digital

industrial basic

OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8 OS/8

digital equipment corporation

DEC-S8-OSIBA-A-D

OS/8 Industrial BASIC

For additional copies, order No. DEC-S8-OSIBA-A-D
from Software Distribution Center, Digital Equipment
Corporation, Maynard, Mass.

digital equipment corporation • maynard. massachusetts

Printed October, 1973
Reprinted July, 1974

The HOW TO OBTAIN SOFTWARE INFORMATION page, located at the back of this document, explains the various services available to DIGITAL software users.

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

CONTENTS

CHAPTER 1

INTRODUCTION TO OS/8 INDUSTRIAL BASIC

NOTE

BASIC IS A REGISTERED TRADEMARK
OF THE TRUSTEES OF DARTMOUTH COLLEGE.

BASIC is an interactive programming language for a variety of applications. It is used in scientific and business environments to solve both simple and complex mathematical problems with a minimum of programming effort. It is used by educators and students as a problem-solving tool and as an aid to learning through programmed instruction and simulation.

In many respects the BASIC language is similar to other programming languages (such as FOCAL and FORTRAN), but BASIC is aimed at facilitating communication between the user and the computer. The BASIC user types in the computational procedure as a series of numbered statements, making use of common English words and familiar mathematical notations. Because of the small number of commands necessary and its easy application in solving problems, BASIC is one of the simplest computer languages to learn. With experience, the user can add the advanced techniques available in the language to perform more intricate manipulations or express a problem more efficiently and concisely.

OS/8 Industrial BASIC has such features as chaining, string manipulation, and file-oriented input/output. OS/8 Industrial BASIC has added features for support of time based, and external event driven segments, coded in BASIC, to service the user's real time data collection and response via DEC's UNIVERSAL DIGITAL CONTROLLER (UDC).

1.1 HARDWARE REQUIREMENT

The standard requirements for an OS/8 system must be met. In addition the only devices supported will be:

1. TD8-E with either ROM or 12K
   UDC with not more than 16 words I/O
   DK8-EC crystal clock jumpered for 50 hz

2. RK8-E and DECtape
   UDC with not more than 16 words I/O
   DK8-EC crystal clock jumpered for 50 hz

1.2 DOCUMENTATION CONVENTIONS

1.2.1 Underlining

Where clarification is required in the programming examples used in this manual, underlined copy denotes user input. Copy not underlined indicates entries typed by OS/8 Industrial BASIC.

1.2.2 Carriage return

A carriage return is always typed at the end of a line to indicate that the line is complete. The carriage return notation used in this manual is the symbol ↵ .

1.2.3 Blank spaces

Blank spaces are denoted by the symbol ␣ where clarification is required.

1.2.4 Control and shift characters

Control characters such as CTRL/O are typed by holding down the CTRL key and pressing the letter O. Shift characters such as SHIFT/L are typed by holding down the SHIFT key and pressing the letter L.

1.2.5 Terminals

The use of the word "terminal" throughout this manual implies a DECwriter, an ASR-33 Teletype, or an equivalent interactive device.

1.3 LOADING AND RUNNING OS/8 INDUSTRIAL BASIC

The following discussion assumes that the OS/8 operating system for the user's hardware configuration is on line. The OS/8 operating system is described in the OS/8 SYSTEM REFERENCE MANUAL. The following paragraphs are a condensation of the editing and control commands described in Chapter 9 of this manual.

1.3.1 Gaining Access to BASIC

Once the Keyboard Monitor has responded with a period to indicate that it is ready to receive a monitor command, the user types the following command:

```
.R INBSIC
```

BASIC responds with the following:

```
NEW OR OLD --
```

The user types in:

```
NEW FILE.EX
```

if the user is going to create a new program, where FILE.EX is the name and extension of the new program. If the user wants to work with a previously created program that he saved on a storage device, he types in the following:

```
OLD DEV:FILE.EX
```

where DEV: is the name of the OS/8 device his old file is stored on.

For example:

```
OLD DTA6:SAMPLE.BA
```

This response to NEW OR OLD -- retrieves the file named SAMPLE from DECtape and replaces the current contents of user core with the file SAMPLE. If you specify a device that does not exist or that is not available for your use, INDUSTRIAL BASIC returns an error message.

For further information regarding OS/8 files and devices, refer to Chapter 9 of this manual and to the OS/8 SYSTEM REFERENCE MANUAL.

1.3.2 Entering the New Program

After the user types in his filename, OS/8 Industrial BASIC responds with the following:

```
READY
```

The user can begin to type in his new program or make changes to his old program. When the new program is being typed, the user must make sure that each line begins with a line number containing no more than five digits and containing no spaces or nondigit characters. The RETURN key must be pressed at the completion of each line. If, in the process of typing a statement, the user makes a typing error and notices it before he terminates the line, he can correct it by pressing the RUBOUT key or SHIFT/O keys once for each character to be erased, going backward until the character in error is reached. Then he may continue typing, beginning with the character in error. Using the RUBOUT key or SHIFT/O keys echoes a backarrow (←) for each character deleted. The following is an example of this correcting process (note that a ← is typed for spaces as well as characters):

```
20 DEN F←←←F FNA(X,Y)=X↑2+3*Y
```

The corrected version of the above example would appear on a subsequent listing of the program as:

```
        20 DEF FNA(X,Y)=X↑2+3*Y
```

Program listings can be generated using the LIST or LISTNH commands.

1.3.3 Executing the Program

After typing the complete program (do not forget to end with an END statement), the user types RUN or RUNNH, followed by the RETURN key. OS/8 Industrial BASIC types the name of the program, the version of OS/8 Industrial BASIC, the current date (unless RUNNH is specified), and then it analyzes the program. If the program can be run, OS/8 Industrial BASIC executes it and, via PRINT statements, types out any results that were requested. The typeout of results does not guarantee that the program is correct (the results could be wrong), but it does indicate that no syntactical errors exist (e.g., missing line numbers, misspelled words, or illegal syntax). If errors of this type do exist, OS/8 Industrial BASIC types a message (or several messages) to the user. A list of these diagnostic messages, with their meanings, is given in Appendices B and C.

NOTE

> RUN and RUNNH are control commands, and like all other OS/8 Industrial BASIC edit and control commands, they do not require a line number preceding the command.

1.3.4 Correcting the Program

If the user receives an error message typeout informing him, for example, that line 60 is in error, the line can be corrected by typing in a new line 60 to replace the erroneous one. If the statement on line 110 is to be deleted from your program, it is accomplished by typing the following:

```
        110↵
```

If he wishes to insert a statement between lines 60 and 70, the user types a line number between 60 and 70 (e.g., 65), followed by the statement.

1.3.5 Interrupting Execution of the Program

If the results being typed out seem to be incorrect and he wants to stop execution of his program, the user types CTRL/C which is echoed by ↑C. The OS/8 Industrial BASIC editor responds with the READY message whereupon the user can modify or add statements and rerun his program.

1.3.6 Leaving the Computer

When the user's program is finished and he no longer requires the use of OS/8 Industrial BASIC, he types the BYE command (or CTRL/C to return control to the Keyboard Monitor.

1.3.7 Example of OS/8 Industrial BASIC Run

The following is a simple example of the use of OS/8 Industrial BASIC.

```
.R INBSIC                                 Instruct monitor to bring
                                          BASIC into core and start its
                                          execution

NEW OR OLD--NEW SAMPLE.BA                 BASIC asks whether new or old
                                          program is to be run

READY                                     BASIC is now ready to receive
                                          statements

10 FOR N=1 TO 7                           Type in statements
20 PRINT N, SQR(N)
30 NEXT N
40 PRINT "DONE"
50 END




RUN                                       Run program


SAMPLE   BA      1.0       22-SEP-73      Program heading and results of
                                          program are printed.
 1              1
 2              1.41421
 3              1.73205
 4              2
 5              2.23607
 6              2.44949
 7              2.64575
DONE

READY
```

1.4 OS/8 INDUSTRIAL BASIC OVERVIEW

The experienced BASIC programmer may elect to skip Chapters 2 and 4 through 6 of this manual since they are rather fundamental. However, he should familiarize himself with the remaining chapters and appendices as they provide information specifically related to OS/8 Industrial BASIC.

1.4.1 General System Description

The OS/8 Industrial BASIC system is divided into five discrete parts:

1. Editor
2. Compiler
3. Loader
4. Runtime System
5. Runtime System Overlays

The OS/8 Industrial BASIC Editor is used to create and edit the source program. On receipt of a RUN command, the Editor stores the program in a temporary file and chains to the Compiler. The Compiler compiles the program into pseudo-instructions which are then loaded into core with the Runtime System by the Loader. The Runtime System interprets each pseudo-instruction, calling each of the Overlays into core as needed. On completion of the program, the Runtime System chains back to the Editor. An OS/8 Industrial BASIC program consists of a mainline segment and user process interrupt routines. User process interrupt routines are executed in response to external events. A more complete description of the OS/8 Industrial BASIC System is provided in Chapter 11 of this manual.

1.4.2 OS/8 Industrial BASIC Statements and Commands

OS/8 Industrial BASIC consists of program statements and system control commands which are needed to write programs. A number the elementary OS/8 Industrial BASIC statements and commands are:

<u>OS/8 INDUSTRIAL BASIC STATEMENTS</u>

LET	Assign a value to a variable.
PRINT	Print out the indicated information.
READ	Initialize variables to values from the data list.

DATA	Provide initial data for a program.
GOTO	Change order of program execution.

IF GOTO
IF THEN
	Conditionally change order of program execution.

FOR TO STEP	Set up a program loop.
NEXT	End a program loop.
GOSUB	Go to a subroutine.
RETURN	Return from a subroutine.
INPUT	Get initial values from the terminal.
REM	Insert a program comment.
RESTORE	Restore the data list.
DEF	Define a function.
STOP	Stop program execution.
END	End a program.
DIM	Define subscripted variables.
UDEF	Define user-coded function.
TIMER CONTACT COUNTER	Associate interrupt service routine with external event.
DISMISS	Return from interrupt service routine.

OS/8 INDUSTRIAL BASIC EDIT AND CONTROL COMMANDS

LIST	List all stored program statements.
RUN	Run the currently stored program.
SCRATCH	Delete the currently stored program.
SAVE	Save the currently stored program.
OLD	Retrieve the old program.
NEW	Prepare for a new program.
NAME	Rename the currently stored program.
BYE	Exit from BASIC.

OS/8 Industrial BASIC may execute BASIC statements in either of two modes, mainline or user process interrupt. Mainline Mode is standard BASIC sequence. User Process Interrupt Mode is used to execute routines, similar to BASIC subroutines, that service external events.

These statements and commands are explained in detail with actual computer output in this manual. For the convenience of the user, a detailed OS/8 Industrial BASIC Statement, Command and Function Summary is included in Appendix A.

CHAPTER 2

OS/8 INDUSTRIAL BASIC ARITHMETIC

2.1 NUMBERS

An OS/8 Industrial BASIC number may be any number in the range of $10^{-616}<N<10^{616}$. OS/8 Industrial BASIC treats all numbers as decimal numbers -- that is, it accepts any number containing a decimal, and assumes a decimal point after an integer. The advantage of treating all numbers as decimal numbers is that the programmer can use any number or symbol in any mathematical expression without regard to its type.

In addition to integer and decimal formats, a third format is recognized and accepted by OS/8 Industrial BASIC and is used to express numbers outside the range $.00001<x<999999$. This format is called exponential or E-type notation and in this format, a number is expressed as a decimal number times some power of 10. The form is:

```
xxEn
```

where E represents "times 10 to the power of", thus the number is read: "xx times 10 to the power of n." For example:

```
23.4E2=23.4*10↑2=2340
```

Data may be input in any one or all three of these forms. Results of computations are output as decimals if they are within the range previously stated; otherwise, they are output in E format. OS/8 Industrial BASIC handles six significant digits in normal operation and input/output, as illustrated below:

Value Typed In	Value Output By OS/8 Industrial BASIC
.01	0.0099999
.0099	0.0099
999999	999999
1000000	.100000E+007
.0000009	.899999E-006

OS/8 Industrial BASIC automatically suppresses the printing of leading and trailing zeros in integer numbers and all but one leading zero in decimal numbers. As can be seen from the preceding examples, OS/8 Industrial BASIC formats all exponential numbers in the form:

```
sign .xxxxxxE(+or-)n
```

where x represents the number carried to six decimal places, E stands for "times 10 to the power of," and n represents the exponential value. For example:

```
-.347021E+009 is equal to -347,021,000
 .726000E-003 is equal to 0.000726
```

2.2 VARIABLES

A simple variable in OS/8 Industrial BASIC is an algebraic symbol representing a number, and is formed by a single letter or a letter followed by a digit. For example:

Acceptable Variables	Unacceptable Variables
I	2C - a digit cannot begin a variable
B3	AB - two or more letters cannot form a variable
X	

The user may assign values to variables either by indicating the values in a LET statement, or by inputting the values as data.

```
10 LET I=53721
20 LET B3=456.9
30 LET X=20E9
40 INPUT Q
```

These operations, as well as subscripted variables, are discussed in detail in Chapter 5. A discussion of subscripted and unsubscripted string variables is provided in Chapter 7.

