: integer;
procedure delay(time: integer);
begin
  repeat
    time := time-1
  until time<1
end;
begin
  { Main program. }
  page(output);      { Clear screen. }
  set_screen(2);
  set_chr_color(96,1,7);
  gotoxy(15,11);
  write(chr(96));     { Put the character on the screen. }
  for c:=1 to 100 do
  begin
    set_pattern(96,a);
    delay(1000);
    set_pattern(96,b);
    delay(1000)
  end;
end.

```

### 6.1.3 GET\_PATTERN

GET\_PATTERN is a procedure with the form

```

GET_PATTERN (CHARACTER_NUMBER: INTEGER; VAR PATTERN
STRING: STRING);

```

GET\_PATTERN returns in PATTERN\_STRING a string that specifies, in hexadecimal notation, the pattern defined for the character specified by the integer CHARACTER\_NUMBER. The hexadecimal notation used is the same as that described in the procedure SET\_PATTERN, Section 6.1.2.

## TEXAS INSTRUMENTS UNITS

### 6.1.4 SET\_CHR\_COLOR

SET\_CHR\_COLOR is a procedure with the form

```
SET_CHR_COLOR (CHARACTER_NUMBER, FOREGROUND_COLOR, BACKGROUND_COLOR: INTEGER);
```

SET\_CHR\_COLOR sets the colors of characters. The colors for all of the characters in a character set are set by specifying the color for any character within that set. Character sets consist of eight consecutive characters.

#### Character Sets

0-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63
64-71	72-79	80-87	88-95	96-103	104-111	112-119	120-127
128-135	136-143	144-151	152-159	160-167	168-175	176-183	184-191
192-199	200-207	108-215	216-223	224-231	232-239	240-247	248-255

CHARACTER\_NUMBER is an integer from 0 through 255 that indicates the character that is to be used. FOREGROUND\_COLOR sets the color of the pixels that are "on," and BACKGROUND\_COLOR sets the color of the pixels that are "off."

FOREGROUND\_COLOR and BACKGROUND\_COLOR can be integers from 0 through 15 and correspond to the following colors.

<u>Color</u>	<u>Code</u>	<u>Color</u>	<u>Code</u>
Transparent	0	Medium Red	8
Black	1	Light Red	9
Medium Green	2	Dark Yellow	10
Light Green	3	Light Yellow	11
Dark Blue	4	Dark Green	12
Light Blue	5	Magenta	13
Dark Red	6	Gray	14
Cyan	7	White	15

SET\_CHR\_COLOR works properly only if the computer is in the pattern mode, which is set by the SET\_SCREEN procedure.



SET\_CHR\_COLOR(44,6,4) sets characters 40 through 47 to have a FOREGROUND COLOR of dark red and a BACKGROUND\_COLOR of dark blue.

### 6.1.5 SET\_SCREEN

SET\_SCREEN is a procedure with the form

```
SET_SCREEN (SCREEN_MODE: INTEGER);
```

SET\_SCREEN sets the screen mode according to the value of SCREEN\_MODE. If SCREEN\_MODE is 0, the screen is turned off. If SCREEN\_MODE is 1, the display is put in text mode. If SCREEN\_MODE is 2, the display is put in pattern mode. If SCREEN\_MODE is 3, the display is put in multicolor mode.

When the screen is turned off, no text or graphics are displayed. The screen is the color defined by BACKGROUND\_COLOR in the SET\_SCR\_COLOR procedure.

Text mode allows the display of ASCII characters 0 through 255. The screen is 40 characters wide and 24 lines high. Each character is six by eight pixels. This is the default mode of the computer when Pascal is running.

Pattern mode allows the use of colored characters and sprites. The screen is 32 characters wide and 24 lines high. Each character is eight by eight pixels.

Multicolor mode allows the use of colored boxes and sprites. The screen is divided into 48 rows, each containing 64 "boxes" that are four by four pixels. Each of the 3072 boxes thus defined can be one of the 16 colors available.

### 6.1.6 SET\_SCR\_COLOR

SET\_SCR\_COLOR is a procedure with the form

```
SET_SCR_COLOR (FOREGROUND_COLOR, BACKGROUND_COLOR: INTEGER);
```

SET\_SCR\_COLOR sets the foreground and background colors of the screen. The foreground color is the color of the text on the screen in text mode, and the background color is the background color of the screen in text mode and the backdrop color in pattern mode.

The colors produced by different values of FOREGROUND\_COLOR and BACKGROUND\_COLOR are given in SET\_CHR\_COLOR, Section 6.1.4, and in the Appendix, Section 8.12.

### 6.1.7 JOY

JOY is a function with the form

```
JOY (STICK_NUM:INTEGER; VAR X,Y:INTEGER): BOOLEAN;
```

JOY returns the X- and Y-positions of the Wired Remote Controller specified by STICK\_NUM, and a Boolean value specifying whether the fire button has been pressed.

A value of 0 or 1 in STICK\_NUM specifies which Wired Remote Controller is to be read.

An X value of 1 indicates that the Wired Remote Controller is to the right. A value of 0 indicates that it is in the center. A value of -1 indicates that it is to the left.

A Y value of 1 indicates that the Wired Remote Controller is up. A value of 0 indicates that it is in the center. A value of -1 indicates that it is down.

If the value of the function is returned as false, the fire button has not been pressed. A true value indicates that the fire button has been pressed.

## 6.2 RANDOM NUMBERS

The UNIT RANDOM contains two procedures and two functions to enable a program to use random numbers. Random number use is initiated with the SET\_RND or RANDOMIZE procedures. Then random integers are obtained with the RND\_INT function and random real numbers are obtained with the RND\_REAL function. To access these procedures and functions, include USES RANDOM; in your program.

The procedures and functions included in RANDOM are listed below.

Section	Name	Description
6.2.1	SET_RND	Initializes the pseudo-random number generator.
6.2.2	RANDOMIZE	Randomizes the pseudo-random number generator.
6.2.3	RND_INT	Returns a random integer.
6.2.4	RND_REAL	Returns a random real number.

### 6.2.1 SET\_RND

SET\_RND is a procedure with the form

```
SET_RND (SEED1, SEED2:REAL);
```

SET\_RND initializes the pseudo-random number generator. Different real values of SEED1 and SEED2 give different random number sequences.

### 6.2.2 RANDOMIZE

RANDOMIZE is a procedure with the form

```
RANDOMIZE;
```

RANDOMIZE initializes the pseudo-random number generator using seeds taken from the system clock.

SET\_RND is automatically executed when you include USES RANDOM; in your program. Thus, each time you run your program the same series of pseudo-random numbers is produced if you do not execute SET\_RND or RANDOMIZE.

## TEXAS INSTRUMENTS UNITS

### 6.2.3 RND\_INT

RND\_INT is a function with the form

```
RND_INT (MAXIMUM_VALUE): INTEGER;
```

RND\_INT returns an integer from 1 through MAXIMUM\_VALUE. MAXIMUM\_VALUE can be up to 32,767, so the function can never return a value greater than 32,767.

### 6.2.4 RND\_REAL

RND\_REAL is a function with the form

```
RND_REAL: REAL;
```

RND\_REAL returns a random real number from 0 up to, but not including, 1.

The following function returns a random real number between any two values previously specified for LOWER\_LIMIT and UPPER\_LIMIT.

```
function random(lower_limit,upper_limit: real):real;  
begin  
  random := (upper_limit-lower_limit)*(rnd_real)+lower_limit  
end;
```

### 6.3 STRINGS

The functions in the UNIT MISC give you additional string and character capabilities. To access these procedures and functions, include USES MISC; in your program.

The procedures and functions included in MISC are listed below.

<u>Section</u>	<u>Name</u>	<u>Description</u>
6.3.1	BREAK	Returns the position of the first character in a string that matches a character in a given string.
6.3.2	SPAN	Returns the position of the first character in a string that does not match any character in a given string.
6.3.3	UPPER_CASE	Returns an upper-case copy of a string.

#### 6.3.1 BREAK

BREAK is a function with the form

```
BREAK (SOURCE_STRING, BREAK_STRING: STRING): INTEGER;
```

BREAK compares SOURCE\_STRING with BREAK\_STRING and returns the position of the first character in SOURCE\_STRING that matches a character in BREAK\_STRING.

If the statement

```
int := break(str1, str2);
```

is used with INT as an integer and STR1 and STR2 as strings, the following results occur.

<u>STR1</u>	<u>STR2</u>	<u>INT</u>	<u>Explanation</u>
'abcdefgh'	'd'	4	'd' is the fourth character in STR1.
'abcdefgh'	'dfg'	4	'd' is the fourth character in STR1.
'abcdefgh'	'dfb'	2	'b' is the second character in STR1.
'abcdefgh'	'a'	1	'a' is the first character in STR1.
'abcdefgh'	'x'	0	No match.

### 6.3.2 SPAN

SPAN is a function with the form

```
SPAN (SOURCE_STRING, SPAN_STRING:STRING): INTEGER;
```

SPAN compares SOURCE\_STRING with SPAN\_STRING and returns the position of the first character in SOURCE\_STRING which does not match any character in SPAN\_STRING.

If the statement

```
int := span(str1,str2);
```

is used with INT an integer and STR1 and STR2 strings, the following results occur.

<u>STR1</u>	<u>STR2</u>	<u>INT</u>	<u>Explanation</u>
'abcdefgh'	'a'	2	'b' is not in 'a'.
'abcdefgh'	'acb'	4	'd' is not in 'acb'.
'abcdefgh'	'b'	1	'a' is not in 'b'.
'abcdefgh'	'abcdefgh'	0	The entire string is included.

### 6.3.3 UPPER\_CASE

UPPER\_CASE is a procedure with the form

```
UPPER_CASE (SOURCE_STRING:STRING; VAR NEW_STRING:STRING);
```

UPPER\_CASE returns the SOURCE\_STRING in NEW\_STRING with all lower-case letters changed to upper-case letters.

The following statement returns "THIS HAS A 6 IN IT." in the string variable NEWT.

```
UPPER_CASE('This has a 6 in it.',newt);
```

## 6.4 SOUND PROCESSING

Sounds can be set up with the procedures and functions in the UNIT SOUND or UNIT BEEP and run without further program control, enabling you to add sounds and coordinate them with the display. This concurrency allows you to present complex visual and auditory displays at the same time that the computer is accepting input and processing information, which is useful especially for educational and recreational applications. To access these procedures and functions, include USES SOUND; or USES BEEP; in your program.

Sound production requires that you first reserve space in memory for the various sound possibilities, next put commands and sounds in that space, and finally have the computer perform the list of commands and sounds. The functions and procedures described below allow you to create and delete sound lists, place sounds and commands in sound lists, and find information about sound lists.

The procedures that are included in the UNIT SOUND but excluded from the UNIT BEEP are as follows.

CALL_SND	GOSUB_SND	RETURN_SND	JUMP_SND
CHAIN_SND	CHN_SND_CHAIN	READ_SND_CHAIN	WRITE_SND_LIST
READ_SND_LIST	PLAY_ALL_SND	KILL_ALL_SND	SET_SND_FLAG
READ_SND_FLAG	SND_BEAT	SND_LST_OFFSET	

The UNIT BEEP is useful for simple sound production. It is provided for use when the advanced procedures and functions in the UNIT SOUND are not needed and when memory space is a problem. You may use only UNIT SOUND or UNIT BEEP, but not both.

When you include USES SOUND; or USES BEEP; in your program, the following declaration is included in your program as part of the unit.

```

type  sndlstptr = ^sndlstrec;
      sndlstrec = record
        listsize: integer;
        curoffset: integer;
        packet: packed array[0..1] of 0..255;
      end;
```

## TEXAS INSTRUMENTS UNITS

The procedures and functions included in SOUND and BEEP are listed below.

<u>Section</u>	<u>Name</u>	<u>Description</u>
6.4.1	MAKE_SND_LIST	Allocates memory space for a sound list. Included in both units SOUND and BEEP.
6.4.2	DEL_SND_LIST	Deletes a sound list. Included in both units SOUND and BEEP.
6.4.3	SND_NOTE	Sets the frequency and duration of a note. Included in both units SOUND and BEEP.
6.4.4	SND_TONE	Sets the frequency and duration of a tone. Included in both units SOUND and BEEP.
6.4.5	WHITE_NOISE	Sets the duration and type of a white noise. Included in both units SOUND and BEEP.
6.4.6	PERIODIC_NOISE	Sets the duration and type of a periodic noise. Included in both units SOUND and BEEP.
6.4.7	SND_VOLUME	Sets the volume of a sound list. Included in both units SOUND and BEEP.
6.4.8	CALL_SND	Allows the use of sound lists as subroutines. Included in unit SOUND only.
6.4.9	GOSUB_SND	Transfers processing to another point in a sound list. Included in unit SOUND only.
6.4.10	RETURN_SND	Returns processing from one point in a sound list to another. Included in unit SOUND only.
6.4.11	JUMP_SND	Unconditionally transfers processing from one point in a sound list to another. Included in unit SOUND only.
6.4.12	CHAIN_SND	Transfers processing from one sound list to another. Included in unit SOUND only.
6.4.13	CHN_SND_CHAIN	Changes the sound list to which processing is transferred by CHAIN_SND. Included in unit SOUND only.
6.4.14	READ_SND_CHAIN	Returns the sound list to which processing is transferred by CHAIN_SND. Included in unit SOUND only.
6.4.15	WRITE_SND_LIST	Writes a sound list to a file. Included in unit SOUND only.
6.4.16	READ_SND_LIST	Reads a sound list from a file. Included in unit SOUND only.
6.4.17	END_SND	Ends a sound list. Included in both units SOUND and BEEP.



6.4.18	SET_SND	Associates a sound list with a sound generator. Included in both units SOUND and BEEP.
6.4.19	PLAY_SND	Plays a sound list. Included in both units SOUND and BEEP.
6.4.20	PLAY_ALL_SND	Plays all sound lists. Included in unit SOUND only.
6.4.21	KILL_SND	Stops playing of a sound list. Included in both units SOUND and BEEP.
6.4.22	KILL_ALL_SND	Stops playing of all sound lists. Included in unit SOUND only.
6.4.23	SET_SND_TEMPO	Sets the tempo of a sound list. Included in both units SOUND and BEEP.
6.4.24	SET_SND_FLAG	Sets a flag in a sound list. Included in unit SOUND only.
6.4.25	READ_SND_FLAG	Returns the number of the most recently encountered flag in a sound list. Included in unit SOUND only.
6.4.26	SND_BEAT	Returns the number of beats played since PLAY_ALL_SND or PLAY_SND was executed. Included in unit SOUND only.
6.4.27	SND_LST_OFFSET	Returns the number of bytes from the beginning of a sound list to the current point of execution. Included in unit SOUND only.

#### 6.4.1 MAKE\_SND\_LIST

MAKE\_SND\_LIST is a procedure with the form

```
MAKE_SND_LIST (VAR LIST_POINTER:SNDLSTPTR; SIZE:INTEGER);
```

MAKE\_SND\_LIST allocates memory space for a sound list. It is included in both SOUND and BEEP.

LIST\_POINTER is the SNDLSTPTR, indicating where the sound list is in memory. SIZE is listsize, indicating the number of bytes to reserve for the sound routine. Each of the commands in a sound list requires a certain number of bytes, as given in the description of the procedure that puts the command in the sound list.

The following statement allocates a sound list of 200 bytes and returns a pointer to it in voice one.