The meaning of the = sign should be clarified. In algebraic notation, the formula X=X+1 is meaningless. However, in OS/8 Industrial BASIC (and most computer languages), the equal sign designates replacement rather than equality. Thus, this formula is actually translated "add one to the current value of X and store the new result back in the same variable X". Whatever values had previously been assigned to X will be combined with the value 1. An expression such as A=B+C instructs the computer to add the values of B and C and store the result in a third variable A. The variable A is not being evaluated in terms of any previously assigned value, but only in terms of B and C. Therefore, if A has been assigned any value prior to its use in this statement, the old value is lost; it is instead replaced by the value of B+C.

2.3 ARITHMETIC OPERATIONS

OS/8 Industrial BASIC performs addition, subtraction, multiplication, division and exponentiation, as well as more complicated operations explained in detail later in the manual. The five operators used in writing most formulas are:

Symbol Operator	Meaning	Example
+	Addition	A + B
-	Subtraction	A - B
*	Multiplication	A * B
/	Division	A / B
↑(or **)	Exponentiation (Raise A to the B Power)	A ↑ B or (A**B)

2.3.1 Priority of Arithmetic Operations

In any given mathematical formula, OS/8 Industrial BASIC performs the arithmetic operations in the following order:

1. Parentheses receive top priority. Any expression within parentheses is evaluated before an unparenthesized expression.

2. In absence of parentheses, the order of priority is :

   a. Exponentiation

   b. Multiplication and Division (of equal priority)

   c. Addition and Subtraction (of equal priority)

3. If either 1 or 2 above does not clearly designate the order of priority, then the evaluation of expressions proceeds from left to right.

The expression A/B*C is evaluated from left to right as follows:

```
1.  A/B                     = step 1

2.  (result of step 1)*C    = answer
```

2.3.2 Parentheses

Parentheses may be used by the programmer to change the order of priority (as listed in rule 2 above), because expressions within parentheses are always evaluated first. Thus, by enclosing expressions appropriately, the programmer can control the order of evaluation. Parentheses may be nested, or enclosed by a second set (or more) of parentheses. In this case, the expression within the innermost parentheses is evaluated first, and then the next innermost, and so on, until all have been evaluated.

Consider the following example:

```
A=7*((B↑2+4)/X)
```

The order of priority is:

```
1.  B↑2                    = step 1

2.  (result of step 1)+4   = step 2

3.  (result of step 2)/X   = step 3

4.  (result of step 3)*7   = A
```

Parentheses also prevent any confusion or doubt as to how the expression is evaluated. For example:

```
A*B↑2/7+B/C+D↑2

((A*B↑2)/7)+((B/C)+D↑2)
```

Both of these formulas will be executed in the same way. However, the inexperienced programmer or student may find that the second is easier to understand.

Spaces may also be used to increase readability. Since the OS/8 Industrial BASIC compiler ignores spaces, the two statements:

```
10 LET B = D↑2 + 1

10 LETB=D↑2+1
```

are identical, but spaces in the first statement provide ease in reading.

2.3.3 Relational Operators

A program may require that two values be compared at some point to discover their relation to one another. To accomplish this, OS/8 Industrial BASIC makes use of the following relational operators:

=	equal to
<	less than
>	greater than
=< or <=	less than or equal to
=> or >=	greater than or equal to
>< or <>	not equal to

Depending upon the result of the comparison, control of program execution may be directed to another part of the program. Relational operators are used in conjunction with the IF-THEN statement which is discussed in Chapter 3.

2.3.4 Rules for Exponentiation

The following rules apply in evaluating the expression A↑B.

	Rule	Example
1.	If B=0, then A↑B=1	3↑0=1
2.	If A=0 and B>0, then A↑B=0	0↑2=0
3.	If A=0 and B<0, then A↑B=0 and a DV error message is printed (See Appendix C).	0↑-2=0
4.	If B is an integer> 0, then A↑B=A *A *A ...*A , where n=B.	3↑5=3*3*3*3*3=243
5.	If B is an integer <0 then A↑B=1/(A *A *A ...*A), where n=B	3↑-5=1/243
6.	If B is a decimal (non-integer) and A>0, then A↑B=EXP(B*LOG(A))	2↑3.6=12.1257
7.	If B is a positive or negative decimal (non-integer) and A<0, program aborts due to fatal error.	-3↑2.6 is illegal. Fatal error message EM printed.

CHAPTER 3

OS/8 INDUSTRIAL BASIC STATEMENTS

The following Example Program is included at this point as an illustration of the format of an OS/8 Industrial BASIC program, the ease in running it, and the type of output that may be produced. This program and its results are for the most part self-explanatory. Following sections and chapters cover the program statements and system commands used in OS/8 Industrial BASIC programming.

```
10 REM  - PROGRAM TO TAKE AVERAGE OF
15 REM  - STUDENT GRADES AND CLASS GRADES
20 PRINT "HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT";
30 INPUT A,B
40 LET I=0
50 FOR J=I TO A-1
55 LET V=0
60 PRINT "STUDENT NUMBER = ";J
75 PRINT "ENTER GRADES"
76 LET D=J
80 FOR K=D TO D+(B-1)
81 INPUT G
82 LET V=V+G
85 NEXT K
90 LET V=V/B
95 PRINT "AVERAGE GRADE =";V
96 PRINT
99 LET Q=Q+V
100 NEXT J
101 PRINT
102 PRINT
103 PRINT "CLASS AVERAGE =";Q/A
104 STOP
105 END

READY
RUNNH
HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT?5,4
STUDENT NUMBER =  0
ENTER GRADES
?78
?86
?88
?74
AVERAGE GRADE = 81.5

STUDENT NUMBER =  1
ENTER GRADES
?59
?86
?70
?87
AVERAGE GRADE = 75.5
```

```
STUDENT NUMBER =  2
ENTER GRADES
?58
?64
?75
?80
AVERAGE GRADE = 69.25

STUDENT NUMBER =  3
ENTER GRADES
?88
?92
?85
?79
AVERAGE GRADE = 86

STUDENT NUMBER =  4
ENTER GRADES
?60
?78
?85
?80
AVERAGE GRADE = 75.75


CLASS AVERAGE = 77.6
```

3.1 STATEMENT NUMBERS

A program is made up of statements. Each line of the program begins with a line number of 1 to 5 digits that serves to identify the line as a statement. The largest allowable line number is 99999. Line numbers serve to specify the order in which these statements are to be performed. Before the program is run, OS/8 Industrial BASIC sorts out and edits the program, putting the statements into the order specified by their line numbers; thus, the program statements can be typed in any order, as long as each statement is prefixed with a line number indicating its proper sequence in the order of execution. Each statement starts after its line number with an English word (except the LET statement where 'LET' is optional) which denotes the type of statement. Unlike program statements, system commands are not preceded by line numbers and are executed immediately after they are typed in. (Refer to Chapter 9 for a further description of commands.) Spaces have no significance in BASIC programs or commands, except in messages or literal strings which are printed out, and in line numbers. Thus, spaces may be used to modify a program and make it more readable.

A common programming practice is to number lines by fives or tens, so that additional lines may be inserted in a program without the necessity of renumbering lines already present. Renumbering a program can be accomplished by using the RESEQ program described in Chapter 9, section 9.1.2.

Multiple statements may be placed on a single line by separating each statement from the preceding statement with a backslash (SHIFT/L). For example:

```
10 A=5\B=.2\C=3\PRINT "ENTER DATA"
```

All of the statements in line 10 will be executed before BASIC continues to the next line. Only one statement number at the beginning of the entire line is necessary. However, it should be remembered that program control cannot be transferred to a statement within a line, but only to the first statement of the line in which it is contained.

NOTE

User process interrupt routines will be entered only when the mainline encounters a line number. I.e.,

```
100   FOR I=1 TO 10000\NEXT I
```

does not allow user process interrupt routines to be entered while executing this line because the entire loop does not contain line numbers.

```
100   FOR I=1 TO 10000
105   NEXT I
```

will allow user process interrupt routines to be executed while processing the loop.

3.2 REMARK -- THE COMMENTING STATEMENT

The REM or REMARK statement allows the programmer to insert comments or remarks into a program without these comments affecting execution. The OS/8 Industrial BASIC compiler ignores everything between REM and the end of the line. The form is:

```
(line number) REM (message)
```

In the Example Program, lines 10 and 15 are REMARK statements describing what the program does. It is often useful to put the name of the program and information relating to its use at the beginning where it is available for future reference. Remarks throughout the body of a long program will help subsequent debugging by explaining the purpose of each statement within the program.

3.3 STATEMENTS FOR TERMINATING A PROGRAM

3.3.1 END

The END statement (line 140 in the Example Program) should be the last statement of the entire program. The form is:

```
(line number) END
```

NOTE

An END statement must be the last statement in the program. A program is terminated when an END statement is executed. In real time operations a program should never execute either an END or a STOP.

3.3.2 STOP

The STOP statement is used synonymously with the END statement to terminate execution; but while END occurs only once at the end of a program, STOP may occur any number of times. The format of the STOP statement is:

```
(line number) STOP
```

This statement signals that execution is to be terminated at that point in the program where it is encountered.

3.4 LET -- THE ASSIGNMENT STATEMENT

The Assignment (LET) statement is probably the most commonly used OS/8 Industrial BASIC statement and is used whenever a value is to be assigned to a variable. It is of the form:

```
(line number) LET x = expression
```

where x represents a variable, and the expression is either a number, another variable, or an arithmetic expression. The word "LET" is optional; thus the following statements are treated the same:

```
100   LET A=A↑B+10  110   LET L=L+1

100   A=A↑B+10      110   L=L+1
```

The LET statement is not strictly an equality. LET means "evaluate the expression to the right of the equal sign and assign this value to the variable on the left." Thus, the statement L=L+1 means "set L equal to a value one greater than it was before."

3.5 INPUT/OUTPUT STATEMENTS AND FUNCTIONS

Input/output statements allow the user to bring data into a program and output results or data at any time during execution.

3.5.1 The INPUT Statement

The INPUT statement is used when data is to be supplied by the user from the terminal keyboard while a program is executing and is of the form:

```
          (line number) INPUT x1, x2,...,xn
```

where x1 through xn represent variable names. For example:

```
          25    INPUT A
```

When this statement occurs in Mainline Mode, the user will be prompted with a question mark and while waiting for input, user process interrupt service routines may execute. If it occurs in the user process interrupt service routine, the user will be prompted with an exclamation mark followed by a question mark and the system will wait for input completion before processing any other user process interrupt routine.

NOTE

> If an input statement occurs in the user process interrupt service routine while the user is entering input in the Mainline Mode, the text is lost to the last terminator. After the request is fulfilled for the user process interrupt service routine the mainline request will be reissued.

The following rules apply to the use of the INPUT statement.

Rule

1. The following characters are recognized as acceptable when inputting numeric data:

   + or - sign
   digits 0 through 9
   the letter E
   leading spaces (ignored)
   . (first decimal point)

   Terminators are carriage return or comma.

   All other characters cause the remaining characters in the field to be ignored.

```
          10    INPUT A,B,C,D,E
          .
          .
          .
```

```
RUNNH

?10,32A16,8 1
?5,6
READY
```

In the above example, A=10, B=32, C=8, D=5, and E=6.

2. When inputting numeric data, two terminators read in succession imply that the data between the terminators is 0.

```
10   INPUT A,B,C,D,E
.
.
.

RUNNH

?5,10,,12,15

READY
```

   In the above example A=5, B=10, C=0, D=12, and E=15.