```
make_snd_list(voice_one,200);
```

### 6.4.2 DEL\_SND\_LIST

DEL\_SND\_LIST is a procedure with the form

```
DEL_SND_LIST (VAR LIST_POINTER:SNDLSTPTR);
```

DEL\_SND\_LIST deletes the sound list specified by LIST\_POINTER, making the memory it was using available for other applications. It is included in both SOUND and BEEP. The value of LIST\_POINTER is returned as nil.

### 6.4.3 SND\_NOTE

SND\_NOTE is a procedure with the form

```
SND_NOTE (LIST_POINTER:SNDLSTPTR; FREQUENCY,DURATION:INTEGER);
```

SND\_NOTE sets the FREQUENCY and DURATION for a note and adds the note to the list specified by the value of LIST\_POINTER. It is included in both SOUND and BEEP. FREQUENCY is an integer from 110 to 16383 Hertz. See the Appendix in Section 8.11 for information on relating FREQUENCY to musical notes.

The DURATION, in beats, is an integer from 1 to 16. A note sounds for seven-eighths of DURATION and is silent for one-eighth of DURATION. LIST\_POINTER must refer to a sound list created with MAKE\_SND\_LIST.

This procedure uses three bytes in the sound list.

The following statement adds an A below middle C (220 Hertz) with a duration of 4 beats (3-1/2 tone and 1/2 silence) to the sound list FIRST.

```
snd_note(first,220,4);
```

#### 6.4.4 SND\_TONE

SND\_TONE is a procedure with the form

```
SND_TONE (LIST_POINTER:SNDLSTPTR; FREQUENCY,DURATION:INTEGER);
```

SND\_TONE sets FREQUENCY and DURATION for a note and adds the note to the list specified by the value of LIST\_POINTER. It is included in both SOUND and BEEP. FREQUENCY is an integer from 110 to 16383 Hertz. See the Appendix in Section 8.11 for information on relating FREQUENCY to musical notes.

The DURATION, in beats, is an integer from 1 to 16. A tone lasts for the entire DURATION specified, with no silence. LIST\_POINTER must refer to a sound list created with MAKE\_SND\_LIST.

This procedure uses three bytes in the sound list.

The following statement adds an A below middle C (220 Hertz) with a duration of 4 beats to the sound list FIRST.

```
snd_tone(first,220,4);
```

#### 6.4.5 WHITE\_NOISE

WHITE\_NOISE is a procedure with the form

```
WHITE_NOISE (LIST_POINTER:SNDLSTPTR; NOISE,DURATION:INTEGER);
```

WHITE\_NOISE sets DURATION for a noise and adds the noise to the list specified by the value of LIST\_POINTER. It is included in both SOUND and BEEP. DURATION, in beats, is an integer from 1 to 16. LIST\_POINTER must refer to a sound list created with MAKE\_SND\_LIST.

Three different white noises are created from NOISE values of 0, 1, and 2. The white noise created by a NOISE value of 3 depends on the value of tone generator number 3.

This procedure uses three bytes in the sound list.

#### 6.4.6 PERIODIC\_NOISE

PERIODIC\_NOISE is a procedure with the form

```
PERIODIC_NOISE (LIST_POINTER:SNDLSTPTR; NOISE,DURATION:
INTEGER);
```

PERIODIC\_NOISE sets DURATION for a noise and adds the noise to the list specified by the value of LIST\_POINTER. It is included in both SOUND and BEEP. DURATION, in beats, is an integer from 1 to 16. LIST\_POINTER must refer to a sound list created with MAKE\_SND\_LIST.

Three different periodic noises are created from NOISE values of 0, 1, and 2. The periodic noise created by a NOISE value of 3 depends on the value of tone generator number 3.

This procedure uses three bytes in the sound list.

#### 6.4.7 SND\_VOLUME

SND\_VOLUME is a procedure with the form

```
SND_VOLUME (LIST_POINTER:SNDLSTPTR; VOLUME:INTEGER);
```

SND\_VOLUME adds a VOLUME setting as an integer from 0 (softest) through 15 (loudest) to the sound list specified by LIST\_POINTER. It is included in both SOUND and BEEP. Of course, the volume setting on the monitor or television also influences the volume.

This procedure uses two bytes in the sound list.

#### 6.4.8 CALL\_SND

CALL\_SND is a procedure with the form

```
CALL_SND (LIST_POINTER, NEW_LIST_POINTER: SNDLSTPTR);
```

CALL\_SND allows separate sound lists to be used as sound subroutines. It is included in only SOUND. LIST\_POINTER specifies the sound list into which the subroutine call is to be placed. NEW\_LIST\_POINTER gives the new sound list that is to be called. The entry point of NEW SOUND\_LIST is the beginning of the list. Processing in the new list proceeds until RETURN\_SND is encountered. Processing then returns to the next command in the original list.

This procedure uses three bytes in the sound list.

The following shows two sound lists.

LIST1	LIST2
note1a	note2a
note1b	note2b
CALL_SND list2	note3b
note1c	RETURN_SND

The notes are played in the following order when LIST1 is played.

```
note1a  
note1b  
note2a  
note2b  
note3b  
note1c
```

## TEXAS INSTRUMENTS UNITS

### 6.4.9 GOSUB\_SND

GOSUB\_SND is a procedure with the form

```
GOSUB_SND (LIST_POINTER:SNDLSTPTR; OFFSET:INTEGER);
```

GOSUB\_SND allows you to call a subroutine within a sound list. It is included in only SOUND. LIST\_POINTER specifies the list in which the command is to be placed. OFFSET is an integer specifying where to transfer control, counting the bytes from the beginning of the sound list. Execution continues at the new point until RETURN SND is encountered.

This procedure uses three bytes in the sound list.

The following shows the possible makeup of a sound list.

<u>Item</u>	<u>First byte</u>
note1	0
GOSUB_SND 18	3
note2	6
GOSUB_SND 18	9
note3	12
JUMP_SND 28	15
note4	18
note5	21
note6	24
RETURN_SND	27
END_SND	28

The notes are played in the following order.

```
note1  
note4  
note5  
note6  
note2  
note4  
note5  
note6  
note3
```

#### **6.4.10 RETURN\_SND**

RETURN\_SND is a procedure with the form

```
RETURN_SND (LIST_POINTER:SNDLSTPTR);
```

RETURN\_SND returns processing to the statement following the CALL\_SND or GOSUB\_SND by which it was called. It is included in only SOUND. The command is added to the sound list specified by LIST\_POINTER.

This procedure uses one byte in the sound list.

See the example under GOSUB\_SND, Section 6.4.9.

#### **6.4.11 JUMP\_SND**

JUMP\_SND is a procedure with the form

```
JUMP_SND (LIST_POINTER:SNDLSTPTR; OFFSET:INTEGER);
```

JUMP\_SND unconditionally transfers control to another point within a sound list. It is included in only SOUND. LIST\_POINTER specifies the list in which the command is to be placed. OFFSET is an integer specifying where to transfer control, counting the bytes from the beginning of the sound list. Execution continues at the new point.

This procedure uses three bytes in the sound list.

See the example under GOSUB\_SND, Section 6.4.9.

#### **6.4.12 CHAIN\_SND**

CHAIN\_SND is a procedure with the form

```
CHAIN_SND (LIST_POINTER,NEW_LIST_POINTER:SNDLSTPTR);
```

CHAIN\_SND transfers control from one sound list to another. It is included in only SOUND. LIST\_POINTER identifies the sound list into which the command is entered. NEW\_LIST\_POINTER specifies the sound list to which control is transferred. The new sound list always runs from its beginning.

This procedure uses three bytes in the sound list.

#### 6.4.13 CHN\_SND\_CHAIN

CHN\_SND\_CHAIN is a procedure with the form

```
CHN_SND_CHAIN (LIST_POINTER:SNLSTPTR; CHAIN_NUMBER:INTEGER;  
NEW_LIST_POINTER:SNLSTPTR);
```

CHN\_SND\_CHAIN changes the NEW\_LIST\_POINTER value previously placed in a sound list by the CHAIN\_SND procedure. It is included in only SOUND. LIST\_POINTER identifies the sound list in which the value is to be changed. The integer CHAIN\_NUMBER indicates which CHAIN\_SND command is to be changed. NEW\_LIST\_POINTER specifies the new sound list.

Suppose sound LIST1 is as follows.

```
note1  
note2  
CHAIN_SND to list2
```

Then the statement

```
CHN_SND_CHAIN(list1,1,list5);
```

changes LIST1 to

```
note1  
note2  
CHAIN_SND to list5
```

#### 6.4.14 READ\_SND\_CHAIN

READ\_SND\_CHAIN is a function with the form

```
READ_SND_CHAIN (LIST_POINTER:SNLSTPTR; CHAIN_NUMBER:INTEGER):  
SNLSTPTR;
```

READ\_SND\_CHAIN returns the list pointer which was set by CHAIN\_SND and possibly reset by CHN\_SND\_CHAIN. It is included in only SOUND. LIST\_POINTER identifies the sound list from which the value is to be read. The integer CHAIN\_NUMBER indicates which CHAIN\_SND command is to be read.



Suppose sound LIST1 is as follows.

```
note1  
CHAIN_SND to list2  
note2  
CHAIN_SND to list3  
note3  
CHAIN_SND to list4
```

Then the statement

```
READ_SND_CHAIN(list1,2);
```

returns a value of LIST3.

#### **6.4.15 WRITE\_SND\_LIST**

WRITE\_SND\_LIST is a procedure with the form

```
WRITE_SND_LIST (LIST_POINTER:SNDLSTPTR; FILE_NAME:STRING);
```

WRITE\_SND\_LIST uses the pointer to a sound list in LIST\_POINTER to designate the sound list that is to be written to the file designated by FILE\_NAME. It is included in only SOUND. This enables you to save sound lists on diskettes.

#### **6.4.16 READ\_SND\_LIST**

READ\_SND\_LIST is a procedure with the form

```
READ_SND_LIST (VAR LIST_POINTER:SNDLSTPTR; FILE_NAME:STRING);
```

READ\_SND\_LIST allocates space for a sound list in main memory, assigns a pointer to the list in LIST\_POINTER, and reads the sound list from the file designated by FILE NAME. It is included in only SOUND. This enables you to read saved sound lists from diskettes. This procedure automatically allocates space for a sound list and returns the pointer to it in LIST\_POINTER.

#### **6.4.17 END\_SND**

END\_SND is a procedure with the form

```
END_SND (LIST_POINTER:SNDLSTPTR);
```

END\_SND provides the required end of a sound list. It is included in both SOUND and BEEP. LIST\_POINTER identifies the sound list to be ended. The END\_SND procedure adds a command to the end of the sound list specified.

This procedure uses one byte in the sound list.

#### **6.4.18 SET\_SND**

SET\_SND is a procedure with the form

```
SET_SND (VOICE_NUMBER:INTEGER; LIST_POINTER:SNDLSTPTR);
```

SET\_SND associates a specific sound list with a particular sound generator. It is included in both SOUND and BEEP. VOICE\_NUMBER is an integer specifying which of the four sound generators is to be used. VOICE\_NUMBERS of 1 through 3 can be used for notes or tones. VOICE\_NUMBER 4 is for noises only.

LIST\_POINTER specifies the sound list that is to be associated with the given sound generator. Note that this procedure does not play a sound list. It only associates a sound list with a particular sound generator.

#### **6.4.19 PLAY\_SND**

PLAY\_SND is a procedure with the form

```
PLAY_SND (VOICE_NUMBER:INTEGER);
```

PLAY\_SND starts the processing of the sound generator specified by the integer VOICE\_NUMBER. It is included in both SOUND and BEEP. The sound generator must have been previously associated with a list by the SET\_SND procedure.

#### **6.4.20 PLAY ALL\_SND**

PLAY ALL\_SND is a procedure with the form

```
PLAY ALL_SND;
```

PLAY ALL\_SND starts the processing of all sound generators. It is included in only SOUND. The sound generators must have been previously associated with sound lists by the SET\_SND procedure. Any sound generators not associated with a sound list remain silent.

#### **6.4.21 KILL\_SND**

KILL\_SND is a procedure with the form

```
KILL_SND (VOICE_NUMBER: INTEGER);
```

KILL\_SND stops the processing of the sound generator specified by the integer VOICE\_NUMBER. It is included in both SOUND and BEEP. VOICE\_NUMBER must be from 1 through 4.

#### **6.4.22 KILL\_ALL\_SND**

KILL\_ALL\_SND is a procedure with the form

```
KILL_ALL_SND;
```

KILL\_ALL\_SND stops the processing of all sound generators. It is included in only SOUND.

#### **6.4.23 SET\_SND\_TEMPO**

SET\_SND\_TEMPO is a procedure with the form

```
SET_SND_TEMPO (VOICE, DURATION: INTEGER);
```

SET\_SND\_TEMPO sets the duration of one beat of sound for the specified VOICE. It is included in both SOUND and BEEP. The integer DURATION specifies the number of milliseconds, from 1 through 32,767, that a beat lasts. The number of beats that a sound lasts is set by the SND\_NOTE or SND\_TONE procedure.

## TEXAS INSTRUMENTS UNITS

The time is processed every sixtieth of a second, or about every 16.7 milliseconds. The most accurate values, therefore, are in multiples of 16.7 milliseconds. For music, a good starting value is approximately 300 milliseconds.

Sound effects are best determined by trial-and-error. They often depend on short intervals and rapid frequency changes.

This procedure uses one byte in the sound list.

### **6.4.24 SET\_SND\_FLAG**

SET\_SND\_FLAG is a procedure with the form

```
SET_SND_FLAG (LIST_POINTER: SNDLSTPTR; FLAG_NUMBER: INTEGER);
```

With SET\_SND\_FLAG, you can set up to 15 "flags" in a sound list. It is included in only SOUND. These flags can be used during program execution to synchronize the sound with the rest of the program. LIST\_POINTER identifies the sound list into which the command is entered. FLAG\_NUMBER is an integer from 1 through 15.

When a flag is encountered during sound list processing, FLAG\_NUMBER is recorded and can be accessed with the READ\_SND\_FLAG function.

This procedure uses one byte in the sound list.

### **6.4.25 READ\_SND\_FLAG**

READ\_SND\_FLAG is a function with the form

```
READ_SND_FLAG (VOICE_NUMBER: INTEGER): INTEGER;
```

READ\_SND\_FLAG returns the number of the most recently encountered flag in the sound list associated with the integer VOICE\_NUMBER. It is included in only SOUND. VOICE\_NUMBER must be from 1 through 4 and must have been associated with a sound generator with the SET\_SND procedure. The sound flags are initialized to zero when USES SOUND; is included in your program, so a value of zero is returned if no flag has been encountered.

### 6.4.26 SND\_BEAT

SND\_BEAT is a function with the form

```
SND_BEAT (VOICE:INTEGER): INTEGER;
```

SND\_BEAT returns an integer equal to the number of beats played by the sound list specified in VOICE since the PLAY\_SND or PLAY ALL\_SND procedure was executed. SND\_BEAT is included in only SOUND. The value returned is from 0 through 32,767. If a value in excess of 32,767 occurs, 32,767 is returned.

The amount of time, in milliseconds, that has passed is equal to the value returned by the SND\_BEAT function multiplied by the length of a beat, for this voice, as set by the SET\_SND\_TEMPO procedure, Section 6.4.23.

The following is an example of the SND\_BEAT function.

```
repeat                { Do nothing. }
until SND_BEAT(1)>247; { Wait for music to catch up. }
```

### 6.4.27 SND\_LST\_OFFSET

SND\_LST\_OFFSET is a function with the form

```
SND_LST_OFFSET (LIST_POINTER:SNDLSTPTR): INTEGER;
```

SND\_LST\_OFFSET returns an integer which indicates the number of bytes from the beginning of the sound list specified by LIST\_POINTER to the current point of execution. It is included in only SOUND. If LIST\_POINTER does not point to an active sound list, a value of zero is returned.

Suppose sound LIST1 consists of the following. Then when the sound list is processed and note2 is being played, the function SND\_LST\_OFFSET (list1) returns a value of 8.

Values	First Byte	Byte Length
SND_VOLUME	1	2
note1	3	3
note2	6	3
SET_SND_FLAG	9	1
note3	10	3

## 6.5 SPRITE HANDLING

Sprites are graphics which have color and can be located anywhere on the screen. They can be set in motion in any direction at a variety of speeds and continue their motion until it is changed by the program. They move more smoothly than the usual character which jumps from one screen position to another. To access these procedures and functions, include `USES SPRITE;` in your program.

You can define up to 32 sprites, numbered 0 through 31, with a `SPRITE_CHANGE LIST`, which is a packed record defined as follows.

```
type
  link = ^sprite_change_list;
  sprite_change_list = packed record
    packet: set of (spr_pattern,spr
                    color,spr_clock,spr_y
                    pos,spr_x_pos,spr_y_vel,spr
                    x_vel);
    pattern_number: integer;
    color: integer;
    clock: integer;
    y_pos: integer;
    x_pos: integer;
    y_vel: integer;
    x_vel: integer;
    countdown: integer;
    link: ^sprite change list;
    auto_dispose: boolean;
  end { sprite_change_list };
```

Sprites are created and controlled by defining and accessing the values in the `SPRITE_CHANGE_LIST`.

The following describes the meanings of the variables in `SPRITE_CHANGE_LIST`.

- **PACKET**--A set which indicates the valid fields within the `SPRITE_CHANGE LIST`. This allows you to modify selected fields. For instance, the statement

```
sc11.packet := [spr_pattern,spr_color];
```

sets a list to alter only the pattern and color of a sprite.

- **PATTERN\_NUMBER**--The pattern which defines the sprite. It is an integer from 0 through 255 and refers to an ASCII character. The pattern can be defined with the SET\_PATTERN procedure in UNIT SUPPORT (see Section 6.1.2).
- **COLOR**--The color of the pixels that are "on" in the sprite's pattern. It is an integer from 0 through 15. The colors are as described in the SET\_CHR COLOR procedure in Section 6.1.4. The pixels that are "off" are always transparent.
- **CLOCK**--The side of the sprite which controls its position. The integer 0 means that the sprite's upper left-hand corner is the specified position. The integer 1 means that the sprite is moved 32 pixels to the left of the specified position. This allows you to control whether the sprite moves off the screen smoothly on the right (a CLOCK value of 0) or the left (a CLOCK value of 1).
- **Y\_POS**--The vertical (y) position of the sprite. The y-position is an integer from 0 (the top of the screen) through 191 (the bottom of the screen). When, because of its motion, the y-position of a sprite would equal 192, it automatically changes to 0. Similarly, when the y-position would be -1, it automatically changes to 191. Values can also be given from 192 to 255, hiding the sprite below the bottom of the screen. A value of 208 causes the sprite in that row and any higher numbered sprites to disappear until the sprite moves to another position.
- **X\_POS**--The horizontal (x) position of the sprite. The x-position is an integer from 0 (the left side of the screen) through 255 (the right side of the screen). When, because of its motion, the x-position of a sprite would equal 256, it automatically changes to 0. Similarly, when the x-position would be -1, it automatically changes to 255.
- **Y\_VEL**--The vertical (y) velocity of the sprite. The y-velocity is an integer from -128 (a fast upward movement) through 127 (a fast downward movement). The y-position is updated by  $Y\_VEL/32$  pixels every sixtieth of a second. A Y\_VEL of 0 indicates no vertical motion.
- **X\_VEL**--The horizontal (x) velocity of the sprite. The x-velocity is an integer from -128 (a fast leftward movement) through 127 (a fast rightward movement). The x-position is updated by  $X\_VEL/32$  pixels every sixtieth of a second. An X\_VEL of 0 indicates no horizontal motion.

## TEXAS INSTRUMENTS UNITS

- COUNTDOWN--The number of sixtieths of a second during which the sprite exists with its current attributes. After that, another Sprite change list is processed or the sprite ceases to exist. COUNTDOWN is an integer from 0 through 32,767. If an original value of 0 is given, the sprite continues with its current attributes.
- LINK--A pointer to the next motion for a sprite to have when COUNTDOWN reaches 0. If LINK is 0, the sprite ceases to exist when COUNTDOWN reaches 0.
- AUTO\_DISPOSE--Reserved for possible future use.

Sprites are coincident if any of the pixels that are "on" in any sprite overlap the pixels that are "on" in any other sprite. In case of coincidence, the sprite with the lowest number covers other sprites. If the pixels from more than four sprites appear anywhere on a horizontal screen line, the pixels on that line disappear except for the pixels belonging to the four sprites with the lowest sprite numbers.

As with sound processing, the appearance and motion of sprites continue without program control while other statements are executed.

The procedures and functions included in SPRITE are listed below.

<u>Section</u>	<u>Name</u>	<u>Description</u>
6.5.1	SET_SPRITE	Creates a sprite.
6.5.2	SET_SPR_ATTRIBUTE	Specifies the attributes of a sprite.
6.5.3	DEL_SPRITE	Deletes a sprite.
6.5.4	SET_SPR_SIZE	Sets the size of a sprite.
6.5.6	SPRITE_COINC	Returns whether any sprites are coincident.
6.5.7	PAST_SPRITE_COINC	Returns the time since any sprites were coincident.
6.5.8	GET_SPRITE	Returns the attributes of a sprite.

### 6.5.1 SET\_SPRITE

SET\_SPRITE is a procedure with the form

```
SET_SPRITE (SPRITE_NUMBER: INTEGER; PACKET: LINK);
```

SPRITE\_NUMBER is an integer from 0 through 31 which specifies the sprite to be affected. PACKET must have been set as described above.



As an example of sprites, the following program moves a sprite in a diamond on the screen.

```

program diamond;
uses sprite, support;
var scl1, scl2, scl3, scl4: link;
    ch: char;
begin
    new(scl1);
    new(scl2);
    new(scl3);
    new(scl4);
    scl1^.packet := [spr_pattern..spr_x_vel];
    with scl1^ do
        { Moves up and to the right. }
        begin
            pattern_number := 65;    { Letter A. }
            color := 4;              { Dark blue. }
            clock := 1;              { Right side defines
                                     x-position. }
            y_pos := 144;            { Starting y-position of 144. }
            x_pos := 128;            { Starting x-position of 128. }
            y_vel := -14;            { Starting y-velocity of -14. }
            x_vel := 14;             { Starting x-velocity of 14. }
            countdown := 60;         { Exists for about 1 second. }
            link := scl2;            { When countdown equals 0,
                                     attributes change to those
                                     described in scl2. }
        end;
    with scl2^ do
        { Moves up and to the left. }
        begin
            packet := [spr_y_vel, spr_x_vel]; { Only y-velocity and
                                                x-velocity are
                                                changed. }
            y_vel := -14;            { New y-velocity of -14. }
            x_vel := -14;            { New x-velocity of 14. }
            link := scl3;            { New link of scl3. }
            countdown := 60;         { Countdown of 60. }
        end;
    with scl3^ do
        { Moves down and to the left. }
        begin

```

```
    packet := [spr_y_vel, spr_x_vel]; { Only y-velocity and
                                      x-velocity are
                                      changed. }
    y_vel := 14;                      { New y-velocity of 14. }
    x_vel := -14;                     { New x-velocity of -14. }
    link := scl4;                     { New link of scl4. }
    countdown := 60;                  { Countdown of 60. }
end;
with scl4^ do                         { Moves down and to the right. }
begin
    packet := [spr_y_vel, spr_x_vel]; { Only y-velocity and
                                      x-velocity are
                                      changed. }
    y_vel := 14;                      { New y-velocity of 14. }
    x_vel := 14;                      { New x-velocity of 14. }
    link := scl1;                     { New link of scl1. }
    countdown := 60;                  { Countdown of 60. }
end;
page(output);                         { Clears the screen. }
set_screen(2);                        { Uses pattern mode. }
set_sprite(1, scl1);                  { Starts the sprite series. }
read(ch);                             { Waits until a character is
                                      typed. }
set_screen(1);                        { Deletes sprite and returns to
                                      text mode. }

end.
```

### 6.5.2 SET\_SPR\_ATTRIBUTE

SET\_SPR\_ATTRIBUTE is a procedure with the form

```
SET_SPR_ATTRIBUTE (SPRITE_NUMBER,PATTERN_NUMBER,COLOR,CLOCK,Y
POS,X_POS,Y_VEL,X_VEL:INTEGER);
```

SET\_SPR\_ATTRIBUTE specifies all of the attributes of a new sprite or changes the attributes of an existing sprite. The procedure constructs the correct SPRITE CHANGE\_LIST to define the specified sprite.

SPRITE\_NUMBER is an integer from 0 to 31, specifying the number that refers to the sprite.

See Section 6.5 for an explanation of PATTERN\_NUMBER, COLOR, CLOCK, Y\_POS, X\_POS, Y\_VEL, and X\_VEL.

### **6.5.3 DEL\_SPRITE**

DEL\_SPRITE is a procedure with the form

```
DEL_SPRITE (SPRITE_NUMBER:INTEGER);
```

DEL\_SPRITE creates the correct SPRITE\_CHANGE\_LIST to make the sprite specified by the integer SPRITE\_NUMBER disappear. The characteristics of the sprite are all set to 0, except for Y\_POSITION, which is set to 192.

### **6.5.4 SET\_SPR\_SIZE**

SET\_SPR\_SIZE is a procedure with the form

```
SET_SPR_SIZE (SIZE:INTEGER);
```

The SET\_SPR\_SIZE sets the size of all sprites according to the value of SIZE. A value of 0 sets all sprites to single-size. A value of 1 sets all sprites to magnified. A value of 2 sets all sprites to double size. A value of 3 sets all sprites to magnified, double size. The default value is 0.

A large sprite takes up four times as many pixels as a small sprite. Enlarging sprites makes the pixels of each sprite four times as large. The explanation is down and to the right if the value of CLOCK is 0, and down and to the left if the value of CLOCK is 1. The diagram on the next page shows how setting SIZE to 1 affects the sprite with the pattern string "8142241818422481".

## TEXAS INSTRUMENTS UNITS

SIZE equals 0;  
Single Size Sprite

```
|X| | | | | |X|
| |X| | | |X| |
| | |X| | |X| |
| | |X|X| | |
| | |X|X| | |
| | |X| | |X| |
| |X| | | |X| |
|X| | | | | |X|
```

Size equals 1;  
Magnified Sprite

```
|X|X| | | | | | | | |X|X| |
|X|X| | | | | | | | |X|X|
| | |X|X| | | | | | |X|X| |
| | |X|X| | | | | |X|X| |
| | | | |X|X| | | |X|X| |
| | | | |X|X| | | |X|X| |
| | | | |X|X|X|X| | | | |
| | | | |X|X|X|X| | | | |
| | | | |X|X|X|X| | | | |
| | | | |X|X| | | |X|X| |
| | | | |X|X| | | |X|X| |
| | |X|X| | | | | |X|X| |
| | |X|X| | | | | |X|X| |
|X|X| | | | | | | | |X|X|
|X|X| | | | | | | | |X|X|
```

A single size sprite is defined by only the character specified by PATTERN\_NUMBER when the sprite is created with the SET\_SPR\_ATTRIBUTE procedure or altered with the SET\_SPR\_PATTERN procedure.

A double size sprite is defined by four characters, including the one specified by PATTERN\_NUMBER when the sprite is created with the SET\_SPR\_ATTRIBUTE procedure or altered with the SET\_SPR\_PATTERN procedure. The expansion is down and to the right if the value of clock is 0. If the value of clock is 1, the expansion is down and to the left. The first character is the one specified when the sprite is created or altered, if that character is evenly divisible by 4, or the next smallest number that is evenly divisible by 4. That character defines the upper left corner of the sprite. The next character defines the lower left corner of the sprite. The next character defines the upper right corner of the sprite. The final character defines the lower right corner of the sprite.

Suppose the following characters have been defined.

<u>Number</u>	<u>Description</u>	<u>Portion of Double Sprite</u>
32	181818FFFF181818	Upper left corner
33	8142241818244281	Lower left corner
34	0000001818000000	Upper right corner
35	FF8181818181FF	Lower right corner

The following diagram shows the effect of the double sprite procedure on a sprite which was defined with a character number of 32, 33, 34, or 35. The upper left corner is character 32. The lower left corner is character 33. The upper right corner is character 34. The lower right corner is character 35.

SIZE equals 2;  
Double Size Sprite

```

| | | |X|X| | | | | | | | | | | |
| | | |X|X| | | | | | | | | |
| | | |X|X| | | | | | | | | |
|X|X|X|X|X|X|X|X| | | |X|X| | |
|X|X|X|X|X|X|X|X| | | |X|X| | |
| | | |X|X| | | | | | | | | |
| | | |X|X| | | | | | | | | |
| | | |X|X| | | | | | | | | |
|X| | | | | | |X|X|X|X|X|X|X|X|X|
| |X| | | | |X| |X| | | | | |X|
| | |X| | |X| |X| | | | | |X|
| | | |X|X| | | |X| | | | | |X|
| | | |X|X| | | |X| | | | | |X|
| | |X| | |X| |X| | | | | |X|
| |X| | | |X| |X| | | | | |X|
|X| | | | | |X|X|X|X|X|X|X|X|X|

```

## Magnified, Double Size Sprite

SPRITE\_COINC returns a Boolean value indicating whether any sprites are coincident when the function is executed. Sprites are coincident if any of the pixels that are "on" in any sprite overlap the pixels that are "on" in any other sprite.

#### **6.5.6 PAST\_SPRITE\_COINC**

PAST\_SPRITE\_COINC is a function with the form

```
PAST_SPRITE_COINC: INTEGER;
```

PAST\_SPRITE\_COINC returns an integer from 0 through 32,767, indicating how many sixtieths of a second have elapsed since the first coincidence of any sprites. The first coincidence of any sprites is the first coincidence that occurred between two sprites since the program started or since the last use of the PAST\_SPRITE\_COINC function. Sprites are coincident if any of the pixels that are "on" in any sprite overlap the pixels that are "on" in any other sprite.

After the maximum value of 32,767 is reached, no further updating of the value occurs. The value is changed back to 0 and the count begins again when the PAST\_SPRITE\_COINC function is executed.

#### **6.5.7 GET\_SPRITE**

GET\_SPRITE is a procedure with the form

```
GET_SPRITE (SPRITE_NUMBER: INTEGER; PACKET: LINK);
```

GET\_SPRITE reads the characteristics of the sprite specified by the integer SPRITE\_NUMBER and returns it in the sprite change list pointed to by PACKET. SPRITE\_NUMBER is an integer from 0 through 31. PACKET is described in Section 6.5. The sprite change list pointed to by PACKET contains PATTERN NUMBER, COLOR, CLOCK, Y\_POS, X\_POS, Y\_VEL, X\_VEL, and COUNTDOWN.

For an example, see the description of SET\_SPRITE, Section 6.5.1.

## TEXAS INSTRUMENTS UNITS

### 6.6 SPEECH HANDLING

Speech on the TI Home Computer requires that the TI Solid State Speech <sup>TM</sup> Synthesizer (sold separately) be attached to the computer.

In order to use speech, you must include the following statement in the declaration portion of your program.