3. In response to an INPUT statement the user can provide more data than is requested by the INPUT statement. The remaining or unused data is saved for subsequent use by the next INPUT statement. The question mark (?) is not printed until the program is out of data.

4. When inputting string data, all characters, except terminators, are recognized as part of the string. See Chapter 7 for further information relating to strings.

5. A line feed is recognized as part of the string data, and therefore is stored in the text buffer.

3.5.2 The PRINT Statement

3.5.2.1 General

The PRINT statement is used to output results of computations, comments, values of variables, or plot points of a graph on the terminal. The format is:

```
(line number) PRINT expression
```

When used without an expression, a blank line will be output on the terminal. For more complicated uses, the type of expression and the type of format control characters (comma or semicolon) following the word PRINT determines which formats will be created.

In order to have the computer print out the results of a computation, or the value of a variable at any point in the program, the user types the line number, PRINT, and the variable name(s) separated by a format control character, in this case, commas:

```
5   A=16\B=5\C=4
10   PRINT A,C+B,SQR(A)
15   END
```

The PRINT statement may also be used to output a message or line of text. The desired message is simply placed in quotation marks in the PRINT statement as follows:

```
10   PRINT "THIS IS A TEST"
```

When line 10 is encountered during execution, the following will be printed:

```
THIS IS A TEST
```

A message may be combined with the result of a calculation or a variable as follows:

```
80   PRINT "AMOUNT PER PAYMENT =";R
```

Assuming R=344.961 when line 80 is encountered during execution, this will be output as:

```
AMOUNT PER PAYMENT = 344.961
```

THE PRINT statement can also cause a constant to be printed on the console. For example:

```
10   PRINT 1.234,SQR(10014)
```

will cause the following to be output at execution time:

```
1.234        100.07
```

Any algebraic expression in a PRINT statement will be evaluated using the current value of the variables. Numbers will be printed according to the format specified in Chapter 2 and in paragraph 3.5.2.3.

3.5.2.2 Format Control Characters

In OS/8 Industrial BASIC, a terminal line is formatted into five fixed zones (called print zones) of 14 columns each. A program such as:

```
5   A=2.3\B=21\C=156.75\D=1.134\E=23.4
10   PRINT A,B,C,D,E
15   END
```

where the control character comma (,) is used to separate the variables in the PRINT statement, will cause the values of the variables to be printed using all five zones.

```
RUNNH
 2.3           21            156.75        1.134         23.4
```

```
|← 14 columns|← 14 columns|← 14 columns|← 14 columns|← 14 columns

READY
```

It is not necessary to use the standard five zone format for output. The control character semicolon (;) causes the text or data to be output immediately after the last character printed.

The following example program illustrates the use of the control characters in PRINT statements

```
10 READ A,B,C
20 PRINT A,B,C,A↑2,B↑2,C↑2
30 PRINT
40 PRINT A;B;C;A↑2;B↑2;C↑2
50 DATA 4,5,6
60 END

READY
RUNNH
 4             5             6             16            25
 36

 4  5  6  16  25  36

READY
```

As this example illustrates, when more than five variables are listed in the PRINT statement, OS/8 Industrial BASIC automatically moves the sixth number to the beginning of the next line.

3.5.2.3 Printing Numbers

For any format (integer, decimal, or E-type) OS/8 Industrial BASIC prints numbers in the form:

sign number space

where sign is either minus (-) or blank (for plus) and a blank space always trails the number.

```
READY
10 A=64\B=-32\C=72
20 PRINT A;B;C
21 END
RUNNH
 64 -32  72
```

3.5.2.4 PRINT Used With INPUT

Another use of the PRINT statement is to combine it with an INPUT statement so as to identify the data expected to be entered. As an example, consider the following program:

```
READY

10 REM - PROGRAM TO COMPUTE INTEREST PAYMENTS
20 PRINT "INTEREST IN PERCENT";
25 INPUT J
26 LET J=J/100
30 PRINT "AMOUNT OF LOAN";
35 INPUT A
40 PRINT "NUMBER OF YEARS";
45 INPUT N
50 PRINT "NUMBER OF PAYMENTS PER YEAR";
55 INPUT M
60 LET N=N*M
65 LET I=J/M
70 LET B=1+I
75 LET R=A*I/(1-1/B↑N)
78 PRINT
80 PRINT "AMOUNT PER PAYMENT =";R
85 PRINT "TOTAL INTEREST     =";R*N-A
90 LET B=A
95 PRINT " INTEREST      APP TO PRIN    BALANCE"
100 LET L=B*I
110 LET P=R-L
120 LET B=B-P
130 PRINT L,P,B
140 IF B>=R GO TO 100
150 PRINT B*I,R-B*I
160 PRINT "LAST PAYMENT =";B*I+B
200 END

READY
RUNNH
INTEREST IN PERCENT?9
AMOUNT OF LOAN?2500
NUMBER OF YEARS?2
NUMBER OF PAYMENTS PER YEAR?4

AMOUNT PER PAYMENT = 344.965
TOTAL INTEREST     = 259.724
 INTEREST      APP TO PRIN    BALANCE
 56.25          288.715        2211.28
 49.7539        295.212        1916.07
 43.1116        301.854        1614.22
 36.3199        308.645        1305.57
 29.3754        315.59         989.982
 22.2746        322.691        667.291
 15.0141        329.951        337.34
 7.59015        337.375
LAST PAYMENT = 344.93

READY
```

As can be noticed in this example, the question mark is grammatically useful in a program in which several values are to be input by allowing the programmer to formulate a verbal question which the input value will answer.

3.5.3 The TAB(X) Function

The TAB function, which may only be used in a PRINT statement, allows the user to position the printing of characters anywhere on the terminal line (or other printing device line when used with PRINT#, see Chapter 10). Print positions can be thought of as being numbered from 1 to 72 across the terminal from left to right. The form of this function is:

```
TAB(X)
```

where the argument X represents the position (from 1 to 72 columns available on the terminal) in which the next character will be typed.

Each time the TAB function is used in a PRINT statement, positions are counted from the beginning of the line, not from the current position of the printing head. For example, the TAB function in the following program causes the character "/" to be printed at 24 equally spaced positions along the line.

```
10   FOR K=3 TO 72 STEP 3
20   PRINT TAB(K);"/";
30   NEXT K
40   END
```

If the argument X in the TAB function is less than the current position of the printing head, printing is started at the current position. If the argument X is greater than 72 (the number of columns available in an output line), a carriage return-line feed is executed and printing resumes at position 1.

3.5.4 The PNT(X) Function

OS/8 Industrial BASIC provides an additional function, PNT(X), to increase input/output flexibility. The function is primarily used for outputting non-printing characters such as the "bell", but can be used for more sophisticated applications. The PNT(X) function, like the TAB(X) function, may only be used in either a PRINT or PRINT# statement. The form of the function is:

```
PNT(X)
```

where the argument X represents the decimal value of the 7-bit ASCII character to be output. For example, the statement:

```
10   PRINT "X=";3.14159;PNT(13);TAB(14);"/"
```

will print the slash (/) on top of the equal sign after executing a carriage return (CR=13) and a TAB to column 2 as shown below:

```
          X ≠ 3.14159
```

Notice that a TAB(14) is required since OS/8 Industrial BASIC remembers the print head to be at column 12 after the carriage return (11 columns for X= 3.14159 and 1 column for the PNT function). A tab to column 2 after the carriage return provides a total of 14 columns. The PNT(13) carriage return does not zero the column count but, in fact, adds to the column count. (This example may not work on some terminals.)

3.6 THE READ AND DATA STATEMENTS

READ and DATA statements are used to provide data to a program. One statement is never used without the other. The form of the READ statement is:

```
          (line number) READ x1,x2,...,xn
```

where x1 through xn represent variable names. For example:

```
          10    READ A,B,C
```

A, B, and C are variables to which values will be assigned. Variables in a READ statement must be separated by commas. READ statements are generally placed at the beginning of a program, but must at least logically occur before that point in the program where the value is required for some computation.

Values which will be assigned to the variables in a READ statement are supplied in a DATA statement of the form:

```
          (line number) DATA x1,x2,...,xn
```

where x1 through xn represent values. The values must be separated by commas and occur in the same order as the variables which are listed in the corresponding READ statement. A DATA statement appropriate for the preceding READ statement is:

```
          70    DATA 1,2,3
```

Thus, at execution time A=1, B=2, and C=3.

The DATA statement is usually placed at the end of a program (before the END statement) where it is easily accessible to the programmer should he wish to change the values.

A READ statement may have more or fewer variables than there are values in any one DATA statement. The READ statement causes OS/8 Industrial BASIC to search all available DATA statements in the order of their line numbers until values are found for each variable in the READ. A second READ statement will begin reading values where the first stopped. If at some point in the program an attempt is made to read data which is not present, OS/8 Industrial BASIC will stop and print the following message at the console:

```
DA AT LINE YYYYY
```

where YYYYY indicates the line which caused the error.

3.7 RESTORE

If it is desired to use the same data more than once in a program, the RESTORE statement will make it possible to recycle through the DATA list beginning with the first DATA statement. The RESTORE statement is of the form:

```
(line number) RESTORE
```

An example of its use follows:

```
15   READ B,C,D
.
.
.
55   RESTORE
60   READ E,F,G
.
.
.
80   DATA 6,3,4,7,9,2
.
.
.
100   END
```

The READ statements in lines 15 and 60 will both read the first three data values provided in line 80. (If the RESTORE statement had not been inserted before line 60, then the second READ would pick up data in line 80 starting with the fourth value.)

The programmer may, if he chooses to do so, use the same variable names the second time through the data, since the values are being read as though for the first time. In order to skip unwanted values, the programmer may insert replacement, or dummy variables. Consider:

```
1 REM - PROGRAM TO ILLUSTRATE USE OF RESTORE
20 READ N
25 PRINT "VALUES OF X ARE:"
30 FOR I=1 TO N
40 READ X
50 PRINT X,
60 NEXT I
70 RESTORE
185 PRINT
190 PRINT "SECOND LIST OF X VALUES"
200 PRINT "FOLLOWING RESTORE STATEMENT:"
210 FOR I=1 TO N
```

```
220 READ X
230 PRINT X,
240 NEXT I
250 DATA 4,1,2
251 DATA 3,4
300 END

READY
RUNNH
VALUES OF X ARE:
 1             2             3             4
SECOND LIST OF X VALUES
FOLLOWING RESTORE STATEMENT:
 4             1             2             3
READY
```

The second time the data values are read, the first X picks up the value originally assigned to N in line 20, and as a result, OS/8 Industrial BASIC prints:

```
4             1             2             3
```

To circumvent this, the programmer could insert a dummy variable which would pick up and store the first value, but would not be represented in the PRINT statement, in which case the output would be the same each time through the list.

3.8 CONTROL STATEMENTS

Certain control statements cause the execution of a program to jump to a different line either unconditionally or depending upon some condition within the program. The following statements give the programmer capabilities in this area.

3.8.1 GOTO

The GOTO (or GO TO) statement is an unconditional statement used to direct program control either forward or backward in a program. The form of the GOTO statement is:

```
(line number) GOTO n
```

where n represents a statement number. When the logic of the program reaches the GOTO statement, the statement(s) immediately following will not be executed; instead execution is transferred to the statement beginning with the line number indicated.

The following program never reaches the END statement; it does a READ, prints something, and jumps back to the READ via a GOTO statement. It attempts to do this over and over until it runs out of data, which is sometimes an acceptable, though not advisable, way to end a program.

```
READY

10 REM - PROGRAM ENDING WITH ERROR
11 REM - MESSAGE WHEN OUT OF DATA
20 READ X
25 PRINT "X=";X,"X↑2=";X↑2
30 GO TO 20
35 DATA 1,5,10,15,20,25
40 END

READY
RUNNH
X= 1           X↑2= 1
X= 5           X↑2= 25
X= 10          X↑2= 100
X= 15          X↑2= 225
X= 20          X↑2= 400
X= 25          X↑2= 625

DA AT LINE 00020

READY
```

3.8.2 IF-THEN and IF-GOTO

If a program requires that two values be compared at some point, control of program execution may be directed to different procedures depending upon the result of the comparison. In computing, values are logically tested to see whether they are equal to, greater than, or less than another value, or possibly a combination of the three. This is accomplished by use of the relational operators discussed in Chapter 2.