```
type longstring = string[255];
```

The procedures and functions included in SPEECH are listed below.

<u>Section</u>	<u>Name</u>	<u>Description</u>
6.6.1	GET_SPEECH	Returns a speech data pattern.
6.6.2	SAY	Causes the computer to speak a word or phrase.

#### 6.6.1 GET\_SPEECH

GET\_SPEECH is a procedure with the form

```
GET_SPEECH (WORD_STRING:LONGSTRING; VAR RETURN  
STRING:LONGSTRING);
```

GET\_SPEECH returns in RETURN\_STRING the speech data pattern which corresponds to the first word or phrase in WORD\_STRING.

The value of WORD\_STRING is any string value listed in the Appendix, Section 8.15. The value of RETURN\_STRING is used with the SAY procedure. See the example with the explanation of the SAY procedure.

#### 6.6.2 SAY

SAY is a procedure with the form

```
SAY (SAY_STRING:LONGSTRING);
```

SAY causes the computer to speak the words, phrases, or speech data patterns in SAY\_STRING when the Speech Synthesizer is connected to the console.

Only the first 63 entries in SAY\_STRING are spoken. If there are more than 63 entries, the computer says "UHOH" to indicate that SAY\_STRING is too long.



The following program causes the computer to say "Hello how are you".

```
program speak;
uses speech;
type longstring=string[255];
var string1, string2: longstring;
begin
  GET_SPEECH('Hello', string1);
  GET_SPEECH('how', string2);
  SAY(concat(string1,string2,"are"," you"));
end.
```

## SECTION 7: USING THE COMPILER

The Pascal Compiler is based on the P2 portable compiler from the Eidgenossische Technische Hochschule in Zurich. It is used by selecting the C(ompile or R(un) command when the System promptline is displayed. If a work file exists, it is compiled. Otherwise, you are asked for a source file name. The Compiler generates code which can then be run by your computer.

While the Compiler is running, it displays a report of its progress on the screen. The report of a sample compilation of the program TEST, with procedures INITIALIZE and DELAY, might appear on the screen as follows.

Compiling...

Pascal compiler - release 99/4 IV.0 C1A-4

<0 >.....

INITIALIZE

<19 >.....

DELAY

<61 >.....

<111 >.....

TEST

<119 >.....

237 lines compiled

TEST

In the first pass, the Compiler displays the name of each routine. The numbers enclosed within angle brackets (< >) are the current line numbers, and each dot on the screen represents the compilation of one source line. In the second pass, each name is the name of a segment, and each dot represents one routine.

For a given compilation, this output can be suppressed with the Q+ Compiler option (see Section 7.1) or by setting HAS SLOW TERMINAL to TRUE in SYSTEM.MISCINFO (see the UCSD p-System Utilities manual).

The code file produced is \*SYSTEM.WRK.CODE if the source file was the work file or if you press <return> when asked for a code file name. If there is no work file, you are asked for both a source and code file name. Any file name is acceptable. The Compiler appends .TEXT to the source file name and .CODE to the code file name. The R(un) command can be used to execute the file SYSTEM.WRK.CODE. The X(ecute) command can be used to execute any code file. See the UCSD p-Code manual.

When the Compiler detects a syntax error, the text surrounding the error is displayed, along with an error number (or error message if \*SYSTEM.SYNTAX is on line) and "<----" pointing to the place in which the compiler detected an error.

If both the Q and L options are set (see Section 7.1), the compilation continues, the syntax error is reported in the listing file, and the screen remains undisturbed.

In the default situation, Q and L are both off (see Section 7.1), so the Compiler gives you the option of typing a space, a <return>, or E when an error occurs. Typing a space continues the compilation, <return> terminates the compilation, and E calls the Editor, which places the cursor at the symbol where the error was detected so that you can correct it.

The syntax errors detected by the Compiler are listed in the Appendix, Section 8.5. All error numbers are accompanied by a message after entering the Editor, provided \*SYSTEM.SYNTAX is available to the system. \*SYSTEM.SYNTAX is on the diskette which contains the Editor and Filer. Any error messages also appear on the screen.

## USING THE COMPILER

### 7.1 COMPILE-TIME OPTIONS

You can direct some of the Compiler's actions with compile-time options included in the source code. Compile-time options are a set of commands that appear within "pseudo-comments." A pseudo-comment is a comment with a dollar sign immediately following the left-hand delimiter. The following are examples of pseudo-comments.

```
{ $I+ }  
{ $U MOLD.CODE }  
(* $I+, S-, L+*)  
{ $R }
```

The two kinds of compile-time options are "switch" options and "string" options. A switch option is one of the letters described below followed by a "+", "-", or "^". A string option is a letter followed by a character string. A pseudo-comment can contain any number of switch options (separated by commas) and one string option. If a string option is present in a pseudo-comment, it must be the last option. The string is delimited by the option letter and the end of the comment.

If the pseudo-comment uses braces ({ and }), the string in a string option cannot contain an asterisk (\*). String options use the string following them. Switch options are either toggles or stack options. If a switch option is a toggle, a "+" turns it on, and a "-" turns it off.

The options I and R are stack options, as are the conditional compilation flags. With each stack option, the current state, "+" or "-", is saved on the top of a stack, up to 15 states deep. The stack can be "popped" by a "^", which re-enables the previous state of that option. If the stack is "pushed" deeper than 15 states, the bottom state of the stack is lost. If the stack is popped when it is empty, the value is always "-".

The following illustrate the use of the stack with the I and R options.

```
{ $I- } ... current value is "-" so there is no I/O checking.  
...  
{ $I+ } ... current value is "+".  
...  
{ $I^ } ... current value is "-" again.  
...  
{ $I^ } ... current value is "+" because this was the default.  
...  
{ $I^ } ... current value is "-" because the stack is now empty.
```

The default options for a compilation are as follows.

{ \$R+, I+, L-, U+, P+ }

These defaults remain in effect unless you override them. The Q option defaults to the Q- unless the value of the HAS SLOW TERMINAL data item has been changed. HAS SLOW TERMINAL can be set in SYSTEM.MISCINFO (see the UCSD p-System Utilities manual).

Compile-time options also control conditional compilation, discussed in Section 7.2.

Individual options are listed below in alphabetical order and discussed in more detail on the following pages.

- B: Begin conditionally compiled source code.
- C: Copyright notice insertion.
- D: Declare or alter value of a conditional computation flag.
- E: End conditionally compiled source code.
- I: (1) Input/output check control.  
(2) Include a file.
- L: (1) List a file control.  
(2) File to write a listing to.
- P: Pagination control.
- Q: Quiet console--determines output to the screen.
- R: Range checking control.
- T: Title insertion.
- U: (1) User or System compilation indicator.  
(2) Use a library.

### **7.1.1 Compile-Time Option Descriptions**

The following are descriptions of each of the compile-time options.

- B: B is a string option. It begins a section of conditionally compiled source code. See Section 7.2.
- C: C is a string option. It places the string directly into the copyright field of the code file's segment dictionary. This lets you include a copyright notice in the code file.

## USING THE COMPILER

- D: D is a string option. It declares or alters the value of a conditional compilation flag. See Section 7.2.
- E: E is a string option. It ends a section of conditionally compiled source code. See Section 7.2.
- I: Two options are named "I". One is a stack switch option (IOCHECK), and the other is a string option (INCLUDE).

### IOCHECK OPTION

Default value: I+

- I+: Instructs the Compiler to generate code after each I/O statement, in order to check that the I/O operation was successful. If not, the program terminates with a runtime error.
- I-: Instructs the Compiler not to generate any I/O checking code. In the case of an unsuccessful I/O operation, the program continues.

The I- option is helpful for testing IORESULT (see Section 3) when there is the chance of an I/O failure but the program should not be stopped. If I- is used and you do not test IORESULT, the effects are unpredictable. IORESULTS are listed in Section 3 and the Appendix, Section 8.2.

### INCLUDE FILE MECHANISM

The string, delimited by the letter "I" and the end of the comment, is interpreted as the name of a file. If that file can be found, it is included in the source file and compiled.

For example,

```
{ $I BOLA }
```

includes the file BOLA.TEXT in the program's source.

If the initial attempt to open the file to be included fails, the Compiler concatenates a ".TEXT" to the file name and tries again. If the second attempt fails or an I/O error occurs while reading the include file, the Compiler responds with a fatal syntax error.

Included files can be nested up to three files deep.

If a file name begins with a "+" or "-", a blank must be inserted between the letter I and the string, as shown in the following example.

```
{ $I +BLBD.BBW }
```

- L: L can be used either as a toggle switch option or a string option. The default is L-, which prevents a listing from being generated. An L+ enables listing. If no listing file is named, the Compiler writes to \*SYSTEM.LST.TEXT. You can specify a different name for a listing file by using L as a string option, as illustrated in the following example.

```
{ $L DEMO1.TEXT }
```

writes to DEMO1.TEXT on the default diskette. Listing files which are sent to the diskette can be edited the same as any other text file, provided they are created with a .TEXT suffix. Without the .TEXT suffix, the System treats the listing as a data file.

Some lines are commented out with braces ({ }) to warn you that a comment may have accidentally removed some Pascal code. The numbers that precede the other source line are as follows.

- The line number.
- The segment number.
- The routine number:lexical level.
- The number of words of data or code storage which the routine requires at that point.

Rather than a lexical level, declaration lines show a "D" following the procedure number.

## USING THE COMPILER

Here is a portion of a listing with errors.

```
596 10 1:5 228 lastpageitem:=min(lastentry,lastentry);
-----> Error #104 <-----
597 10 1:5 239
598 10 1:5 239 { Loop through the page. }
599 10 1:5 239 PageInx:=0;
600 10 1:5 242 { Function returns next greater. }
601 10 1:5 242 Repeat {Until found or (PageInx>lastentry).}
602 10 1:6 242 Assert(PageInx<lastpageitem,'bad PageInx');
-----> Error #104 <-----
The previous error is on line 596
607 10 1:6 271 found:=(data{ PageInx }.key>key);
```

The error messages indicate the position of the previous error. The Compiler also lists readable error messages from \*SYSTEM.SYNTAX, provided that file is available to the System.

Regardless of whether the compilation is completed, the listing is saved.

P: P is a switch option. P- turns off pagination in the listing, P+ (the default) turns it on again, and a P by itself starts a new page in the listing.

Q: Q is the "quiet compile" option. It suppresses the Compiler's output to the screen. The default value is Q-, which uses the value of SLOWTERM in \*SYSTEM.MISCINFO (see the ucsd p-System Utilities manual).

Q+ causes the Compiler to suppress output to the screen, while Q- causes the Compiler to check to see if SLOWTERM is TRUE or FALSE. If SLOWTERM is TRUE, then information is not sent to the screen. If it is FALSE, then information is sent to the screen. On the Home Computer the default is Q- and program information is sent to the screen.

R: R is a stack switch option. The default value is R+, which turns range checking on. R- turns range checking off.

Programs compiled with the R- option set run faster and require less space. However, if an invalid index occurs or an invalid assignment is made, the



program is not terminated with a runtime error, and the results are exceedingly difficult to debug. Until a program has been completely tested, it is strongly advised to compile it with the R+ option left on.

- T: T is a string option. The string becomes the new title of the pages in the listing file.
- U: Two options are indicated by U. One is a toggle switch option (USER PROGRAM), and the other is a string option (USE LIBRARY).

#### USER PROGRAM OPTION

This option determines whether this compilation is a user program compilation or a compilation of a System program. If present, it must appear before the reserved word PROGRAM or UNIT.

The default value is U+, which specifies user source. U- allows compilation of units with names that are predeclared in the System. U- also sets R- and I-. The average user never uses this option.

#### USE LIBRARY OPTION

This is a string option with the string interpreted as a file name. If the file named in the U option can be found, the Compiler searches it for the code of UNITS named in subsequent USES declarations. If a UNIT is not found, the Compiler searches \*SYSTEM.LIBRARY.

If a program contains USES declarations but no U option, the Compiler looks for the USED UNITS first in the source file itself, and then in \*SYSTEM.LIBRARY.

The following is an example of a valid USES clause using the "U" option.

```
USES UNIT1, UNIT2, { Found in *SYSTEM.LIBRARY. }  
  {$U A.CODE}  
  UNIT3,  
  {$U B.LIBRARY}  
  UNIT4, UNIT5;
```

**Note:** SCREENOPS.CODE and COMMANDIO.CODE, on the Compiler diskette, are libraries used by some of the UCSD Pascal intrinsics described in Section 3.

## 7.2 CONDITIONAL COMPILATION

Portions of source code can be conditionally compiled. Whether they are compiled depends on the value of a flag that is declared by a compile-time option at the beginning of the source file.

A section of source code to be compiled conditionally must be delimited by the options B and E. Both of these options must name the flag which determines whether the code is compiled. The flag itself is declared by a D option at the beginning of the source. D options can change the value of an existing flag at other locations in the source.

Each flag in a program must appear in a D option before the source heading. The flag name follows the rules for Pascal identifiers. If the flag name is followed by a "-", that flag is set FALSE. The flag can be followed by a "+", which sets it to TRUE. If no sign is present, a flag is TRUE. The flag name can also be followed by a "'", as described below.

The state of a flag can be changed by a D option which appears after the source heading. If the flag has not been declared, an error results. The B and E options delimit a section of code to be compiled conditionally. When the Compiler encounters a B option, it scans for an E option which names the same flag and resumes compilation from that point. The B option can follow the flag name with a "-", which causes the delimited code to be compiled if the flag is FALSE. In the absence of a "-", the code is compiled if the flag is TRUE. Although the flag name can also be followed by a "+" or "'", these are ignored. In an E option, the flag name can be followed by a "+", "-", or "'". However, these symbols are ignored.

The state of each flag is saved in a stack, just as the state of a stack switch option is saved. Thus, using a D option with "'" yields the previous value of the flag. Each flag's stack is 15 values deep. If a 16th value is pushed, the bottom of the stack is lost. If an empty stack is popped with "'", the value returned is always FALSE.

If a section of code is not compiled, any pseudo-comments it may contain are ignored.

The following example illustrates the use of conditional compilation options.