IF-THEN and IF-GOTO statements allow the programmer to test the relationship between two variables, numbers, or expressions. Providing the relationship described in the IF statement is true at the point it is tested, control will transfer to the line number specified. If the relationship described in the IF statement is not true at the point it is tested, control will transfer to the line following the IF statement. The statements are of the form:

```
                                    GOTO
(line number)IF v1 <relation> v2    or     x
                                    THEN
```

where v1 and v2 represent variable names, numbers, or expressions, and x represents a line number. The use of either THEN or GOTO is acceptable.

If the following example, the value of the variable A is changed or remains the same depending on A's relation to B.

```
100   IF A>B THEN 120
110   A=A↑B-1
120   C=A/D
```

NOTE

When using non-integer arithmetic in the IF-THEN statement, the test for equality may not always be appropriate due to the nature of the floating-point arithmetic used by the computer. To avoid this problem, the programmer should either avoid using non-integer arithmetic in fractional values less than the tolerance desired. IF-THEN statements that test the running variable in FOR-NEXT loops (see Chapter 4) are particularly sensitive to this problem. For example:

```
10   FOR A=-5 TO 5 STEP .1
20   IF A=0 THEN 50
30   NEXT A
40   STOP
50   PRINT "EQUAL TO ZERO"
60   END
```

The above margin will never go to line 50.

CHAPTER 4

LOOPS

Frequently programmers are interested in writing a program in which one or more portions are executed a number of times, usually with slight variations each time. To write the simplest program in which the portion of the program to be repeated is written just once, a loop is used. A loop is a block of instructions that the computer executes repeatedly until a specified terminal condition is met. OS/8 Industrial BASIC provides two statements to specify a loop: FOR and NEXT.

4.1 FOR AND NEXT STATEMENTS

The FOR statement is of the form:

```
          (line number) FOR v=x1 TO x2 STEP x3
```

where v represents a variable name, and x1, x2, and x3 all represent expressions (a numerical value, variable name, or mathematical expression). v is termed the index, x1 the initial value, x2 the terminal value, and x3 the incremental value. For example:

```
          15 FOR K=2 TO 20 STEP 2
```

This means that the loop will be repeated as long as K is less than or equal to 20. Each time through the loop, K is incremented by 2, so the loop will be executed a total of 10 times.

A variable used as an index in a FOR statement must not be subscripted, although a common use of loops is to deal with subscripted variables, using the value of the index as the subscript of a previously defined variable (this is illustrated in Chapter 5, section 5.1 concerning Subscripted Variables).

The NEXT statement is of the form:

```
          (line number) NEXT v
```

and signals the end of the loop. When execution of the loop reaches the NEXT statement, the computer adds the STEP value to the index and checks to see if the index is less than or equal to the terminal value. If so, the loop is executed again. If the value of the index exceeds the terminal value, control falls through the loop to the statement following the NEXT statement, with the value of the index equalling the value it was assigned the final time through the loop.

If the STEP value is omitted, a value of +1 is assumed. Since +1 is the usual STEP value, that portion of the statement is frequently omitted. The STEP value may also be a negative number.

The following example illustrates the use of loops. This loop is executed 10 times: the value of I is 10 when control leaves the loop. +1 is the assumed STEP value.

```
READY
10 FOR I=1 TO 10
20 NEXT I
30 PRINT I
40 END
RUNNH
 10

READY
```

If line 10 had been:

```
10 FOR I=10 TO 1 STEP -1
```

the value printed by the computer would be 1.

As indicated earlier, the numbers used in the FOR statement are expressions; these expressions are evaluated upon first encountering the loop. While the index, initial, terminal, and STEP values may be changed within the loop, the value assigned to the initial expression remains as originally defined until the terminal condition is reached. To illustrate this point, consider the last example program. The value of I (in line 10) can be successfully changed as follows:

```
10   FOR I=1 TO 10
15   LET I=10
20   NEXT I
```

The loop will be executed only once since the value 10 has been reached by the variable I and the terminal condition is satisfied.

If the value of the counter variable is originally set equal to the terminal value, the loop will execute once, regardless of the STEP value. If the starting value is beyond the terminal value, the loop will never execute because an initial check is made of the starting and terminal values before the loop is executed. The following statement is executed but the loop it describes would never be executed:

```
10 FOR I=10 TO 20 STEP -2
```

It is possible to exit from a FOR-NEXT loop without the index reaching the terminal value via an IF statement. Control may only transfer into a loop which has been left earlier without being completed, ensuring that the terminal and STEP values are assigned.

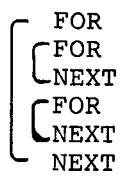
4.2 NESTING LOOPS

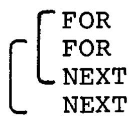
It is often useful to have one or more loops within a loop. This technique is called nesting, and is allowed as long as the field of one loop (the numbered lines from the FOR statement to the corresponding NEXT statement, inclusive) does not cross the field of another loop. The following diagram illustrates acceptable nesting procedures:

ACCEPTABLE NESTING TECHNIQUES

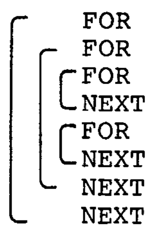
UNACCEPTABLE NESTING TECHNIQUES

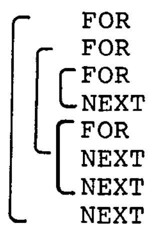
Two Level Nesting





Three Level Nesting





CHAPTER 5

LISTS AND TABLES

5.1 SUBSCRIPTED VARIABLES

In addition to single variable names, OS/8 Industrial BASIC accepts another class of variables called Subscripted Variables. Subscripted variables provide the programmer with additional computing capabilities for handling lists, tables, matrices, or any set of related variables. Variables are allowed one or two subscripts. A single letter or a letter followed by a digit forms the name of the variable; this is followed by one or two integers in parentheses and separated by commas, indicating the place of that variable in the list. Up to 31 arrays are possible in any program, subject only to the amount of core space available for data storage. For example, a list might be described as A(I) where I goes from 1 to 5, as follows:

```
          A(1),A(2),A(3),A(4),A(5)
```

This allows the programmer to reference each of the five elements in the list A. A two dimensional matrix A(I,J) can be defined in a similar manner, but the subscripted variable A can be used only once (i.e., A(I) and A(I,J) cannot be used in the same program). It is possible, however, to use the same variable name as both a subscripted and an unsubscripted variable. Both A and A(I) are valid variable names and can be used in the same program.

Subscripted variables allow data to be input quickly and easily, as illustrated in the following program (the index of the FOR statement in lines 20, 42, and 44 is used as the subscript):

```
10 REM - PROGRAM DEMONSTARTING READING
11 REM - OF SUBSCRIPTED VARIABLES
15 DIM A(5),B(2,3)
18 PRINT "A(I) WHERE A=1 TO 5:"
20 FOR I=1 TO 5
25 READ A(I)
30 PRINT A(I);
35 NEXT I
38 PRINT
39 PRINT
40 PRINT "B(I,J) WHERE I=1 TO 2:"
41 PRINT "          AND J=1 TO 3:"
42 FOR I=1 TO 2
43 PRINT
44 FOR J=1 TO 3
48 READ B(I,J)
50 PRINT B(I,J);
55 NEXT J
56 NEXT I
60 DATA 1,2,3,4,5,6,7,8
61 DATA 8,7,6,5,4,3,2,1
65 END
```

```
READY
RUNNH
A(I) WHERE A=1 TO 5:
 1  2  3  4  5

B(I,J) WHERE I=1 TO 2:
          AND J=1 TO 3:

 6  7  8
 8  7  6
READY
```

5.2 THE DIM STATEMENT

From the preceding example, it can be seen that the use of subscripts requires a dimension (DIM) statement to define the maximum number of elements in the array. The DIM statement is of the form:

```
(line number) DIM v (n ), v (n ,m )
```

where v indicates an array variable name and n and m are integer numbers indicating the largest subscript value required during the program. For example:

```
15    DIM A(6,10)
```

The first element of every array is automatically assumed to have a subscript of zero. Dimensioning A(6,10) sets up room for an array with 7 rows and 11 columns. This matrix can be thought of as existing in the following form:

$$\begin{array}{cccc}
A_{\emptyset,\emptyset} & A_{\emptyset,1} & \cdots & A_{\emptyset,1\emptyset} \\
A_{1,\emptyset} & A_{1,1} & \cdots & A_{1,1\emptyset} \\
A_{2,\emptyset} & A_{2,1} & \cdots & A_{2,1\emptyset} \\
\vdots & \vdots & & \vdots \\
A_{6,\emptyset} & A_{6,1} & \cdots & A_{6,1\emptyset}
\end{array}$$

and is illustrated in the following program:

```
10 REM - MATRIX CHECK PROGRAM
15 DIM A(6,10)
20 FOR I=0 TO 6
22 LET A(I,0)=I
25 FOR J=0 TO 10
```

```
28 LET A(Ø,J)=J
3Ø PRINT A(I,J);
35 NEXT J
4Ø PRINT
45 NEXT I
5Ø END

READY
RUNNH
 Ø  1  2  3  4  5  6  7  8  9  1Ø
 1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 2  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 3  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 4  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 5  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 6  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø

READY
```

Notice that a variable assumes a value of zero until another value has been assigned. If the user wishes to conserve core space by not making use of the extra variables set up within the array, he should set his DIM statement to one less than necessary, DIM A(5,9). This results in a 6 by 10 array which may then be referenced beginning with the A(0,0) element.

More than one array can be defined in a single DIM statement:

```
10   DIM A(20), B(4,7)
```

This dimensions both the list A and the matrix B.

A number must be used to define the maximum size of the array. A variable inside the parentheses is not acceptable and will result in an error message by BASIC at compile time. The amount of user core not filled by the program will determine the amount of data the computer can accept as input to the program at any one time. In some programs a TB error (too big) may occur, indicating that core will not hold an array of the size requested. In that event, the user should change his program to process part of the data in one run and the rest later.

NOTE

If a subscripted variable is not defined by a DIM statement, the variable is assigned an array size of ten.

CHAPTER 6

OS/8 INDUSTRIAL BASIC FUNCTIONS AND SUBROUTINES

6.1 GENERAL INFORMATION ON OS/8 INDUSTRIAL BASIC FUNCTIONS

OS/8 Industrial BASIC provides a number of functions, as part of the language, which perform calculations. The use of these functions eliminates the need for writing small programs to perform the calculations. Functions have a three letter call name, followed by an argument, X, which can be a number, variable, expression or another function. Generally, functions may be used anywhere a number or a variable is legal in a mathematical expression.

The following OS/8 Industrial BASIC functions are discussed in this chapter.

Function	Meaning
SIN(X)	Sine of X (X is expressed in radians)
COS(X)	Cosine of X (X is expressed in radians)
ATN(X)	Arctangent of X (result expressed in radians)
EXP(X)	e^X (e = 2.718282)
LOG(X)	Natural log of X ($\log_e X$)
RND(X)	Random number
ABS(X)	Absolute value of X ($\|X\|$)
INT(X)	Integer value of X
SGN(X)	Sign of X - assign a value of +1 if X is positive, 0 if X is zero, or -1 if X is negative
SQR(X)	Square root of X ($\sqrt{X}$)
FNA(X)	User-defined function
TRC(X)	Trace function - Used for debugging OS/8 Industrial BASIC programs.

In addition, there are a number of other functions provided by OS/8 Industrial BASIC, which include printing functions, string handling functions, and UDC functions. These functions are described in other parts of this manual as indicated below.

PRINTING FUNCTIONS	Refer to Paragraph
PNT(X)	3.5.4
TAB(X)	3.5.3

LEN(X$)	7.2.1
ASC(X)	7.2.2
CHR$(X)	7.2.2
VAL(X)	7.2.3
STRING HANDLING FUNCTIONS	
STR$(x)	7.2.3
POS(X$,Y$,Z)	7.2.4
SEG$(X$,Y,Z)	7.2.5
DAT$(X)	7.2.6
REAL TIME FUNCTIONS	
ANI(X,Y)	8.3.2
ANO(X,Y)	8.3.3
RDI(X,Y)	8.3.4
SDO(X,Y,Z)	8.3.5
RDO(X,Y)	8.3.6
CNI(X)	8.3.7
CNO(X,Y)	8.3.8
CLK(X)	8.3.1
LNE(X)	8.4.1
STA(X)	8.4.2
CNT(X)	8.4.3

6.2 ARITHMETIC FUNCTIONS

6.2.1 The Random Number Function - RND(X)

The RND(X) function produces pseudo-random numbers between 0 and 1. The argument X is a dummy argument and can be any number.