```
{ $D DEBUG- } { Declares DEBUG and sets it FALSE. }
PROGRAM SIMPLE;
...
BEGIN
    { $D DEBUG+ }      { Changes DEBUG to TRUE. }
    ...
    { $B DEBUG }       { If DEBUG is TRUE, this section is compiled. }
    WRITELN('There is a bug. ');
    { $E DEBUG }       { This ends the section. }
    ...
    { $D DEBUG^ }      { Restores previous value of DEBUG--in this
                        case, FALSE. }
    { $B DEBUG- }      { If DEBUG is FALSE, this section is compiled. }
    WRITELN('Nothing has failed. ');
    { $E DEBUG }
    ...
END                                { SIMPLE }.
```

## SECTION 8: APPENDICES

The following are the appendices contained in this section.

<u>Appendix</u>	<u>Section</u>
Execution Errors	8.1
I/O Results	8.2
Device Numbers	8.3
Pascal Syntax Errors	8.4
Summary of Differences between UCSD Pascal and Standard Pascal	8.5
Summary of Differences Between Versions	8.6
Converting Programs for use under IV.0	8.7
Reserved Words	8.8
Assembler Syntax Errors	8.9
American Standard Code for Information Interchange (ASCII)	8.10
Musical Tone Frequencies	8.11
Color Codes	8.12
High-Resolution Color Combinations	8.13
Mathematical Functions	8.14
List of Speech Words	8.15
Program Development with Multi-Drive Systems	8.16

## 8.1 EXECUTION ERRORS

- 0     System error... FATAL
- 1     Invalid index, value out of range
- 2     No segment, bad code file
- 3     Procedure not present at exit time
- 4     Stack overflow
- 5     Integer overflow
- 6     Divide by zero
- 7     Invalid memory reference <bus timed out>
- 8     User break
- 9     System I/O error... FATAL
- 10    User I/O error
- 11    Unimplemented instruction
- 12    Floating point math error
- 13    String too long
- 14    Halt, Breakpoint
- 15    Bad Block

All run time errors cause the System to Initialize itself; the errors marked FATAL also cause the System to reinitialize. Some FATAL errors leave the System in an irreparable state, in which case you must reinitialize the system by turning the computer off and starting it again.

## APPENDICES

### 8.2 I/O RESULTS

- 0 No error
- 1 Bad block, parity error (CRC)
- 2 Bad device number
- 3 Illegal I/O request
- 4 I/O operation cancelled by user (REMIN:, REMOUT:, PRINTER:)
- 5 Volume is no longer on-line
- 6 File is no longer in directory
- 7 Bad file name
- 8 No room, insufficient space on volume
- 9 No such volume on-line
- 10 No such file on volume
- 11 Duplicate directory entry
- 12 Not closed: attempt to open an open file
- 13 Not open: attempt to access a closed file
- 14 Bad format: error in reading real or integer
- 15 Ring buffer overflow
- 16 Volume is write-protected
- 17 Illegal block number
- 18 Illegal buffer

See also the information in Section 6.2.14, IORESULT.

### 8.3 DEVICE NUMBERS

<u>Device Number</u>	<u>Volume Name</u>	<u>Description</u>
0		System use.
1	CONSOLE:	Keyboard and display with echo.
2	SYSTEM:	Keyboard and display without echo.
3	GRAPHIC:	
4	Disk	First disk drive.
5	Disk	Second disk drive.
6	PRINTER:	9600 Baud RS232 input/output.
7	REMIN:	300 Baud RS232 input.
8	REMOUT:	300 Baud RS232 output.
9	Disk	Third disk drive.
10		User-defined disk or other device.
11		User-defined disk or other device.
14	OS:	System use.
31	TAPE:	Cassette tape.
32	TP:	Thermal Printer.

Devices with numbers 9 or greater are user-defined devices. Devices 4 and 5 are diskettes. REMIN: and REMOUT: are often set to the same bidirectional port.

More information on devices can be found in the UCSD P-Code and Filer manuals.

## APPENDICES

### 8.4 PASCAL SYNTAX ERRORS

- 1: Error in simple type
- 2: Identifier expected
- 3: Unimplemented error
- 4: ')' expected
- 5: ': ' expected
- 6: Illegal symbol (terminator expected)
- 7: Error in parameter list
- 8: 'OF' expected
- 9: '(' expected
  
- 10: Error in type
- 11: '^' expected
- 12: ']' expected
- 13: 'END' expected
- 14: ';' expected
- 15: Integer expected
- 16: '=' expected
- 17: 'BEGIN' expected
- 18: Error in declaration part
- 19: Error in <field-list>
  
- 20: '.' expected
- 21: '\*' expected
- 22: 'INTERFACE' expected
- 23: 'IMPLEMENTATION' expected
- 24: 'UNIT' expected
  
- 50: Error in constant
- 51: ': =' expected
- 52: 'THEN' expected
- 53: 'UNTIL' expected
- 54: 'DO' expected
- 55: 'TO' or 'DOWNT0' expected in for statement
- 56: 'IF' expected
- 57: 'FILE' expected
- 58: Error in <factor> (bad expression)
- 59: Error in variable



- 60: Must be of type 'SEMAPHORE'
- 61: Must be of type 'PROCESSID'
- 62: Process not allowed at this nesting level
- 63: Only main task may start processes
  
- 101: Identifier declared twice
- 102: Low bound exceeds high bound
- 103: Identifier is not of the appropriate class
- 104: Undeclared identifier
- 105: Sign not allowed
- 106: Number expected
- 107: Incompatible subrange types
- 108: File not allowed here
- 109: Type must not be real
  
- 110: <tagfield> type must be scalar or subrange
- 111: Incompatible with <tagfield> part
- 112: Index type must not be real
- 113: Index type must be a scalar or a subrange
- 114: Base type must not be real
- 115: Base type must be a scalar or a subrange
- 116: Error in type of standard procedure parameter
- 117: Unsatisfied forward reference
- 118: Forward reference type identifier in variable declaration
- 119: Re-specified params not OK for a forward declared procedure
  
- 120: Function result type must be scalar, subrange or pointer
- 121: File value parameter not allowed
- 122: A forward declared function's result type can't be re-specified
- 123: Missing result type in function declaration
- 124: F-format for reals only
- 125: Error in type of standard procedure parameter
- 126: Number of parameters does not agree with declaration
- 127: Illegal parameter substitution
- 128: Result type does not agree with declaration
- 129: Type conflict of operands

## APPENDICES

- 130: Expression is not of set type
- 131: Tests on equality allowed only
- 132: Strict inclusion not allowed
- 133: File comparison not allowed
- 134: Illegal type of operand(s)
- 135: Type of operand must be Boolean
- 136: Set element type must be scalar or subrange
- 137: Set element types must be compatible
- 138: Type of variable is not array
- 139: Index type is not compatible with the declaration
  
- 140: Type of variable is not record
- 141: Type of variable must be file or pointer
- 142: Illegal parameter solution
- 143: Illegal type of loop control variable
- 144: Illegal type of expression
- 145: Type conflict
- 146: Assignment of files not allowed
- 147: Label type incompatible with selecting expression
- 148: Subrange bounds must be scalar
- 149: Index type must be integer
  
- 150: Assignment to standard function is not allowed
- 151: Assignment to formal function is not allowed
- 152: No such field in this record
- 153: Type error in read
- 154: Actual parameter must be a variable
- 155: Control variable cannot be formal or non-local
- 156: Multidefined case label
- 157: Too many cases in case statement
- 158: No such variant in this record
- 159: Real or string tagfields not allowed

- 160: Previous declaration was not forward
- 161: Again forward declared
- 162: Parameter size must be constant
- 163: Missing variant in declaration
- 164: Substitution of standard proc/func not allowed
- 165: Multidefined label
- 166: Multideclared label
- 167: Undeclared label
- 168: Undefined label
- 169: Error in base set
  
- 170: Value parameter expected
- 171: Standard file was re-declared
- 172: Undeclared external file
- 173: FORTRAN procedure or function expected
- 174: Pascal function or procedure expected
- 175: Semaphore value parameter not allowed
  
- 182: Nested UNITs not allowed
- 183: External declaration not allowed at this nesting level
- 184: External declaration not allowed in INTERFACE section
- 185: Segment declaration not allowed in INTERFACE section
- 186: Labels not allowed in INTERFACE section
- 187: Attempt to open library unsuccessful
- 188: UNIT not declared in previous uses declaration
- 189: 'USES' not allowed at this nesting level
  
- 190: UNIT not in library
- 191: Forward declaration was not segment
- 192: Forward declaration was segment
- 193: Not enough room for this operation
- 194: Flag must be declared at top of program
- 195: Unit not importable
  
- 201: Error in real number--digit expected
- 202: String constant must not exceed source line
- 203: Integer constant exceeds range
- 204: 8 or 9 in octal number

## APPENDICES

- 250: Too many scopes of nested identifiers
- 251: Too many nested procedures or functions
- 252: Too many forward references of procedure entries
- 253: Procedure too long
- 254: Too many long constants in this procedure
- 256: Too many external references
- 257: Too many externals
- 258: Too many local files
- 259: Expression too complicated
  
- 300: Division by zero
- 301: No case provided for this value
- 302: Index expression out of bounds
- 303: Value to be assigned is out of bounds
- 304: Element expression out of range
  
- 398: Implementation restriction
- 399: Implementation restriction
  
- 400: Illegal character in text
- 401: Unexpected end of input
- 402: Error in writing code file, not enough room
- 403: Error in reading include file
- 404: Error in writing list file, not enough room
- 405: 'PROGRAM' or 'UNIT' expected
- 406: Include file not legal
- 407: Include file nesting limit exceeded
- 408: INTERFACE section not contained in one file
- 409: Unit name reserved for system
  
- 410: disk error
  
- 500: Assembler error

## **8.5 SUMMARY OF DIFFERENCES BETWEEN UCSD PASCAL AND STANDARD PASCAL**

The following summarize the attributes of UCSD Pascal that are different from Standard Pascal. The differences include string handling, input/output intrinsics, memory management, concurrency, and miscellaneous differences. Section 8.5.6 gives some suggestions for writing a program that can be used in different versions of Pascal.

### **8.5.1 String Handling**

STRING is an intrinsic data type, consisting of a PACKED ARRAY OF CHAR together with a length. Strings can be assigned, passed, and input or output.

The following UCSD intrinsics are for the manipulation of strings.

```
function CONCAT (source [,source...:string]): string

function COPY (source:string; index,size:integer): string

procedure DELETE (destination:string; index,size:integer )

procedure INSERT (source,destination:string; size:integer)

function LENGTH (source:string): integer

function POS (pattern,source:string ): integer
```

### **8.5.2 I/O Intrinsics**

READ, READLN, WRITE, and WRITELN can only be used on files of type TEXT, which is a FILE OF CHAR, in Standard Pascal. In UCSD Pascal, they may also be used on files of type INTERACTIVE with a slightly different meaning for READ and READLN.

In addition to the Standard file types, files can be untyped or INTERACTIVE. The predefined files INPUT, OUTPUT, and KEYBOARD, are INTERACTIVE in UCSD Pascal. KEYBOARD is a non-echoing equivalent of INPUT.

## APPENDICES

If a file is INTERACTIVE, the EOF function is set by input of an <etx> character, which is defined in SYSTEM.MISCINFO (see the UCSD Pascal Utilities manual), the EOLN function is set by a <return>, READ and READLN perform a GET before loading the file's window variable requiring that a READ or READLN is required on an INTERACTIVE file before testing EOF or EOLN, and RESET does not load the file's window variable.

If a file is untyped, all I/O to that file must use the BLOCKREAD and BLOCKWRITE intrinsics.

RESET and REWRITE generally behave the same as Standard intrinsics, but they both can take an optional second parameter that is a diskette file name. This parameter makes the Pascal file equivalent to the physical diskette file.

The intrinsic SEEK does random access on files. The intrinsic CLOSE controls the closing of a diskette file. UNITREAD, UNITWRITE, and other UNIT intrinsics are for direct control of peripheral devices. IORESULT returns the status of an I/O operation.

WRITE and WRITELN are incapable of writing Booleans or record variables. STRINGs and PACKED ARRAYs OF CHAR can be output in a single WRITE.

The following are the UCSD intrinsics that handle devices and files.

```
function BLOCKREAD (fileid:{untyped}file; buffer:packed array
                    of char; blocks [,relblock]:integer):
                    integer
```

```
function BLOCKWRITE (fileid:{untyped}file; buffer:packed array
                    of char; blocks [,relblock]:integer):
                    integer
```

```
procedure CLOSE (fileid:{any sort of}file; <,option>)
                { <option> is LOCK, NORMAL, PURGE, or
                  CRUNCH. })
```

```
function IORESULT: integer
```

```
procedure SEEK (fileid:{structured}file; recnum:integer)
```

```
function UNITBUSY (unitnumber:integer): Boolean
```

```
procedure UNITCLEAR (unitnumber:integer)
```

```
procedure UNITREAD (unitnumber:integer; buffer:packed array of char; length  
                    [,blocknumber] [,option]]:integer)
```

```
procedure UNITWAIT (unitnumber:integer)
```

```
procedure UNITWRITE (unitnumber:integer; buffer:packed array of char; length  
                    [,blocknumber] [,option]]:integer)
```

### 8.5.3 Memory Management

A SEGMENT PROCEDURE behaves the same as any other procedure but is diskette-resident and present in main memory only when it is being executed.

A UNIT is a separately compiled collection of procedures and data structures. The following is an outline of a UNIT.

```
UNIT <unitname>;
```

```
INTERFACE
```

```
    {declarations and procedure headings appear here.}  
    {these and only these can be used by the host.}
```

```
IMPLEMENTATION
```

```
    {declarations and procedure code appear here.}  
    {this portion is private to the UNIT.}
```

```
BEGIN
```

```
    {initialization code.}  
    ***;  
    {termination code.}
```

```
END
```

## APPENDICES

The initialization code is executed before any host program code. The host program invokes a unit by code such as the following.

```
PROGRAM <program_name>;  
  USES <unitname> <,more_unitnames ... >;
```

The Standard intrinsics NEW and DISPOSE are implemented.

The following UCSD intrinsics are available for memory management.

```
procedure MARK (var heapptr:^integer)  
  
function MEMAVAIL: integer  
  
procedure MEMLOCK (seglist:string)  
  
procedure MEMSWAP (seglist:string)  
  
procedure RELEASE (var heapptr:^integer)  
  
function VARAVAIL (seglist:string): integer  
  
procedure VARDISPOSE (pointer:^{any type}; count:integer)  
  
procedure VARNEW (pointer:^{any type}; count:integer)
```

### **8.5.4 Concurrency**

A PROCESS is declared in the same way as a procedure, and can be STARTed any number of times by the main program. Processes can be controlled by semaphores. The UCSD predeclared type SEMAPHORE is the subrange 0 through 32,767. The UCSD predeclared type PROCESSID is used only by the System.

The following program outline shows the use of a PROCESS.

```
PROCESS ZIP;  
  BEGIN ... END;  
process DINNER (var SPLIT, BLACKKEYED: peas);  
  begin ... end;
```



The following UCSD intrinsics are for the control of processes.

```
procedure ATTACH (sem:semaphore; vector:integer)

procedure SEMINIT (var sem:semaphore; sem_count:integer)

procedure SIGNAL (var sem:semaphore)

procedure START (<process call>; [,id:processid;]
                [,stacksize:integer;]
                [,priority:byterange])
  { <process call> is {a normal procedure
    call} type byterange: 0..255 }

procedure WAIT (var sem:semaphore)
```

#### **8.5.5 Miscellaneous**

The following syntax exists in UCSD Pascal and not in Standard Pascal.

CASE statements fall through if no label matches the selector.

Comments can be enclosed by either "{ }" or "(\* \*)"; the two different types can be nested (only one comment deep).

"=" and "<>" can be used for extended array or record comparisons.

GOTOs are restricted to labels within the same block.

Procedure EXIT ( procid: <procedure identifier> ) is used to immediately stop a procedure.

A length attribute defines a LONG INTEGER. For example, the following defines LONG as a variable with up to eight digits.

```
var LONG: integer[8];
```

Procedure STR ( value: integer[n]; destination: string ) converts an integer into a string. It is usually used for the output of long integers. The length attribute is optional.

## APPENDICES

PACK and UNPACK are not implemented. Packing and unpacking are done automatically. A PACKED ARRAY OF CHAR can be assigned, input, and output as a single entity, as with a STRING.

Packed variables cannot be used as call-by-reference (var) parameters.

Sets of subranges of integers must include only positive integers.

Set comparisons must be between sets of the same underlying type.

The arctangent function can be called either ATAN or ARCTAN.

The following UCSD intrinsics are for the handling of large arrays.

```
procedure FILLCHAR (destination:packed array of char;  
                    length:integer; character:char)
```

```
procedure MOVELEFT (source,destination {any sort of} array;  
                    length:integer)
```

```
procedure MOVERIGHT (source,destination {any sort of} array;  
                     length:integer)
```

```
function SCAN (length:integer; <partial expression>;  
               source:packed array of char): integer  
{ <partial expression> is '<char>' or  
<>'<char>'. }
```

```
function SIZEOF ({any variable or type identifier}): integer
```

The following are miscellaneous UCSD intrinsics.