If the user wants the first 20 random numbers, he can write the program shown below and get 20 six-digit decimals.

```
READY
1Ø FOR L=1 TO 2Ø
2Ø PRINT RND(X),
3Ø NEXT L
4Ø END
```

```
RUNNH
 0.361572        0.332764        0.633057        0.350342        0.670166
 0.539795        0.8479          0.026123        0.54126         0.934326
 0.125244        0.389404        0.974853        0.516357        0.465088
 0.440186        0.970947        0.285889        0.867432        0.178467

READY
```

A second RUN gives exactly the same sequence of numbers as the first RUN; this is done to facilitate the debugging of programs.

```
RUNNH
 0.361572        0.332764        0.633057        0.350342        0.670166
 0.539795        0.8479          0.026123        0.54126         0.934326
 0.125244        0.389404        0.974853        0.516357        0.465088
 0.440186        0.970947        0.285889        0.867432        0.178467

READY
```

If the user wants 20 random one-digit integers, he can change line 20 to read as follows:

```
20 PRINT INT(10*RND(X)),
RUNNH
```

The results will be as follows:

```
 3               3               6               3               6
 5               8               0               5               9
 1               3               9               5               4
 4               9               2               8               1
```

To vary the type of random numbers (20 random numbers ranging from 1 to 9, inclusive), the user can change line 20 as follows:

```
20    PRINT INT(9*RND(X)+1);
```

To obtain random numbers which are integers from 5 to 24, inclusive, the user can change line 20 to the following:

```
20    PRINT INT(20*RND(X)+5);
```

If random numbers are to be chosen from the A integers of which B is the smallest, the user can call for INT(A*RND(X)+B).

6.2.1.1 The RANDOMIZE Statement

As noted in the first program in paragraph 6.2.1, the same numbers in the same order resulted both times the program was run. However, a different set will be produced with the RANDOMIZE statement, as in the following program:

```
5 RANDOMIZE
10 FOR L=1 TO 20
20 PRINT INT(10*RND(X));
30 NEXT L
40 END

READY
RUNNH
 3  3  6  3  6  5  8  0  5  9  1  3  9  5  4  4  9  2  8  1
READY
RUNNH
 0  1  9  7  9  5  3  9  1  1  4  5  4  6  2  8  9  3  7  6
READY
```

RANDOMIZE resets the numbers based on elapsed time spent waiting for terminal I/O. For example, if RANDOMIZE appears after a PRINT or INPUT instruction but before a statement with the RND(X) function, then repeated RUNs of the program produce different results. If the instruction is absent, the official list of random numbers is obtained in the usual order. It is suggested that a simulated model should be debugged without this instruction so that one always obtains the same random numbers in test runs. After the program is debugged, and before starting production runs, the user inserts the following:

```
(line number) RANDOMIZE
```

at the appropriate place in the program.

6.2.2 The Sign Function - SGN(X)

The SGN function is one which assigns the value 1 if the argument is any positive number, 0 if zero, and -1 if any negative number. Thus, SGN(7.23) = 1, SGN(0) = 0, and SGN(-.2387) = -1. For example, the following statement:

```
25    LET X=SQR(A↑2+2*B*C)*SGN(A)
```

assigns the square root of the sine of A to X.

6.2.3 The Integer Function - INT(X)

The integer function returns the value of the nearest integer not greater than X. For example, INT(34.67) = 34. By specifying INT(X+.5) the INT function can be used to round numbers to the nearest integer; thus, INT(34.67+.5) = 35. INT can also be used to round numbers to any given decimal place by specifying:

```
INT (X*10↑D+.5)/10↑D
```

where D is the number of decimal places desired. The following program illustrates this function; execution has been stopped by typing a CTRL/C:

```
10 REM - INT FUNCTION EXAMPLE
20 PRINT "NUMBER TO BE ROUNDED";
30 INPUT A
40 PRINT "NO. OF DECIMAL PLACES:";
50 INPUT D
60 LET B=INT(A*10↑D+.5)/10↑D
70 PRINT "A ROUNDED =";B
80 GO TO 20
90 END

READY
RUNNH
NUMBER TO BE ROUNDED?55.65342
NO. OF DECIMAL PLACES:?2
A ROUNDED = 55.65
NUMBER TO BE ROUNDED?78.375
NO. OF DECIMAL PLACES:?-2
A ROUNDED = 100
NUMBER TO BE ROUNDED?67.89
NO. OF DECIMAL PLACES:?-1
A ROUNDED = 70
NUMBER TO BE ROUNDED?↑C
READY
```

If the argument is a negative number, the value returned is the largest negative integer (rounded to the higher value) contained in the number. For example, INT(-23) = -23 but INT(-14.39) = -15.

6.2.4 The Absolute Value Function - ABS(X)

The absolute value function is used to obtain the absolute (positive) value of an expression. For example:

```
READY
5 PRINT ABS(-66)
10 END
RUNNH
 66
```

6.2.5 The Square Root Function - SQR(X)

The square root function is used to compute the square root of an expression. For example:

```
5 LET B=4\A=2.5\C=.5
10 PRINT  SQR(B↑2-4*A*C)
20 END
RUNNH
 3.31662

READY
```

If the argument of the SQR(X) function is <0, the absolute value of the argument is used.

6.3 TRANSCENDENTAL FUNCTIONS

6.3.1 The Sine Function - SIN(X)

The sine function is used to calculate the sine of an angle specified in radians. For example:

```
5 REM - CALCULATE SINE 30 DEGREES
10 LET P=3.14159
20 PRINT SIN(30*P/180)
25 END
RUNNH
 0.5

READY
READY
```

6.3.2 The Cosine Function - COS(X)

The cosine function is used to calculate the cosine of an angle specified in radians. For example:

```
5 REM - CALCULATE THE COSINE OF 45 DEGREES
10 PRINT COS(45*3.14159/180)
20 END

READY
RUNNH
 0.707108
```

6.3.3 The Arctan Function - ATN(X)

This function calculates the angle (in radians) whose tangent is given as the argument of the function. For example:

```
READY
5 REM - CALCULATE ATN(.57735)
10 PRINT ATN(.57735)
20 END
RUNNH
 0.523598
```

6.3.4 The Exponential Function - EXP(X)

The EXP(X) function calculates the value of e raised to the X power, where e is equal to 2.71828. For example:

```
5 REM - CALCULATE EXPONENTIAL VALUE OF 1.5
10 PRINT EXP(1.5)
20 END

READY
RUNNH
 4.48168
```

6.3.5 The Natural Logarithm Function - LOG(X)

The LOG(X) function calculates the natural logarithm of X. For example:

```
5 REM - CALCULATE THE LOG OF 959
10 PRINT LOG(959)
20 END
RUNNH
 6.86589

READY
```

6.4 USER DEFINED FUNCTIONS

6.4.1 The FNA(X) Function and the DEF Statement

In addition to the standard functions OS/8 Industrial BASIC provides, the user may define up to 26 functions of his own with the DEF statement. The name of the defined function must be three letters, the first two of which are FN, i.e., FNA, FNB,...,FNZ. Each DEF statement introduces a single function and is of the form:

```
          (line number) DEF FNA(X)=expression (X)
```

where A may be any letter and X is a dummy variable, but must be the same on each side of the equal sign. The DEF statement may appear anywhere in the program so long as it appears before the first use of the function it defines. The function itself can be defined in terms of numbers, several variables, other functions, or mathematical expressions. For example, if the user repeatedly uses the function $e^{-X^2}+5$, he can introduce the function by the following:

```
          30    DEF FNE(X)=EXP(-X↑2)+5
```

and call for various values of the function by FNE(.1), FNE(3.45), FNE(A+2), etc. This statement saves a great deal of time when the user needs values of the function for a number of different values of the variable.

The statement:

```
          DEF FNA(S)=S↑2
```

will cause the later statement:

```
          20    LET R=FNA(4)+1
```

to be evaluated as R=17.

The user-defined function can be a function of more than one variable, as shown below:

```
          25    DEF FNL(X,Y,Z)=SQR(X↑2+Y↑2+Z↑2)
```

A later statement in a program containing the above function might appear as follows:

```
          55    LET B=FNL(D,L,R)
```

where D, L, and R have been defined in the program.

6.4.2 The UDEF Function Call and the USE Statement

OS/8 Industrial BASIC has the capability for adding one or more user-coded assembly language functions. The user functions may use four numeric and two string arguments, and once properly interfaced to OS/8 Industrial BASIC, they can be used as any other OS/8 Industrial BASIC function. Complete instructions for writing and interfacing such functions are provided in Chapter 11 of this manual. A user-coded function, if present, is specified in an OS/8 Industrial BASIC program as:

```
(line number) UDEF function name (argument)
```

For example:

```
10   LET R=4
15   LET B=6
20   LET Q=10
25   UDEF PLT(X,Y,Z)
30   LET D=PLT(R,B,0)
35   PRINT 4*D
40   END
```

Line 25 introduces the function PLT to OS/8 Industrial BASIC and indicates the number and type of arguments associated with the function. In line 30 the function is used as any other standard function might be used in an OS/8 Industrial BASIC program. If the function requires the use of an array, a USE statement identifying the array must precede the statement that calls the function.

```
10   DIM S(15,5)
 .
 .
20   LET Q=10
22   USE S
25   UDEF PLT(X,Y,Z)
 .
 .
 .
```

NOTE

A UDEF function name may consist of alphabetic characters only and must have at least one argument (a dummy argument if necessary).

6.5 THE DEBUGGING FUNCTION - TRC(X)

The TRC(X) function is used by the programmer to follow the progress of a program and is, therefore, a useful debugging aid. The form of this function is:

```
(line number) vl=TRC(X)
```

where vl is a dummy variable, X=1 turns the function on and X=0 turns the function off. When TRC(1) is encountered in a program, OS/8 Industrial BASIC prints the line number of each line in the program as it is executed. The line numbers are printed between a pair of percent signs so as to be distinguishable from other material that is printed by the program. Program execution time is slowed down considerably to accommodate the function and the extra printing it causes. When TRC(0) is encountered by the program the function is turned off and normal program operation resumes.

The following example shows the effect of using the TRC(X) function in a program to check the operation of a loop. The same program with the TRC(X) function removed from the program, is also shown.

```
5 REM - BASIC
6 REM - FACTORIAL PROGRAM
10 FOR J=1 TO 5
20 GOSUB 60
30 NEXT J
40 STOP
60 LET S=1
62 T=TRC(1)
65 FOR K=1 TO J
70 LET S=S*K
75 NEXT K
77 T=TRC(0)
80 PRINT J,S
85 RETURN
90 END


RUNNH
% 65 %
% 70 %
% 77 %
 1              1
% 65 %
% 70 %
% 70 %
% 77 %
 2              2
% 65 %
% 70 %
% 70 %
% 70 %
% 77 %
 3              6
% 65 %
% 70 %
% 70 %
% 70 %
% 70 %
% 77 %
 4              24
% 65 %
% 70 %
% 70 %
% 70 %
% 70 %
% 70 %
% 77 %
 5              120
```

```
5 REM - BASIC
6 REM - FACTORIAL PROGRAM
10 FOR J=1 TO 5
20 GOSUB 60
30 NEXT J
40 STOP
60 LET S=1
65 FOR K=1 TO J
70 LET S=S*K
75 NEXT K
80 PRINT J,S
85 RETURN
90 END

READY


RUNNH
 1              1
 2              2
 3              6
 4              24
 5              120

READY
```

NOTE

OS/8 Industrial BASIC idles on input statements. Therefore, a trace should not be on when input is requested.

6.6 SUBROUTINES

A subroutine is a part of the program performing some operation that is required at more than one point in the program. Subroutines are generally placed physically at the end of a program, usually before DATA statements, if any, and always before the END statement.

6.6.1 GOSUB and RETURN

Two statements are used exclusively in OS/8 Industrial BASIC to handle subroutine; these are the GOSUB and RETURN statements.

When a program encounters a GOSUB statement of the form:

```
(line number) GOSUB x
```

where x represents the first line number of the subroutine, control then transfers to that line. For example:

```
.
.
.
50   GOSUB 200
```

When program execution reaches line 50, control transfers to line 200; the subroutine is processed until execution encounters a RETURN statement of the form:

```
(line number) RETURN
```

which causes control to return to the statement following the GOSUB statement. Before transferring to the subroutine, OS/8 Industrial BASIC internally records the next statement to be processed after the GOSUB statement; thus the RETURN statement is a signal to transfer control to this statement. In this way, no matter how many different subroutines are called, or how many times they are used, OS/8 Industrial BASIC always knows where to go next.

The following program demonstrates a simple subroutine:

```
1 REM - THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
10 DEF FNA(X)=ABS(INT(X))
20 INPUT A,B,C
30 GOSUB 100
40 LET A=FNA(A)
50 LET B=FNA(B)
60 LET C=FNA(C)
```

```
70 PRINT
80 GOSUB 100
90 STOP
100 REM - THIS SUBROUTINE PRINTS OUT THE SOLUTIONS
110 REM - OF THE EQUATION A(X↑2)+B(X)+C=0
120 PRINT "THE EQUATION IS";A;"*X↑2+";B;"*X+";C
130 LET D=B*B-4*A*C
140 IF D<>0 THEN 170
150 PRINT "ONLY ONE SOLUTION...X=";-B/(2*A)
160 RETURN
170 IF D<0 THEN 200
180 PRINT "TWO SOLUTIONS...X=";
185 PRINT (-B+SQR(D))/(2*A);"AND X=";(-B-SQR(D))/(2*A)
190 RETURN
200 PRINT "IMAGINARY SOLUTIONS...X=(";
205 PRINT -B/(2*A);",";SQR(-D)/(2*A);") AND (";
207 PRINT -B/(2*A);",";-SQR(-D)/(2*A);")"
210 RETURN
900 END

READY
RUNNH
?1,.5,-.5
THE EQUATION IS 1 *X↑2+ 0.5 *X+-0.5
TWO SOLUTIONS...X= 0.5 AND X=-1

THE EQUATION IS 1 *X↑2+ 0 *X+ 1
IMAGINARY SOLUTIONS...X=( 0 , 1 ) AND ( 0 ,-1 )
```

Line 100 begins the subroutine. There are several places in which control may return to the main program, depending upon a certain condition being satisfied. The subroutine is executed from line 30 and again from line 80. When control returns to line 90, the program encounters the STOP statement and execution is terminated.

It is important to remember that subroutines should generally be kept distinct from the main program. The last statement in the main program should be a STOP or GOTO statement, and subroutines are normally placed following this statement.

More than one subroutine may be used in a single program in which case these can be placed one after another at the end of the program (in line number sequence). A useful practice is to assign distinctive line numbers to subroutines. For example, if the main program is numbered with line numbers up to 199, 200 and 300 could be used as the first numbers of two subroutines.

6.6.2 Nesting Subroutines

Nesting of subroutines occurs when one subroutine calls another subroutine. If a RETURN statement is encountered during execution of a subroutine, control returns to the statement following the GOSUB which called it. From this point, it is possible to transfer to the beginning or any part of a subroutine, even back to the calling subroutine. Multiple entry points and RETURN statements make subroutines more versatile.

The maximum level of GOSUB nesting is ten levels, which should prove more than adequate for all normal uses. Exceeding this limit results in the message:

```
GS AT LINE YYYYY
```

where YYYYY represents the line number where the error occurred. An example of GOSUB nesting follows (execution has been stopped by typing a CTRL/C, as the program would otherwise continue in an infinite loop).

```
10 REM - FACTORIAL PROGRAM USING GOSUB TO
15 REM - RECURSIVELY COMPUTE THESE FACTORS
40 INPUT N
50 IF N>20 THEN 120
60 X=1
70 K=1
80 GOSUB 200
90 PRINT "FACTORIAL ";N;"=";X
110 GO TO 40
120 PRINT "MUST BE 20 OR LESS"
130 GO TO 40
200 X=X*K
210 K=K+1
220 IF K<=N THEN 200
230 RETURN
240 END

READY
RUNNH
?2
FACTORIAL  2 = 2
?4
FACTORIAL  4 = 24
?5
FACTORIAL  5 = 120
?21
MUST BE 20 OR LESS
?6
FACTORIAL  6 = 720
?↑C
READY
```

CHAPTER 7

ALPHANUMERIC INFORMATION (STRINGS)

In previous chapters we have dealt only with numerical information. However, OS/8 Industrial BASIC also processes, or manipulates, alphanumeric information called strings. A string is a sequence of characters, each of which is a letter, a digit, a space, or some character other than a statement terminator (backslash or carriage return).

7.1 STRING CONVENTIONS

7.1.1 Constants and Variables

Strings may appear as constants or variables just as numerics may. We have already used string constants in PRINT statements. For example:

```
100   PRINT "THIS IS A STRING CONSTANT"
```

where the alphanumerics enclosed in quotes are the string constant. Naming a string variable is similar to naming a numeric variable. It consists of a letter followed by a dollar sign ($) or a letter and a single digit followed by $. A$ and A1$ are both legitimate string variable names; 2A$ and AA$ are not legitimate string variable names.

7.1.2 Dimensioning Strings

OS/8 Industrial BASIC assumes that a string length is 8 characters or less unless a string has been dimensioned in the form:

```
10   DIM A$(I)
```

where I is the length of string variable A$. I cannot exceed 72.

String lists (equivalent to single subscripted numeric variables) are permitted in OS/8 Industrial BASIC and must be dimensioned in the form:

```
20   DIM A$(K,L)
```

where K is the number of strings in the list and L is the length of each string.

When referencing a subscripted string variable in a LET or IF-THEN statement, for example:

```
25   LET B$(I) = "YES"
```

the expression I represents the place of that string variable in the list B$.

Double subscripted string variables (string tables) are not permitted in OS/8 Industrial BASIC.

7.1.3 Inputting String Data

String data may be included in a DATA list but must always be enclosed by quotation marks. In fact, any string written into a program must be enclosed by quotation marks to be recognized by the OS/8 Industrial BASIC Compiler.

```
10   READ A$,B$,C$
20   PRINT C$;B$;A$
25   DATA "NG","RI","ST"
30   END
```

The program above prints STRING.

Quotation marks may be included in strings by indicating two quotation marks in succession. For example the string A"B would appear in a program as:

```
10   LET A$ = "A""B"
```

Both string data and numeric data may be intermixed in a DATA list but the burden falls on the programmer to assemble the list in the correct sequence, since all READ statements for both string and numeric data remove data serially from the DATA list. If he does not, the results of the READ statement are unpredictable.

The INPUT statement may also be used for inputting string data to a program. Quotation marks are not necessary when inputting string data in response to the question mark (?) generated by the INPUT statement unless the quotation marks are deliberately meant to be part of the string.

```
 .
 .
 .
330   PRINT "DO YOU WISH TO CONTINUE?"
340   INPUT A$
350   IF A$="YES" THEN 410
360   PRINT "ARE YOU SURE?"
370   INPUT B$
380   IF B$="NO" THEN 410
390   PRINT "PROGRAM STOPPED"
400   STOP
410   PRINT "LET'S CONTINUE"
 .
 .
 .
 .
 .
 .
490   END
```

Each string literal requested by an INPUT statement must be terminated by a carriage return or comma which acts as the data delimiter. This is necessary since all characters, except for the carriage return or comma, are recognized as part of the data string.

```
READY

10   INPUT A$,B$,C$
 .
 .
 .


RUNNH
?ABCD
?EFGH
?IJ
```

In the above example A$="ABCD", B$="EFGH" and C$="IF".

7.1.4 Strings in LET and IF-THEN Statements

Strings may be used in both LET and IF-THEN statements as already indicated by some of the previous examples. Any of the relational operators decribed in paragraph 2.3.3 may be used in an IF-THEN statement to compare strings. Strings are compared on the basis of the ASCII numeric value of each character in the string (see Appendix D for numeric values of ASCII characters).

When comparing strings in an IF-THEN statement, the relational operators have the following significance:

Operator	Meaning
<	earlier in ASCII numeric order than
>	later in ASCII numeric order than
=< or <=	same ASCII numeric order as or earlier in ASCII numeric order than
=	same ASCII numeric order
>< or <>	different ASCII numeric order from
=> or >=	same ASCII numeric order as or later in ASCII numeric order than

For example:

```
10   IF "ABCD"<"ABC@" THEN 50
20   STOP
 .
 .
 .
50   LET A$="ABCD"
```

Each character in string ABCD is compared, left-to-right, with the respective character in string ABC@. A, B, and C match but D and @ do not. From Appendix D, the character @ has a lower numeric value than the character D. Therefore the string ABCD is not earlier in ASCII numeric sequence than ABC@ and the program stops at line 20.

If the strings in an IF-THEN comparison are of unequal length, then OS/8 Industrial BASIC lengthens the shorter string to make it equal in length to the longer string by appending an appropriate number of ASCII space characters. In the following example

```
10   IF "ABCD"<"AB" THEN 50
20   STOP
 .
 .
50   LET A$="ABCD"
```

string "AB" is treated as "AB␣␣". Since the character C is earlier in ASCII numeric order than the character "space", the IF-THEN statement is true and control is transferred to line 50.

7.1.5 String Concatenation

Strings can be concatenated by means of the operator ampersand (&). The ampersand can be used to concatenate string expressions wherever a string expression is legal, with the exception that information cannot be stored by means of a LET statement in concatenated string variables. That is, concatenated string variables cannot appear to the left of the equal sign in a LET statement. For example, LET A$=B$&C$ is legal, but LET A$&B$=C$ is not. An example of string concatenation is:

```
10   READ A$,B$,C$
20   PRINT C$&B$&A$
25   DATA "NG","RI","ST"
30   END
```

Running this program (a modification of the program in paragraph 7.1.3) causes STRING to be printed.

7.2 STRING HANDLING FUNCTIONS

A number of functions have been implemented that perform manipulations on strings. These functions are LEN, ASC, CHR$, VAL, STR$, POS, SEG$, and DAT$. Functions that return strings have names that end in a dollar sign ($); those functions that return numbers have names that do not end in a dollar sign.

7.2.1 The LEN Function

The LEN function returns the number of characters in a string. It has the form:

```
LEN(X$)
```

Example:

```
5 DIM B$(10)
10 READ A$,B$
20 PRINT LEN(A$&B$&"AROUND")
30 DATA "UP, ","DOWN, AND "
40 END

READY
RUNNH
 20

READY
```

7.2.2 The ASC and CHR$ Functions

The ASC and CHR$ functions perform conversion from and to ASCII, respectively. The ASC function converts a one-character string to its ASCII decimal equivalent, and the CHR$ function converts a decimal number to its equivalent ASCII character.

The ASC function has the form:

```
ASC (argument)
```

The argument is a one character string. ASC returns the equivalent ASCII decimal number for the character.

The CHR$ function has the form:

```
CHR$ (numeric expression)
```

The value of the numeric expression is truncated to an integer that is in the range 0 to 63. Integers greater than 63 are treated modulo 64. That is, they are divided by 64 and the remainder becomes the new integer. This integer is then interpreted as an ASCII decimal number that is converted to its equivalent character (refer to Appendix D for the ASCII decimal numbers and the equivalent characters).

An example of the ASC and CHR$ functions follows:

```
5  FOR T=ASC("A") TO ASC("A")+3
7  NEXT T
10 PRINT "THIS IS TEST "&CHR$(T)
```

This is the beginning of a FOR loop that successively prints:

```
THIS IS TEST A
      .
      .
      .
THIS IS TEST B
      .
      .
      .
THIS IS TEST C
      .
```

```
          .
          .
THIS IS TEST D
```

7.2.3 The VAL and STR$ Functions

The VAL and STR$ functions perform conversions from strings to numbers and numbers to strings. The form of the VAL function is:

```
VAL (string expression)
```

The string expression must look like any number which may be legally typed in response to an INPUT statement. VAL returns the actual number that the string represents. The VAL function does not return the ASCII value of the number that the string represents, it returns the number. For example, VAL ("25") returns the number 25. The 25 that is the argument to VAL is a string, the 25 that VAL returns is a number.