```
procedure GOTOXY (x,y:integer)
```

```
procedure HALT
```

```
function PWROFTEN (exponent:integer): real
```

```
procedure TIME (var hiword,loword:integer)
```

### **8.5.6 Writing a Transportable Program**

The following are a few hints and suggestions for writing a program that can be used in different versions of Pascal.

- Avoid the abilities of UCSD Pascal detailed above.
- Untagged case variant records often cause trouble. The value of the case tag is either checked by the run-time system or not at all.
- Assume nothing about variable allocation. The size of variables, packing algorithms, and representations of real numbers and Booleans all vary from system to system.
- Make sure variables are unique in the first 8 characters.
- Do not assume that all of an expression will be evaluated. Some compilers try to optimize around subexpressions.

## APPENDICES

### 8.6 SUMMARY OF DIFFERENCES BETWEEN SYSTEM VERSIONS

The UCSD p-System has gone through a number of versions since its first release. The names it has borne are: I.3, I.4, I.5, II.0, II.1, III.0, and IV.0. Most changes to the System have expanded its capabilities. The single-user microprocessor environment, portable code, and hierarchical operating system are features of the design which have not changed. Increasing the capabilities has led to a proliferation and diversification of features. This trend has been countered by efforts for standardization and portable code. The latest release, IV.0, was designed to incorporate the capabilities of II.0, II.1, and III.0, while cleaning up some rough edges of the user interface, UCSD Pascal code, and System internals.

IV.0 offers upward compatibility at the source code level, introduces multitasking to interpreter-based implementations of UCSD Pascal, and provides more flexible and cleaner memory-management techniques than previous versions.

Before new changes are explained in detail, here is a bit of history.

After a series of releases internal to UCSD and its computer science program, I.3 was made available to the general public. It was a very simple and very stable version of the System. Although a screen-oriented editor had existed for some time, I.3's System editor was YALOE (Yet Another Line Oriented Editor).

I.4 was the first version to be available on other microprocessors. I.4 also introduced the full Screen Oriented Editor.

I.5 introduced separate compilation and assembly. External routines and UNITS could be bound into host programs with the Linker. Still more microprocessors were supported.

II.0 was essentially a more stable version of I.5. It was released by UCSD shortly before SofTech Microsystems assumed responsibility for Pascal licensing and support.

II.1 has the INTRINSIC UNIT feature and a number of minor differences.

III.0 runs on a hardware-emulated processor, thus requiring many changes, mostly internal. At the level of Pascal object code, III.0 introduced concurrent procedures called processes.

IV.0 is new and pulls together the user-level features of the last three versions.

### 8.6.1 Version IV.0

The following describe some things which you must keep in mind when translating programs written in earlier versions of UCSD Pascal to release IV.0.

1. Media--The logical format of diskette directories and diskette files has not changed; therefore, no conversion of text or data is required.
2. Source Code--Pascal source from versions II.0, II.1, and III.0 will compile under IV.0. Most programs will then run. Those that will not are programs dependent on former implementations of the System's data structures and memory management, or possibly dependent on the memory requirements of a given machine.
3. Object Code--Old programs must be recompiled.
4. Pascal--Has been extended with the PROCESS construct for concurrency. SEPARATE UNITS and INTRINSIC UNITS no longer exist, although they will still be compiled as regular UNITS. UNITS need not be bound in by the Linker and therefore can be shared. The IMPLEMENTATION part of a UNIT can contain SEGMENT PROCEDURES. A program can refer to up to 256 compilation units, and a compilation unit can refer to up to 256 segments and can contain up to 16 segments.
5. The Editors--The Screen Oriented Editor remains much the same; X(change is more flexible, and a K(olumn command has been added.
6. The Assemblers--No macro parameters are allowed within ASCII strings, the radix switch characters have changed, alphabetic alternatives to some special characters are provided, and relocatable procedures have been added. Old assembly language procedures which use type STRING and old assembly language FUNCTIONS require some changes to run under IV.0.
7. Memory Management--SEGMENT routines can be declared, as in earlier versions. A compilation module (program or UNIT) can contain up to 16 segments. The bodies of all segment routines must be declared before the bodies of any non-segment routines are declared. The Standard Pascal intrinsics NEW and DISPOSE are now implemented. UCSD intrinsics MEMLOCK, MEMSWAP, VARAVAIL, VARNEW, and VARDISPOSE have been added.

## APPENDICES

8. External Compilation--There is now only one type of UNIT. INTRINSIC and SEPARATE UNITs which exist in old programs will be compiled into regular IV.0 UNITs. A IV.0 UNIT is like an old II.1 INTRINSIC UNIT in that it need not be linked and can be shared, but is unlike an INTRINSIC UNIT in that it does not have a fixed segment number. UNITs can now contain SEGMENT routines which must be declared in their IMPLEMENTATION part.
9. Concurrency--As in version III.0, you can declare a PROCESS which is declared like a procedure but is started by the UCSD intrinsic START. Once a process is STARTed, it appears to run simultaneously with the host program and (possibly) other processes until it is complete. The predeclared type SEMAPHORE has been introduced to aid in process synchronization. SEMAPHOREs can be manipulated with the intrinsics SIGNAL and WAIT.
10. Internals--The codes have been slightly modified, and run time memory management has changed. Rather than being placed on the Stack, procedure code now resides in a "code pool" which resides between the Stack and the Heap, and is relocatable. The code pool is a highly flexible structure, and allows for much run time swapping. In addition, the following UCSD intrinsics have been created to aid in system-level memory management: MEMLOCK, MEMSWAP, VARAVAIL, VARNEW, and VARDISPOSE.
11. Disk Swapping--Since code is swapped more frequently in IV.0, a number of prompts have been added which request that you insert a needed volume.
12. Incompatibilities--The following practices, which run under II.0, II.1, or III.0, require modification before a program can run under version IV.0.

System Data Structure Dependencies--Many System data structures have changed. Therefore, programs which directly access such things as SYSCOM, SIBs, etc. will have to be modified.

Heap Storage Utilization--A program cannot assume that the memory immediately following that obtained by a NEW is unoccupied and available.

Similarly, consecutive calls to NEW do not necessarily yield a contiguous area of memory. The practice of indexing across the boundary separating storage obtained by consecutive calls to NEW will fail under version IV.0.

Calls to MARK and RELEASE must be paired correctly. The pointer value obtained by calling MARK must not be modified prior to calling RELEASE. Furthermore, the pointer obtained from MARK cannot be used as a base pointer for storage references.

Tightly Fitting Programs--IV.0 in general uses more memory at run time than previous versions, so programs that have been tailored to fit in main memory will possibly need to be tailored again. The improved memory management in IV.0 should make this an easier task than it has been in the past.

## APPENDICES

### **8.7 CONVERTING PROGRAMS FOR USE UNDER IV.0**

This section discusses how to go about converting programs written in another version of Pascal to this Pascal, and how to convert assembly language programs so that they can use this version of Pascal.

#### **8.7.1 Converting Pascal Programs**

This section describes changes that must be made to Pascal programs in order to run them on the IV.0 System. Some of the changes are concerned with interfaces to the System; others affect version II and III programming practices.

##### **8.7.1.1 Use (and Misuse) of the Heap**

Version IV.0 is the first version of the UCSD p-System to implement a true Heap as defined in Standard Pascal. For this reason, most of the programming tricks associated with the rudimentary Heap implementations of past versions no longer work.

Consecutive calls to the Standard procedure NEW no longer guarantee the allocation of a contiguous area of memory. Therefore, creating variable-sized buffers using a sequence of NEWs does not work. The UCSD intrinsics VARNEW and VARDISPOSE should now be used to allocate variable-sized buffers. The version IV.0 Heap is as sensitive to range violations as the stack has always been, so use it with corresponding care.

The Standard procedures MARK and RELEASE must be used only for the purposes for which they were devised. Using a MARKed pointer as a pointer to Heap data does not work in version IV.0. The contents of a MARKed pointer must not be altered in any way until the matching call to RELEASE has been performed. RELEASEs must only be performed on variables that have been previously MARKed (and not yet RELEASEd).

##### **8.7.1.2 Code Segment Management**

With the code pool scheme, code segments need to be loaded from diskette much more frequently (and less predictably) than in the past. Several System segments may require loading during the course of a single System call, so the System diskette must be on-line to complete the call. This can affect the usefulness of programs which manipulate the diskette volumes, such as the Filer.



Two solutions address this problem. A program can use the memory management procedure MEMLOCK to lock into the code pool all code segments required for its execution. The procedure MEMSWAP can later be used to unlock these segments. Note that segments should not be left locked if they do not need to be, as locked segments use much space and can slow the Operating System.

The other solution is more direct, but possibly less efficient. If direct control of code residency is undesirable, the System prompts you to place the proper diskette in a drive so the required code segment can be loaded.

### **8.7.1.3 Compiler Directives**

The F (byte-flipping), G (no gotos), and S (segment swapping) compiler directives have no effect in version IV.0 and can be removed. Goto restrictions were a carry-over from the university and are no longer needed. User-controlled segment swapping is no longer necessary because the Compiler now handles swapping automatically.

Leaving these directives in your source code causes no harm at present. However, it is possible that in the future these letters will acquire new meanings as compiler directives, so the most prudent course is to remove them from your programs.

### **8.7.1.4 Compiling System-Level Programs**

Examples pertaining to the following discussion appear at the end of this section.

The outermost (Operating System) lexical level common to versions II and III no longer exists. The compile-time program directive U- sets the options R- and I- and allows units to be compiled with reserved System names. See the section below for details on version IV.0 units. However, these changes do not affect the lexical level of programs or units. These changes have the following effects on existing System-level programs.

The outermost dummy lexical level is invalid and must be removed. Because there is no distinction between a System and a user program, the segment procedure declaration for the System program in question must be replaced with a normal program declaration. The dummy parameters associated with the segment declaration are no longer necessary. Also, the dummy body at the end of existing System programs which corresponds to the old System lexical level must be removed.

## APPENDICES

Dummy segment procedure declarations are unnecessary and can be removed, because version IV.0 segment numbers are not System-wide resources. The scope of these declarations only extends to the enclosing program or unit. Failure to remove the dummy declarations does not affect the execution of a program, but causes an unnecessary increase in the size of its code file.

The version IV.0 System globals reside in the interface section of the Operating System's KERNEL unit. System-level programs which include the file GLOBALS.TEXT must now use the UNIT KERNEL. Because the version of the kernel unit contained in the standard SYSTEM.PASCAL does not contain an interface section, a separate code file containing the unit with its interface section is supplied.

The System-level variables and data type declarations in the kernel unit are almost identical to those of the older System globals. The only objects missing in version IV.0 are the variable DEBUGINFO in the System variables and the BUGSTATE and SEGTABLE fields in SYSCOM. All other variables and data types have the same identifier names.

Programs which use modified versions of GLOBALS.TEXT to access a subset of the old System globals can do so in version IV.0 by moving their own global declarations into a stubbed version of the kernel unit's interface section. This is done by declaring a kernel unit containing the appropriate declarations in its interface section and using it in the manner described below. This dummy kernel unit must be compiled with the U- option, and the unit name must be KERNEL. Care should be taken to ensure that the subset declarations correspond with the version IV.0 System globals.

Programs which require direct, as opposed to compiler-generated, accesses to Operating System procedures must explicitly use the Operating System unit containing the needed routines. This is done in a manner similar to the use of the kernel unit described below. A description of the Operating System unit names, interfaces, and file names can be found in the Internal Architecture Guide.

Programs which refer to the System globals to gain access to the screen control characters and data that reside in SYSCOM work correctly in version IV.0. However, the data within SYSCOM is currently also contained in the screen control unit, described in the UCSD p-System Utilities manual. The screen control unit will replace SYSCOM in the near future, so it is desirable to make the extra effort now to move user and System programs away from SYSCOM dependencies.

The following are examples of system level programs.

Before1

```
{ $U- }
program System_level;

{ $I GLOBALS.TEXT }

segment procedure II_style(dum1,dum2:integer);

segment procedure dummy2;
begin
end;
...

segment procedure dummy9;
begin
end;

segment procedure mysegment;
begin
...
end;

begin {II_style.}
...
mysegment;
...
end;

begin {dummy outerblock.}
end.
```

After1

## APPENDICES

In this example, KERNEL.CODE is the file containing the kernel unit's interface section.

```
program IV.0_style;  
uses {$U KERNEL.CODE} kernel;
```

```
segment procedure mysegment;  
begin  
...  
end;
```

```
begin {IV.0 style.}  
...  
mysegment;  
...  
end.
```

Before2

```
{$U-}  
program System_level;  
type myuserinfo = record  
stub: integer;  
end;  
var filler: array 0..6 of integer;  
userinfo: myuserinfo;  
  
segment procedure II_style(dum1,dum2:integer);  
  
segment procedure dummies2to9;  
begin  
end;  
  
segment procedure mysegment;  
begin  
...  
end;
```

```
begin {II_style.}
```

```
...
```

```
mysegment;
```

```
...
```

```
end;
```

```
begin {dummy outerblock.}
```

```
end.
```

After2

```
{ $U- }
```

```
program IV.0_style_1;
```

```
    unit kernel {dummy};
```

```
    interface
```

```
        type myuserinfo = record
```

```
stub: integer;
```

```
end;
```

```
var filler: array 0..6 of integer;
```

```
userinfo: myuserinfo;
```

```
implementation
```

```
end;
```

```
uses kernel;
```

```
segment procedure mysegment;
```

```
begin
```

```
...
```

```
end;
```

```
begin {IV.0_style_1.}
```

```
...
```

```
mysegment;
```

```
...
```

```
end.
```

## APPENDICES

### **8.7.1.5 Architectural Ramifications**

The physical in-memory relationship between parameters and declared variables has changed in version IV.0. Therefore programs which depend on the old architecture must be changed. The following is an example (courtesy of the version II Filer) of this problem.