Example:

```
READY

10   INPUT A$
20   PRINT VAL(A$)*2
       .
       .
       .
100    END


RUNNH

?2.46111
 4.92222

READY
```

The STR$ function returns the string representation (as a number) of its argument. The form of STR$ is:

```
STR$ (numeric expression)
```

The string that is returned is in the form in which numbers are output in BASIC. For example, PRINT STR$(1.76111124) prints the string 1.76111.

7.2.4 The POS Function

The POS function is of the form:

```
POS(X$,Y$,Z)
```

The function returns the location in string X$ of the first occurrence of string Y$ starting with Zth character in string X$. For example:

```
        20   LET X$="MONDAY"
        25   LET X=POS(X$,"DAY",1)
```

After line 25, X will be equal to 4. The arguments of the POS function may be constants, variables, or expressions.

The following rules apply in evaluating the POS(X$,Y$,Z) function.

1. If Y$ is a null string (no characters) then POS(X$,Y$,Z)=1

2. If X$ is a null string (no characters) then POS(X$,Y$,Z)=0

3. If Z<0, a fatal error (PA) is detected and program execution stops

4. If Y$ is not found, then POS(X$,Y$,Z)=0

7.2.5 THE SEG$ Function

This function is of the form:

```
        SEG$(X$,Y,Z)
```

The function returns the substring of X$ which is between positions Y and Z inclusively. For example:

```
        20   LET X$="MONDAY"
        25   LET B=6
        30   LET A$=SEG$(X$,2*B/3,B)
```

After line 30, A$ is equal to "DAY". The arguments of the SEG$ function may be variables, constants, or expressions.

The following rules apply in evaluating the SEG$(X$,Y,Z) function.

1. If Y<0, Y is set equal to 1

2. If Y> length X$, then SEG$(X$,Y,Z) = null string (no characters)

3. If Z,0, then SEG$(X$,Y,Z) = null string

4. If Z > length X$, then Z is set equal to length of X$

5. If Z,Y, then SEG$(X$,Y,Z) = null string

7.2.6 The DAT$ Function

The DAT$ function is of the form:

```
        DAT$(X)
```

The function returns an eight character string giving the current date in the form MM/DD/YY. For example:

```
SCRATCH

READY

20    PRINT DAT$(X)
30    END


RUNNH

 7/ 1/73

READY
```

The use of DAT$ function assumes the user has specified the date in the OS/8 monitor command "DATE". If the DATE command was not used, the DAT$ function outputs a null string (no characters).

If the Industrial BASIC clock is set, and then exceeds one day, the next call to the DAT$ function will return the next day and update the OS/8 date.

CHAPTER 8

REAL TIME OPERATIONS

8.1 GENERAL DESCRIPTION

This chapter contains the statements and functions which give the user the facility to support asynchronous operation, and control and monitor external devices.

The following BASIC Statements are used to associate BASIC routines with external events:

TIMER	-time based operations
COUNTER	-UDC counter operations
CONTACT	-UDC contact operations
DISMISS	-termination of service routines

These statements are explained in detail later in this chapter.

The following function calls are used to access, synchronously, Real Time devices:

ANI	-Analog Input
ANO	-Analog Output
RDI	-Read digital input
RDO	-Read digital output
SDO	-Send digital output
CNI	-Set counter
CNO	-Read counter
CLK	-Set or read clock
CNT	-Information regarding
LNE	asynchronous operations
STA	

These function calls are also explained in detail later in this chapter.

8.2 REAL TIME BASIC STATEMENTS

8.2.1 TIMER

```
TIMER V THEN X
```

where X is the line number of the first statement of the User Process Interrupt Service Routine (UPIR), and V is the time interval in seconds.

When the time interval has elapsed the user service routine is scheduled for execution and the timer is restarted. Time intervals of less than .1 second may have significant inaccuracies in timing. Time intervals which are zero or negative deactivate all the timers associated with the specified line number. This allows the user to change the elapsed time intervals or stop the timers entirely. A maximum of 4 timers may be active at any time.

Examples of TIMER Statements:

```
          200 TIMER 10 THEN 400
```

Every 10 seconds, the UPIR beginning at line 400 will be executed.

```
          300 TIMER 0 THEN 400
```

This statement will cause all timers associated with line 400 of the UPIR to be deactivated.

```
          100 TIMER 7 THEN 400
          110 TIMER 3 THEN 400
```

These statements cause 2 timers to start; a 7-second timer that will cause the UPIR at line 400 to be scheduled for execution every 7 seconds, and a 3-second timer that will cause the UPIR at line 400 to be scheduled for execution every 3 seconds. Note that at 21-second intervals the UPIR will be scheduled twice.

The following is an example of a time based operation:

```
           10  TIMER 10 THEN 400
           20  TIMER 15 THEN 500
           25  LET Y = 0
           30  GOTO 25
          400  PRINT "SEGMENT 400"
          410  DISMISS
          500  PRINT "SEGMENT 500"
          510  DISMISS
          600  END
```

this example will print "SEGMENT 400" every 10 seconds, and "SEGMENT 500" every 15 seconds until terminated with a CTRL/C.

8.2.2 COUNTER

```
          COUNTER V THEN X
```

where X is the line number of the first statement of the UPIR and V is the counter module to associate with the given line number. There is a maximum of 4 counter modules supported. As with timers, when the counter number is zero or negative all counters associated with the line number are disabled.

Counter modules should be loaded via the CNO function for counting operations. For additional information on Counter Modules refer to the UDC8 UNIVERSAL DIGITAL CONTROL SUBSYSTEM MAINTENANCE MANUAL, DEC-08-HZDC-D.

Example of a Counter Statement:

```
          200 COUNTER 1 THEN 1000
```

When Counter 1 counts to zero the UPIR at line 1000 will be scheduled.

The following is an example of a program to load a counter with a number of items to count and log the completion of the count:

```
 10  COUNTER 1 THEN 100
 20  REM NOW LOAD COUNTER
 25  LET A = CNO(1,100)
 30  LET A = 0
 35  GOTO 30
100  PRINT "COUNT CYCLE COMPLETE"
110  DISMISS
200  END
```

8.2.3 CONTACT

```
CONTACT V THEN X
```

where X is the line number associated with the specified CONTACT. The value V is the CONTACT number. The maximum number of CONTACTs is 36, in the range 1 to 36. If zero is specified as a CONTACT number all active CONTACT user interrupt service routines associated with the line number in the CONTACT statement are deactivated.

Example of a CONTACT statement:

```
200 CONTACT 1 THEN 600
300 CONTACT 2 THEN 600
400 CONTACT 3 THEN 700
```

After these statements are executed a change of state in CONTACT 1 or 2 will cause the UPIR at line 600 to be scheduled. A change of state in CONTACT 3 will schedule the UPIR at line 700.

8.2.4 DISMISS

A DISMISS statement is used to terminate a user interrupt service routine. This causes the mainline code to resume execution. If another user process interrupt routine was scheduled, it will receive control at this point. DISMISS performs an action similar to that of RETURN.

8.3 EXTENDED FUNCTIONS FOR INPUT OR OUTPUT

The Extended I/O Functions allow the user to turn a digital output "on" or "off", read a switch position, set an analog voltage, or read analog voltage via the UDC. An added function allows reading or setting of the system clock.

Each class of functional I/O module (analog input, analog output, digital input, digital output) operates on a continuous range of logical addresses. The mapping between the logical address and the corresponding physical address of the UDC module and subchannel is performed within the system, based on tables which define the UDC configuration and which the user generates during "SYSTEM GENERATION".

Thus, analog input point one (1) is different from analog output point (1). This prevents the user from specifying meaningless operations (attempting to set a bit "on" in an input module). In addition, UDC reconfigurations will require only a new table generation and no BASIC level programming changes. The logical addressing simplifies the use of FOR - NEXT loops.

In general, all arguments to the Industrial Functions must be within the range:

0   argument   4095

Negative arguments will generate a fatal error, FM; arguments that are out of range generate a FO.

8.3.1 The Clock Function - CLK(X)

This function has two operations:

1. If X is positive, the system clock is set to the value of X in seconds. The value of X must be less than 86400 (the number of seconds in 24 hours).
2. If X is zero or negative, the value (in seconds) of the system clock is returned.

Example:

1. Set the system clock to 12:00:00

```
100 LET T = 12*3600
200 LET X = CLK(T)
300 END
```

2. Read the system clock

```
100 LET T = CLK(0)
```

3. Set the system clock to the value entered on the terminal

```
 50  GOTO 100
 75  PRINT "INVALID TIME"
100  PRINT "ENTER PRESENT TIME AS HH:MM:SS";
110  INPUT A,B,C
120  IF A < 0 THEN 75
125  IF A > 23 THEN 75
130  IF B < 0 THEN 75
135  IF B > 59 THEN 75
140  IF C < 0 THEN 75
145  IF C > 59 THEN 75
150  LET T = CLK(A*3600+B*60+C)
.
.
.
900  END
```

8.3.2 Analog Input Function - ANI(C,G)

This function returns the results, in volts, of reading channel (C) at gain (G). The valid gains for the ADU01 are: 1000,200,100,50,20,10,2,1.

Example:

Read analog input channel 6 at a gain of 10, input voltage is .5 volts.

```
          100 LET V= ANI (6,10)
          200 PRINT V
```

the value printed will be 5.

8.3.3 Analog Output Function - ANO(C,V)

The ANO function sets the analog output channel (C) to the value (V). The actual output generated is a function of the modules used. However, full scale output will be generated by a value (V) of 1023, and minimum output will be generated by a value of zero. Values greater than 1023 or less than zero are illegal.

Example of full scale output on channel 3:

```
          100 LET A = ANO (3,1023)
```

8.3.4 Read Digital Input - RDI (P,N)

This function returns the value of the digital input points starting at point P for N number of points. The read may not cross a boundary between modules (points divisible by twelve, i.e., 12,24 etc.).

Example:

```
          100 LET A = RDI (25,4)
          200 LET B = RDI (1,1)
```

Line 100 reads 4 points (25-28) and returns the value as A, this is useful for BCD input devices. Line 200 reads the value of point 1 and returns a 1 if the point is "on", and a zero if it is "off".

8.3.5 Send Digital Output - SDO (P,N,V)

The SDO function sets the digital output point(s) specified by P and N to the value V. The value must be a positive, and able to be represented within the range of N. The definition of P and N is the same as in the RDI function. The value of V, in binary notation, is sent right-justified into the range stated.

Example:

```
          100 LET A = SDO (4,1,1)
          200 LET A = SDO (1,4,10)
```

Line 100 will turn point 4 on, while line 200 will turn on points 1 and 3.

8.3.6 Read Digital Output - RDO (P,N)

This function is identical to the RDI function except that the "read" is of the digital output channel(s). The data return is read from a table, not the actual physical channels. Therefore, output modules that alter state independently of computer operations (single shot output) may be "on" in the output table but "off" physically.

8.3.7 Counter Input - CNI (P)

This function returns the number of counts remaining in the counter module.

Example:

```
100 LET A = CNI (3)
```

A is assigned the value of the number of events remaining to be counted in counter 3.

8.3.8 Counter Output - CNO (P,V)

Counter output is used to set a counter module. P is the channel of the counter module, and V is the value of the number of items to count before generating an interrupt.

Example:

```
100 LET A = CNO (2,300)
```

This will set counter 2 to count 300 events.

8.4 UPIR IDENTIFICATION FUNCTIONS

The following functions provide the user with a means of identifying the channel (LNE) and state (STA) of the device that caused the interrupt. This will allow the user to have more than one UPIR for several modules of the same type.

8.4.1 Line - LNE (X)

The LNE function returns the channel number of the module that caused the interrupt. The LNE function has meaning only in a UPIR.

Example:

```
100 LET A = LNE (0)
```

Upon entry to the UPIR at line 100, A is assigned the channel number that caused the UPIR to be invoked.

8.4.2 State - STA (X)

The STA function is used to find the state of contact associated with the channel that scheduled the UPIR.

Example:

```
          100 LET A = STA (LNE (0))
```

A is assigned the state of the contact that changed to cause the interrupt.

8.4.3 Count - CNT (X)

This function returns the number of identical schedule requests for the same UPIR.