```
procedure GetAddr (var MyVar: MyType);
    var TrickArray 0..0 of integer;
    AddressOfActualParameter: MyType;
begin
    {$R-}
    AddressOfActualParameter := TrickArray -1
    {$R+}
end;
```

This procedure could obtain the memory address of a variable of type MyType by making the assumption that local variables are allocated in memory immediately following the procedure's parameters. This assumption is true in version II but false in version IV.0. Programs containing usages of this type need to be modified.

### **8.7.1.6 Dummy Segment Procedures and the System Librarian**

In versions II and III you can create and maintain programs that are too large to be compiled at one time (due to memory constraints) by compiling each segment of the program separately. The tools used for this task are the LIBRARY utility, described in the UCSD p-System Utilities manual, and a collection of programs, each of which contains only the necessary variable declarations; a single segment procedure, and sufficient dummy segment procedure declarations to assign the correct segment number to the real code segment.

The situation described in the preceding paragraph does not apply in release IV.0 because the compiler now performs dynamic assignment of local segment numbers to a program. (Standard and System procedure calls get local segment numbers.) Now the replacement of a code segment by a dummy body may cause a different segment number to be assigned to the target segment procedure. There exists no simple method for determining the local segment number assigned to most segment procedures in a program containing multiple segment procedures, explicitly used units, and implicitly used Operating System units.

Therefore, using the LIBRARY utility to combine the separately compiled segment procedures does not produce executable code files. Version IV.0 presents an elegant solution to programs which have required this treatment in the past: modularize the program by splitting it into a collection of separately compilable version IV.0 units.

#### **8.7.1.7 Compiling Units**

Version IV.0 accepts the syntax for regular, separate, and intrinsic units as input, but maps them all into a single unit scheme. The Operating System unit names are reserved for System use only. The Compiler only allows the compilation of units with reserved names when the U- Compiler option is used.

##### **Reserved Unit Names**

COMPUNIT	CONCURRENCY
DEBUGGER	EXTRAHEAP
EXTRAIO	FILEOPS
GOTOXY	HEAPOPS
KERNEL	LONGOPS
OSUTIL	PASCALIO
REALOPS	SCREENOPS
SOFTOPS	STRINGOPS

Version IV.0 units must contain an interface and an implementation section even if one is empty. Intrinsic data units from version II.1 may require the insertion of the reserved word "implementation" before the "end" in order to compile successfully.

#### **8.7.1.9 Program Headings**

Contrary to past versions of the UCSD Pascal Compiler, program or unit headings (for example, Program stuff; or Unit stuff;) are mandatory. The Compiler gives an error message for programs lacking a heading.

#### **8.7.1.10 Standard Real-Valued Functions**

Version IV.0 does not require the statement "USES TRANSCENDENTALS;" when a program uses Standard real-valued functions such as SIN and COS. If this statement is present, it must be removed before you compile the program.

## APPENDICES

### **8.7.2 Converting Assembly Language Programs**

This section describes changes that must be made to assembly language programs in order to run them on the IV.0 System. Some of the changes are concerned with interfaces to the System; others affect version II and III programming practices.

#### **8.7.2.1 Macro Parameters and ASCII Strings**

Unlike in previous assemblers, macro parameters are not expanded within ASCII strings in the version IV.0 Assembler.

#### **8.7.2.2 Assembler Identifiers**

Two changes have occurred to assembly language identifiers in version IV.0. First, lower-case alphabetic characters are allowed in identifiers and, as in Pascal, they are internally mapped into their upper-case equivalents. Second, the underscore character "\_" is no longer significant in identifiers; this too is consistent with Pascal usage.

The following are examples of equivalent assembly language identifiers:

```
readloop  
Read_Loop  
READLOOP
```

#### **8.7.2.3 Pascal/Assembly Language Procedure Interface**

Byte-array variables (types STRING and PACKED ARRAY OF CHAR) passed as value parameters are handled differently in version IV.0. A two-word string descriptor is passed in place of the old one word pointer. Processing byte-array variables will require some extra assembly code.

The order and number of parameter words pushed on the stack prior to an assembly procedure/function call is different for version IV.0. The function return words are now below all parameters on the stack, rather than being on the top of stack. Assembly procedures have zero words of function return space on the stack, real-valued functions have four words of return space, and all other functions have one word of return space. As in previous versions, these words must be popped from the stack by the assembly routine before the function return value is pushed.



The TI-99/4 and TI-99/4A have two code pool areas. The main code pool is in VDP RAM and is memory mapped. The alternate code pool is in CPU RAM and is directly addressable. This is a problem if an assembly language routine is passed a string constant because it is difficult to determine which code pool to access to find the string. The solution is to have a dummy Pascal procedure or function, with the same parameters, call the assembly language procedure or function. This ensures that the parameters are in CPU RAM.

#### **8.7.2.4 Assembly Level Stack Manipulation**

Assembly routines which allocate memory above the hardware stack pointer for data space may require changes. In version IV.0, the code pool can be as close as 40 words to the hardware top of stack. Because assembly routines cannot determine the code pool's location, the routines must use the stack sparingly in order to prevent later System crashes.

#### **8.7.2.5 Radix Switch Characters**

The Assembler uses the same characters to indicate the radix of a number. Thus, source code for some versions may require changes. The two most significant changes are that binary integer constants are defined with the radix switch character "T" and octal integer constants are defined with the radix switch character "Q".

## 8.8 RESERVED WORDS

The following are the words reserved for use by Standard Pascal and UCSD Pascal, as well as predeclared identifiers and UCSD predeclared identifiers.

### 8.8.1 Standard Pascal Reserved Words

and	array	begin	case
const	div	do	downto
else	end	set	then
to	type	until	var
while	with		

### 8.8.2 UCSD Pascal Reserved Words

external	file	for	forward
function	goto	if	implementation
in	interface	label	mod
nil	not	of	or
packed	procedure	process	program
record	repeat	segment	

### 8.8.3 Standard Predeclared Identifiers

abs	arctan	atan	Boolean
char	chr	cos	eof
eoln	exp	false	get
input	integer	ln	maxint
new	odd	ord	output
[pack]	page	pred	put
read	readln	real	reset
rewrite	round	sin	sqr
sqr	succ	text	true
trunc	[unpack]	write	writeln

#### 8.8.4 UCSD Predeclared Identifiers

attach	blockread	blockwrite	close
concat	copy	delete	exit
fillchar	gotoxy	halt	insert
interactive	ioresult	keyboard	length
mark	memavail	memlock	memswap
moveleft	moveright	pos	processid
pwroften	release	scan	seek
semaphore	seminit	signal	sizeof
start	str	string	time
unitbusy	unitclear	unitread	unitstatus
unitwait	unitwrite	varavail	vardispose
varnew	wait		

## APPENDICES

### 8.9 ASSEMBLER SYNTAX ERRORS

The following are the syntax errors which may be issued by the Assembler.

- 1: Undefined label
- 2: Operand out of range
- 3: Must have procedure name
- 4: Number of parameters expected
- 5: Extra garbage on line
- 6: Input line over 80 characters
- 7: Not enough ifs
- 8: Must be declared in ASECT before use
- 9: Identifier previously declared
  
- 10: Improper format
- 11: EQU expected
- 12: Must EQU before use if not to a label
- 13: Macro identifier expected
- 14: Word addressed machine
- 15: Backward ORG not allowed
- 16: Identifier expected
- 17: Constant expected
- 18: Invalid structure
- 19: Extra special symbol
  
- 20: Branch too far
- 21: Variable not PC relative
- 22: Illegal macro parameter index
- 23: Not enough macro parameters
- 24: Operand not absolute
- 25: Illegal use of special symbols
- 26: Ill-formed expression
- 27: Not enough operands
- 28: Cannot handle this relative
- 29: Constant overflow

- 30: Illegal decimal constant
- 31: Illegal octal constant
- 32: Illegal binary constant
- 33: Invalid key word
- 34: Unexpected end of input--after macro
- 35: Include files must not be nested
- 36: Unexpected end of input
- 37: Bad place for an include file
- 38: Only labels & comments may occupy column one
- 39: Expected local label
  
- 40: Local label stack overflow
- 41: String constant must be on 1 line
- 42: String constant exceeds 80 characters
- 43: Illegal use of macro parameter
- 44: No local labels in ASECT
- 45: Expected key word
- 46: String expected
- 47: Bad block, parity error (CRC)
- 48: Bad unit number
- 49: Bad mode, illegal operation
  
- 50: Undefined hardware error
- 51: Lost unit, no longer on-line
- 52: Lost file, no longer in directory
- 53: Bad title, illegal file name
- 54: No room, insufficient space
- 55: No unit, no such volume on-line
- 56: No file, no such file on volume
- 57: Duplicate file
- 58: Not closed, attempt to open an open file
- 59: Not open, attempt to access a closed file
  
- 60: Bad format, error in reading real or integer
- 61: Nested macro definitions not allowed
- 62: '=' or '<>' expected
- 63: May not EQU to undefined labels
- 64: Must declare .ABSOLUTE before first .PROC

## APPENDICES

- 76: Illegal immediate operand
- 77: Index must be WR
- 78: Close paren ")" expected
- 79: Indirect and autoincr must be WR
  
- 80: Autoincr must be WR indirect
- 81: Comma "," expected
- 82: No operand allowed
- 83: Illegal map file

## 8.10 AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE (ASCII)

The following table gives the decimal, octal, and hexadecimal codes for the ASCII characters.

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

## APPENDICES

### 8.11 MUSICAL TONE FREQUENCIES

The following table gives the frequencies (rounded to integers) of four octaves of the tempered scale (one-half step between notes). While this list does not represent the entire range of tones that the computer can produce, it can be helpful in programming music.

<u>FREQUENCY</u>	<u>NOTE</u>	<u>FREQUENCY</u>	<u>NOTE</u>
110	A	440	A (above middle C)
117	A <sup>#</sup> , B <sup>b</sup>	466	A <sup>#</sup> , B <sup>b</sup>
123	B	494	B
131	C (low C)	523	C (high C)
139	C <sup>#</sup> , D <sup>b</sup>	554	C <sup>#</sup> , D <sup>b</sup>
147	D	587	D
156	D <sup>#</sup> , E <sup>b</sup>	622	D <sup>#</sup> , E <sup>b</sup>
165	E	659	E
175	F	698	F
185	F <sup>#</sup> , G <sup>b</sup>	740	F <sup>#</sup> , G <sup>b</sup>
196	G	784	G
208	G <sup>#</sup> , A <sup>b</sup>	831	G <sup>#</sup> , A <sup>b</sup>
220	A (below middle C)	880	A (above high C)
220	A (below middle C)	880	A (above high C)
233	A <sup>#</sup> , B <sup>b</sup>	932	A <sup>#</sup> , B <sup>b</sup>
247	B	988	B
262	C (middle C)	1047	C
277	C <sup>#</sup> , D <sup>b</sup>	1109	C <sup>#</sup> , D <sup>b</sup>
294	D	1175	D
311	D <sup>#</sup> , E <sup>b</sup>	1245	D <sup>#</sup> , E <sup>b</sup>
330	E	1319	E
349	F	1397	F
370	F <sup>#</sup> , G <sup>b</sup>	1480	F <sup>#</sup> , G <sup>b</sup>
392	G	1568	G
415	G <sup>#</sup> , A <sup>b</sup>	1661	G <sup>#</sup> , A <sup>b</sup>
440	A (above middle C)	1760	A



**8.12 COLOR CODES**

<u>COLOR</u>	<u>CODE</u>	<u>COLOR</u>	<u>CODE</u>
Transparent	0	Medium Red	8
Black	1	Light Red	9
Medium Green	2	Dark Yellow	10
Light Green	3	Light Yellow	11
Dark Blue	4	Dark Green	12
Light Blue	5	Magenta	13
Dark Red	6	Gray	14
Cyan	7	White	15

## APPENDICES

### 8.13 HIGH-RESOLUTION COLOR COMBINATIONS

The following color combinations produce the sharpest, clearest character resolution.

#### BEST

1, 7	Black on Cyan	1, 12	Black on Dark Green
1, 6	Black on Dark Red	1, 14	Black on Gray
1, 5	Black on Light Blue	1, 13	Black on Magenta
1, 2	Black on Medium Green	1, 8	Black on Medium Red
4, 7	Dark Blue on Cyan	4, 14	Dark Blue on Gray
4, 5	Dark Blue on Light Blue	4, 3	Dark Blue on Light Green
4, 13	Dark Blue on Magenta	4, 15	Dark Blue on White
12, 7	Dark Green on Cyan	12, 10	Dark Green on Dark Yellow
12, 13	Dark Green on Gray	12, 3	Dark Green on Light Green
12, 11	Dark Green on Light Yellow	12, 2	Dark Green on Medium Green
6, 14	Dark Red on Gray	6, 9	Dark Red on Light Red
6, 11	Dark Red on Light Yellow	13, 9	Magenta on Light Red
2, 11	Medium Green on Light Yellow	2, 15	Medium Green on White

#### SECOND BEST

1, 4	Black on Dark Blue	1, 10	Black on Dark Yellow
1, 3	Black on Light Green	1, 9	Black on Light Red
1, 11	Black on Light Yellow	12, 9	Dark Green on Light Red
12, 15	Dark Green on White	6, 15	Dark Red on White
5, 14	Light Blue on Gray	5, 3	Light Blue on Light Green
5, 15	Light Blue on White	3, 15	Light Green on White

#### THIRD BEST

1, 15	Black on White	4, 11	Dark Blue on Light Yellow
6, 8	Dark Red on Medium Red	3, 11	Light Green on Light Yellow
13, 14	Magenta on Gray	13, 15	Magenta on White
2, 10	Medium Green on Dark Yellow	2, 14	Medium Green on Gray
8, 14	Medium Red on Gray	8, 9	Medium Red on Light Red
8, 11	Medium Red on Light Yellow	8, 15	Medium Red on White
15, 6	White on Dark Red		

FOURTH BEST

7, 1	Cyan on Black	7, 15	Cyan on White
6, 1	Dark Red on Black	6, 3	Dark Red on Light Green
14, 15	Gray on White	5, 1	Light Blue on Black
3, 1	Light Green on Black	9, 1	Light Red on Black
9, 15	Light Red on White	13, 11	Magenta on Light Yellow
8, 3	Medium Red on Light Green	15, 5	White on Light Blue

## APPENDICES

### 8.14 MATHEMATICAL FUNCTIONS

The following mathematical functions can be determined as shown. PI is equal to 3.14159265... . The sign of a number is returned by the following function.

```
function SGN(x:real):integer;
begin
  if x>0 then SGN:=1
    else if x<0 then SGN:=-1
      else SGN:=0;
end;
```

<u>Function</u>	<u>Formula</u>
Tangent	$\text{SIN}(X)/\text{COS}(X)$
Secant	$1/\text{COS}(X)$
Cosecant	$1/\text{SIN}(X)$
Cotangent	$\text{COS}(X)/\text{SIN}(X)$
Inverse Sine	$\text{ATAN}(X/\text{SQR}(1-X*X))$ if $X < 1$
Inverse Cosine	$-\text{ATAN}(X/\text{SQR}(1-X*X)) + \text{PI}/2$ if $X < 1$
Inverse Secant	$\text{ATAN}(\text{SQR}(X*X-1)) + (\text{SGN}(X)-1)*\text{PI}/2$
Inverse Cosecant	$\text{ATAN}(1/\text{SQR}(X*X-1)) + (\text{SGN}(X)-1)*\text{PI}/2$
Inverse Cotangent	$\text{PI}/2 - \text{ATAN}(X)$ or $= \text{PI}/2 + \text{ATAN}(-X)$ if $X < 1$
Hyberbolic Sine	$(\text{EXP}(X) - \text{EXP}(-X))/2$
Hyberbolic Cosine	$(\text{EXP}(X) + \text{EXP}(-X))/2$
Hyperbolic Tangent	$-2*\text{EXP}(-X)/(\text{EXP}(X) + \text{EXP}(-X)) + 1$
Hyperbolic Secant	$2/(\text{EXP}(X) + \text{EXP}(-X))$
Hyperbolic Cosecant	$2/(\text{EXP}(X) - \text{EXP}(-X))$
Hyperbolic Cotangent	$2*\text{EXP}(-X)/(\text{EXP}(X) - \text{EXP}(-X)) + 1$
Inverse Hyperbolic Sine	$\text{LN}(X + \text{SQR}(X*X+1))$
Inverse Hyperbolic Cosine	$\text{LN}(X + \text{SQR}(X*X-1))$
Inverse Hyperbolic Tangent	$\text{LN}((1+X)/(1-X))/2$
Inverse Hyperbolic Secant	$\text{LN}((1+\text{SQR}(1-X*X))/X)$ if $X < 0$
Inverse Hyperbolic Cosecant	$\text{LN}((\text{SGN}(X)*\text{SQR}(X*X+1)+1)/X)$ if $X < 0$
Inverse Hyperbolic Cotangent	$\text{LN}((X+1)/(X-1))/2$

For example, the function to find the secant can be written as follows.

```
function sec(x:real):real;
begin
  sec := 1/cos(x);
end;
```

## 8.15 LIST OF SPEECH WORDS

The following is a list of all the letters, numbers, words, and phrases that can be accessed with SAY and GET SPEECH.

- (NEGATIVE)	+ (POSITIVE)	. (POINT)	0
1	2	3	4
5	6	7	8
9			
A (ay)	A1 (uh)	ABOUT	AFTER
AGAIN	ALL	AM	AN
AND	ANSWER	ANY	ARE
AS	ASSUME	AT	
B	BACK	BASE	BE
BETWEEN	BLACK	BLUE	BOTH
BOTTOM	BUT	BUY	BY
BYE			
C	CAN	CASSETTE	CENTER
CHECK	CHOICE	CLEAR	COLOR
COME	COMES	COMMA	COMMAND
COMPLETE	COMPLETED	COMPUTER	CONNECTED
CONSOLE	CORRECT	COURSE	CYAN
D	DATA	DECIDE	DEVICE
DID	DIFFERENT	DISKETTE	DO
DOES	DOING	DONE	DOUBLE
DOWN	DRAW	DRAWING	
E	EACH	EIGHT	EIGHTY
ELEVEN	ELSE	END	ENDS
ENTER	ERROR	EXACTLY	EYE
F	FIFTEEN	FIFTY	FIGURE
FIND	FINE	FINISH	FINISHED
FIRST	FIT	FIVE	FOR
FORTY	FOUR	FOURTEEN	FOURTH
FROM	FRONT		

## APPENDICES

G	GAMES	GET	GETTING
GIVE	GIVES	GO	GOES
GOING	GOOD	GOOD WORK	GOODBYE
GOT	GRAY	GREEN	GUESS
H	HAD	HAND	HANDHELD UNIT
HAS	HAVE	HEAD	HEAR
HELLO	HELP	HERE	HIGHER
HIT	HOME	HOW	HUNDRED
HURRY			
I	I WIN	IF	IN
INCH	INCHES	INSTRUCTION	INSTRUCTIONS
IS	IT		
J	JOYSTICK	JUST	
K	KEY	KEYBOARD	KNOW
L	LARGE	LARGER	LARGEST
LAST	LEARN	LEFT	LESS
LET	LIKE	LIKES	LINE
LOAD	LONG	LOOK	LOOKS
LOWER			
M	MADE	MAGENTA	MAKE
ME	MEAN	MEMORY	MESSAGE
MESSAGES	MIDDLE	MIGHT	MODULE
MORE	MOST	MOVE	MUST
N	NAME	NEAR	NEED
NEGATIVE	NEXT	NICE TRY	NINE
NINETY	NO	NOT	NOW
NUMBER			
O	OF	OFF	OH
ON	ONE	ONLY	OR
ORDER	OTHER	OUT	OVER

P  
PERIOD  
POINT  
PRINT  
PROGRAM

PART  
PLAY  
POSITION  
PRINTER  
PUT

PARTNER  
PLAYS  
POSITIVE  
PROBLEM  
PUTTING

PARTS  
PLEASE  
PRESS  
PROBLEMS

Q

R  
READY TO START  
REMEMBER  
ROUND

RANDOMLY  
RECORDER  
RETURN

READ (read)  
RED  
REWIND

READ1 (red)  
REFER  
RIGHT

S  
SAYS  
SEES  
SHAPE  
SHORTER  
SIX  
SMALLEST  
SPACE  
START  
SUPPOSED

SAID  
SCREEN  
SET  
SHAPES  
SHOULD  
SIXTY  
SO  
SPACES  
STEP  
SUPPOSED TO

SAVE  
SECOND  
SEVEN  
SHIFT  
SIDE  
SMALL  
SOME  
SPELL  
STOP  
SURE

SAY  
SEE  
SEVENTY  
SHORT  
SIDES  
SMALLER  
SORRY  
SQUARE  
SUM

T  
TEN  
THAT IS INCORRECT  
THEIR  
THEY  
THIRD  
THREE  
TO  
TOP  
TWELVE

TAKE  
TEXAS INSTRUMENTS  
THAT IS RIGHT  
THEN  
THING  
THIRTEEN  
THREW  
TOGETHER  
TRY  
TWENTY

TEEN  
THE (thee)  
THERE  
THINGS  
THIRTY  
THROUGH  
TONE  
TRY AGAIN  
TWO

TELL  
THAN THAT  
THE1 (thuh)  
THESE  
THINK  
THIS  
TIME  
TOO  
TURN  
TYPE

U  
UNTIL

UHOH  
UP

UNDER  
UPPER

UNDERSTAND  
USE

V

VARY

VERY

W

WAIT

WANT

WANTS

## APPENDICES

WAY  
WELL  
WHEN  
WHO  
WON  
WORKING

WE  
WERE  
WHERE  
WHY  
WORD  
WRITE

WEIGH  
WHAT  
WHICH  
WILL  
WORDS

WEIGHT  
WHAT WAS THAT  
WHITE  
WITH  
WORK

X

Y  
YOU

YELLOW  
YOU WIN

YES  
YOUR

YET

Z

ZERO



## **8.16 PROGRAM DEVELOPMENT WITH MULTI-DRIVE SYSTEMS**

Section 1 describes the use of the Pascal Compiler with a single-drive system. With a single drive, the Compiler diskette must be on-line during the entire process, which limits the size of the programs which you may compile. The following describe using the System with two or three drives.

### **8.16.1 Two-Drive System**

Two disk drives allow you much more flexibility than a single-drive system. To efficiently use two drives, place the Compiler and Editor programs on one diskette, and place that diskette in #5. Place the diskette that contains the Filer in #4. The source and object code that you create should be kept on the diskette in #4. This allows you to develop quite large programs, with the software needed always on line.

Once the development is complete, the source and object code files can be copied to an applications diskette and deleted from the diskette which contains the Filer.

### **8.16.2 Three-Drive System**

Three drives provide the most convenient and flexible development system. The Compiler and Editor should be placed on one diskette and placed in #5. The Filer diskette should be placed in #4. The source and object code of the program you are developing can then be put on the diskette in #9.

## SECTION 9: IN CASE OF DIFFICULTY

1. Be sure that the diskette you are using is the correct one. Use the L(dir (list directory) command in the Filer to check for the correct diskette or program.
2. Ensure that your Memory Expansion unit, P-Code peripheral, and Disk System are properly connected and turned on. Be certain that you have turned on all peripheral devices and have inserted the appropriate diskette before you turn on the computer.
3. If your program does not appear to be working correctly, end the session and remove the diskette from the disk drive. Reinsert the diskette, and follow the "Set-Up Instructions" carefully. If the program still does not appear to be working properly, remove the diskette from the disk drive, turn the computer and all peripherals off, wait 10 seconds, and turn them on again in the order described above. Then load the program again.
4. If you are having difficulty in operating your computer or are receiving error messages, refer to the "Maintenance and Service Information" and "Error Messages" appendices in your User's Reference Guide or UCSD p-System P-Code manual for additional help.
5. If you continue to have difficulty with your Texas Instruments computer or the UCSD p-System Pascal Compiler package, please contact the dealer from whom you purchased the unit or program for service directions.

## **THREE-MONTH LIMITED WARRANTY HOME COMPUTER SOFTWARE MEDIA**

Texas Instruments Incorporated extends this consumer warranty only to the original consumer purchaser.

### **WARRANTY COVERAGE**

This warranty covers the case components of the software package. The components include all cassette tapes, diskettes, plastics, containers, and all other hardware contained in this software package ("the Hardware"). This limited warranty does not extend to the programs contained in the software media and in the accompanying book materials ("the Programs").

The Hardware is warranted against malfunction due to defective materials or construction. **THIS WARRANTY IS VOID IF THE HARDWARE HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLIGENCE, IMPROPER SERVICE, OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIAL OR WORKMANSHIP.**

### **WARRANTY DURATION**

The Hardware is warranted for a period of three months from the date of original purchase by the consumer.

### **WARRANTY DISCLAIMERS**

**ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR LOSS OF USE OF THE PRODUCT OR OTHER INCIDENTAL OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER.**

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.

## **LEGAL REMEDIES**

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

## **PERFORMANCE BY TI UNDER WARRANTY**

During the three-month warranty period, defective Hardware will be replaced when it is returned postage prepaid to a Texas Instruments Service Facility listed below. The replacement Hardware will be warranted for a period of three months from the date of replacement. TI strongly recommends that you insure the Hardware for value prior to mailing.

## **TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES**

### **U. S. Residents:**

Texas Instruments Service Facility  
P. O. Box 2500  
Lubbock, Texas 79408

### **Canadian Residents only:**

Geophysical Services Incorporated  
41 Shelley Road  
Richmond Hill, Ontario, Canada L4C5G4

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

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

Texas Instruments Consumer Service  
831 South Douglas Street  
El Segundo, California 90245  
(213) 973-1803

## **IMPORTANT NOTICE OF DISCLAIMER REGARDING THE PROGRAMS**

The following should be read and understood before purchasing and/or using the software media.

TI does not warrant the Programs will be free from error or will meet the specific requirements of the consumer. The consumer assumes complete responsibility for any decisions made or actions taken based on information obtained using the Programs. Any statements made concerning the utility of the Programs are not to be construed as express or implied warranties.

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

**IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THE PROGRAMS AND THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE SOFTWARE MEDIA. MOREOVER, TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER BY ANY OTHER PARTY AGAINST THE USER OF THE PROGRAMS.**

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.

## INDEX

<b>A</b>			
Appendices	166	Compile-time options	158
Arctan	180	Concat	53
Arithmetic operations	40	Concurrency	178
Arrays	26	Concurrent processes	33, 109
Arrays, packed	41	Conditional compilation	164
ASCII codes	201	Converting assembly language	194
Assembler syntax errors	198	Converting programs to IV.0	186
Assembly language	194	Copy	54
Atan	180		
Attach	48		
<b>B</b>		<b>D</b>	
B option	159	D option	160
Beep	129	Del_snd_list	132
Blockread	49	Del_sprite	149
Blockwrite	50	Delete	55
Break	127	Device numbers	169
<b>C</b>		Differences between UCSD and Standard	175
C option	159	Differences between UCSD versions	182
Call_snd	135	Difficulty	212
Case statements	35, 179	Directives	187
Chain	51	Dispose	31
Chain_snd	137		
Character sets	122	<b>E</b>	
Chn_snd_chain	138	E option	160
Chr_default	117	End_snd	140
Close	52	Eof	21
Color codes	122, 203	Eoln	21
Color combinations	204	Error codes	62
Commandio.code	163	Errors, I/O	168
Comments	179	Errors, execution	167
Comments	35	Errors, syntax	170, 198
Comparisons	36, 40, 179	Example, string	19
Compilation, conditional	164	Exception	56
Compilation, separate	95	Exclusion, mutual	113
Compiler directives	187	Execution errors	167
Compiler use	156	Exit statements	36
		External routines	32

<b>F</b>					<b>J</b>	
Files	22				Joy	124
Files in records					Jump_snd	137
and arrays	26					
Files without a type	24				<b>K</b>	
Files, interactive	22				Keys, special	15
Files, random access	25				Kill_all_snd	141
Fillchar	57				Kill_snd	141
Functions, mathematical	206					
Functions,					<b>L</b>	
transcendental	46, 193				L option	161
					Length	63
<b>G</b>					Library	108, 192
Get	23				Linking	93, 107
Get_pattern	121				Long integers	39
Get_speech	154					
Get_sprite	153				<b>M</b>	
Gosub_snd	136				Make_snd_list	131
Goto	36, 179				Mark	31, 64
Gotoxy	58				Mathematical functions	206
					Memavail	65
<b>H</b>					Memlock	66
Halt	59				Memory allocation	31
Hardware needs	9				Memory limitations	46
Headings	45, 193				Memory management	94, 177
Heap use	186				Memswap	67
					Moveleft	68
<b>I</b>					Moveright	69
I option	160				Multi-drive systems	211
I/O errors	168				Musical tone frequencies	202
I/O intrinsics	21, 175				Mutual exclusion	113
Identifiers	194					
Insert	60				<b>N</b>	
Integers, long	39				New	31
Interactive files	22				Numbers, device	169
Internal Architecture					Numbers, random	125
Guide	11					
Ioresult	61					

## INDEX

P		Redirect	72
P option	162	Release	31, 73
Packed arrays	41	Reserved words	196
Packed records	43	Reset	28
Packed variables	41	Return_snd	137
Page	30	Rewrite	29
Parametric procedures		Rnd_int	126
and functions	44	Rnd_real	126
Past_sprite_coinc	153		
Periodic_noise	134	S	
Play_all_snd	141	Say	154
Play_snd	140	Scan	74
Pos	70	Screenops.code	163
Priority	111	Seek	75
Procedures and functions	47	Segments	31, 93, 98,
Processes	110	Semaphores	33, 112
Processes, concurrent	33, 109	Seminit	76
Processid	111, 115	Separate compilation	31, 95
Program conversion	186	Set_chr_color	122
Program development	211	Set_pattern	117
Program headings	45	Set_rnd	125
Programming tactics	96	Set_scr_color	124
Put	23	Set_screen	123
Pwroften	71	Set_snd	140
		Set_snd_flag	142
Q		Set_snd_tempo	141
Q option	162	Set_spr_attribute	148
		Set_spr_size	149
R		Set_sprite	146
R option	162	Set-up Instructions	12
Radix switch characters	195	Sets	45
Random access files	25	Signal	33, 77
Random numbers	125	Size limitations	46
Randomize	125	Sizeof	78
Read	20, 27	Snd_beat	143
Read_snd_chain	138	Snd_lst_offset	143
Read_snd_flag	142	Snd_note	132
Read_snd_list	139	Snd_tone	133
Readln	20, 27	Snd_volume	134
Records	26	Sound processing	129
Records, packed	43	Span	128



Special keys 15  
 Speech 154  
 Speech words 207  
 Sprite\_coinc 152  
 Sprites 144  
 Stacksize 111  
 Start 33, 79  
 Str 80  
 String example 19  
 String handling 175  
 Strings 18, 127  
 Support 117  
 Synchronization 114  
 Syntax errors 170, 198  
 System differences 182  
 System level program  
     compilation 187  
 System.Syntax 157

## T

T option 163  
 Tactics, programming 96  
 Texas Instruments units 116  
 Time 81  
 Transcendental functions 46, 193

## U

U option 163  
 UCSD System  
     differences 182  
 Underscore 194  
 Unitbusy 82  
 Unitclear 83  
 Unitread 84  
 Units 32, 100, 193  
 Units, Texas  
     Instruments 116  
 Unitstatus 86  
 Unitwait 87  
 Unitwrite 88  
 Upper\_case 128

Utility library 108

## V

Varavail 89  
 Vardispose 90  
 Variables, packed 41  
 Varnew 91

## W

Wait 33, 92  
 Warranty 213  
 White\_noise 133  
 Write 29  
 Write\_snd\_list 139  
 Writeln 29



---

## **ADDENDUM**

### ***UCSD p-System Compiler Manual***

---

In Section 3.37, UNITSTATUS, of the UCSD p-System Compiler manual, a description of the changes this procedure makes to the first four words of STATUS\_REC is given. With early versions of the P-Code peripheral, words two, three, and four always return certain values as follows.

- Word two:    The number of bytes per sector on the device is always returned as 256.  
Word three:   The number of sectors per track is always returned as 9.  
Word four:    The number of tracks is always returned as 40.

**TEXAS INSTRUMENTS**  
INCORPORATED

*Texas Instruments invented the integrated circuit,  
the microprocessor, and the microcomputer.  
Being first is our tradition.*