8.5 EXAMPLE CONTROL PROGRAM

The example control program demonstrates the maintaining of a constant temperature bath via the following operations:

1. Open the hot and cold water valves.
2. Open the drain and turn the "BATH OK" lamp off.
3. Measure the bath temperature; if it is not 68 +or- .5 degrees turn the "BATH OK" lamp off and adjust the hot water valve. If the bath temperature is 68 +or- .5 degrees turn the "BATH OK" lamp on.
4. Close the hot and cold water valves, and open the drain when the switch is closed to shut down the process.

The hardware interfaces are as follows:

ELEMENT	TYPE	POINT
HOT VALVE	UDC/DAC	1
COLD VALVE	UDC/DAC	2
DRAIN VALVE	UDC/DO	1
BATH OK LAMP	UDC/DO	3
TEMP. SENSOR	UDC/ADU01	4
SHUT-DOWN SWITCH	UDC/CI	2

The Industrial BASIC program would be:

```
100 REM CONTACT 2 SHUTS DOWN ENTIRE RUN
120  CONTACT 2 THEN 960
140  REM SCAN FOR TEMPERATURE EVERY 5 SECONDS
160  TIMER 5 THEN 500
180  REM SET SOFTWARE SWITCH FOR RESTART
200  LET S=1
220  REM OPEN DRAIN VALVE
240  LET A=SDO(1,1,1)
260  REM OPEN HOT AND COLD VALVES 50%
```

```
280  LET H=1023*.50
300  LET C=1023*.50
320  LET A=ANO(1,H)
340  LET A=ANO(2,C)
360  REM SHUT BATH OK LAMP OFF
380  LET A=SDO(3,1,0)
400  REM NOW IDDLE
420  LET A=0
440  IF S=0 GOTO 140
460  GOTO 420
480  REM
500  LET T=ANI(4,200)
520  GOSUB 680
540  REM CONVERTS VOLTS TO DEGREES TEMPERATURE
560  IF T>68.5 THEN 740
580  IF T<67.5 THEN 800
600  REM TEMPERATURE OK TURN BATH OK LAMP ON
620  LET B=SDO(3,1,1)
640  DISMISS
660  REM
680  REM SUBROUTINE TO CONVERT VOLTS TO DEGREES
700  RETURN
720  REM
740  LET H=H*.95
760  LET C=C*1.05
780  GOTO 860
800  LET H=H*1.05
820  LET C=C*.95
840  REM STEP VALVE AND TURN OFF BATH OK LAMP
860  LET B=ANO(1,H)
880  LET B=ANO(2,C)
900  LET B=SDO(3,1,0)
920  GOTO 640
940  REM
960  LET S=STA(LNE(0))
980  IF S=0 THEN 1140
1000  REM SHUT BATH DOWN
1020  LET B=ANO(1,0)
1040  LET B=ANO(2,0)
1060  LET B=SDO(3,1,0)
1080  LET B=SDO(1,1,1)
1100  REM NOW STOP TIMER
1120  TIMER 0 THEN 500
1140  DISMISS
1160  END
```

NOTE

The system will resume control when the SHUT-DOWN SWITCH is restored to normal position.

8.6 POWER FAIL - RESTART

If a power failure occurs at Run-Time the system will close any open files and chain to a BASIC file (PWRUP.BA). The user may write his own restart operations.

NOTE

TD8/E file operations require that the Interrupt System be "off". Therefore, power fail on TD8/E's may not be detected if file I/O is in progress. Closing an open file will delete any file with the same name and extension.

CHAPTER 9

EDITING AND CONTROL COMMANDS

Several commands for editing OS/8 BASIC programs and for controlling their execution enable the user to perform such operations as:

- erase characters or lines
- list part or all of a program
- save programs on various storage devices, and
- call program from storage devices.

NOTE

See Chapter 10 for a description of OS/8 storage devices and files.

9.1 CORRECTING PROGRAMS

9.1.1 Erasing Characters and Lines

Errors made while typing programs at the terminal are easily corrected. Typing a SHIFT/O or pressing the RUBOUT key causes deletion of the last character typed, and echoes back arrow (←) on the terminal. One character is deleted each time the key is typed. For example:

```
          20 DEN F←←←F FNA(X,Y)=X↑2+3*Y
```

The user types N instead of F and immediately notices his mistake. He presses the RUBOUT key (or SHIFT/O) three times, which is once for as many characters including spaces to be deleted. He makes the correction and continues typing the line. The typed line enters the computer only when the RETURN key is pressed.

```
          20 DEF FNA (X,Y)=X↑2+3*Y
```

Sometimes it is easier to delete a line being typed and retype the line rather than attempt a correction using rubouts. Typing CTRL/U or pressing the ALTMODE key will delete the line currently being worked on and echoes DELETED and a carriage return-line feed. Use of the CTRL/U or ALTMODE command is equivalent to typing rubouts back to the beginning of the line.

To delete a line that has already been entered into the computer the user simply types the line number followed by a carriage return. Both the line number and the line are removed from his program.

The user may change individual lines by simply typing them in again. Whenever a line is entered, it replaces any existing line which has the same line number. New lines may be inserted anywhere in the program by giving them line numbers which are between two other existing line numbers. Using these editing capabilities, the program may be modified and re-run until it works properly.

9.1.2 The RESEQ Program

After the user has extensively modified his program, he may find that some portions of the program have line numbers spaced so closely together that they do not permit any further addition of statements should he wish to do so. Renumbering the lines in the program so as to provide a practical increment between line numbers can be accomplished automatically by using the RESEQ program listed below. The RESEQ program is saved as RESEQ.BA. It should be noted that the RESEQ program modifies the line numbers in all statements containing them (such as GOSUB and IF-THEN statements) to agree with the new line numbers assigned to statements by the program. Line lengths must not exceed 72 characters.

```
990    REM PROGRAM RESEQ
1000   DIM L$(72),F$(1),C$(1),N$(16)
1010   DIM L2$(72)
1020   DIM N(350)
1030   LET F$=CHR$(28)
1040   PRINT "FILE";
1050   INPUT N$
1060   PRINT "START, STEP";
1070   INPUT S1,S
1080   LET S1=INT(ABS(S1))
1090   LET S=INT(ABS(S))
1100   LET T=0
1110   LET N2=0
1120   FILE #1:N$
1130   LET I=1
1140   INPUT #1:L$
1150   IFEND #1 THEN 1320
1160   LET L=LEN(L$)
1170   GOSUB 1980
1180   IF N1>0 THEN 1220
1190   PRINT "NO LINE NUMBER"
1200   PRINT L$
1210   GO TO 1130
1220   IF N1>N2 THEN 1260
1230   PRINT "OUT OF SEQUENCE"
1240   PRINT L$
1250   GO TO 1130
1260   LET N2=N1
1270   LET T=T+1
1280   LET N(T)=N1
1290   IF T<350 THEN 1130
1300   PRINT "TOO MANY LINES"
1310   STOP
1320   RESTORE #1
1330   FILEV #2:N$
```

```
1340  LET N2=S1
1350  INPUT #1: L$
1360  IFEND #1 THEN 1730
1370  LET I=1
1380  LET L=LEN(L$)
1390  GOSUB 1980
1400  LET L2$=STR$(N2)
1410  PRINT #2:L2$;
1420  LET L$=SEG$(L$,I,72)
1430  LET N2=N2+S
1440  LET F=0
1450  LET D=POS(L$,F$,1)\LET P=D
1460  IF D=0 THEN 1490
1470  LET L2$=SEG$(L$,P+1,72)
1480  LET L$=SEG$(L$,1,P-1)
1490  LET I=POS(L$,"GOTO",1)+4
1500  IF I>4 THEN 1750
1510  LET I=POS(L$,"GOTO",1)+5
1520  IF I>5 THEN 1750
1530  LET I=POS(L$,"THEN",1)+4
1540  IF I>4 THEN 1750
1550  LET I=POS(L$,"GOSUB",1)+5
1560  IF I>5 THEN 1750
1570  LET I=POS(L$,"GOSUB",1)+6
1580  IF I>6 THEN 1750
1590  IF F=0 THEN 1610
1600  PRINT #2:F$;
1610  PRINT #2:L$;
1620  LET F=F+1
1630  IF D>0 THEN 1660
1640  PRINT #2:
1650  GO TO 1350
1660  LET D=POS(L2$,F$,1)\LET P=D
1670  IF D>0 THEN 1700
1680  LET L$=L2$
1690  GO TO 1490
1700  LET L$=SEG$(L2$,1,P-1)
1710  LET L2$=SEG$(L2$,P+1,72)
1720  GO TO 1490
1730  CLOSE #2
1740  STOP
1750  LET L=LEN(L$)
1760  GOSUB 1920
1770  IF C=32 THEN 1760
1780  IF C<0 THEN 1890
1790  LET I=I-1
1800  LET P=I
1810  GOSUB 1980
1820  IF N1=0 THEN 1890
1830  FOR J=1 TO T
1840  IF N1<>N(J) THEN 1880
1850  LET Q$=STR$(J*S-S+S1)
1860  LET L$=SEG$(L$,1,P-1)&Q$
1870  GO TO 1590
1880  NEXT J
1890  PRINT "BAD REFERENCE"
1900  PRINT L$
1910  GO TO 1590
1920  IF I<=L THEN 1950
```

```
1930  LET C=-1
1940  RETURN
1950  LET C=ASC(SEG$(L$,I,I))
1960  LET I=I+1
1970  RETURN
1980  LET N1=0
1990  GOSUB 1920
2000  IF C<48 THEN 2040
2010  IF C>57 THEN 2040
2020  LET N1=N1*10+C-48
2030  GO TO 1990
2040  IF C<0 THEN 2060
2050  LET I=I-1
2060  RETURN
2070  END
```

Typically, the program would be used as follows:

SAVE SYS:SAMPLE	User saves program SAMPLE which requires renumbering.
READY	BASIC is ready for next command.
OLD SYS:RESEQ	User calls for program RESEQ.
READY	BASIC is ready for next command.
RUNNH	User runs program.
FILE? SYS:SAMPLE.BA	Program asks for filename. User responds with name of program to be renumbered.
START, STEP?100,10	Program asks for a starting line number (START) and for the increment between line numbers (STEP). User requested that SAMPLE start with line number 100 and each line be incremented by 10.
READY	Renumbering is accomplished. BASIC ready for next command.
OLD SYS:SAMPLE	User calls back his program.
READY	BASIC ready for next command.
LISTNH	User gets listing of program SAMPLE for further modification.

NOTE

All commands described in the following paragraphs, except LISTNH and RUNNH, may be abbreviated using the first two letters of the command.

9.2 THE LIST AND LISTNH COMMANDS

An entire program can be listed on the terminal by typing LIST followed by a carriage return. A heading is printed before the program itself and is of the form:

```
FILE EX VERSION NO. DATE
```

For example, if the user is working on a program named USER.BA and wants a listing he types:

```
LIST
```

and BASIC responds with:

```
USER    BA    1.0    26-JUL-72

100 LET X=1
  .
  .
  .
  .
  .
  .
200 END

READY
```

NOTE

> When any OS/8 BASIC command is completed, the message READY is printed at the terminal. OS/8 BASIC is then ready to accept any other commands from the user.

A part of a program may be listed by typing LIST followed by a line number. This causes that line and all following lines in the program to be listed. For example:

```
LIST 100
```

will list line 100 and all remaining lines in the program. Typing CTRL/O while the listing is being printed terminates the printing and outputs the READY message.

The LISTNH command may be used exactly as the LIST command, but it eliminates the heading from the listing.

9.3 THE SCRATCH COMMAND

The command:

```
SCRATCH
```

is provided to allow the programmer to clear his storage area, deleting any commands, or a program which may have been previously entered, and leaving a clean area in which to work. If the storage area is not cleared before entering a new program, lines from previous programs may be executed along with the new program, causing errors or misinformation. The SCRATCH command eliminates all old statements and numbers and should be used before new programs are created.

9.4 THE NEW COMMAND

The NEW command is used to name a program you are going to create and performs an inherent SCRATCH on the storage area. The command is in the form:

```
NEW FILE.EX
```

It assigns the filename to the program to be created. For example:

```
NEW USERA.BA
```

creates file USERA.BA.

An alternate method of naming programs is to type NEW followed by the RETURN key. BASIC responds with:

```
FILE NAME--
```

The user types the filename and extension followed by the RETURN key.

```
NEW
FILE NAME--USER.BA         (.BA is assumed and need not
                           be typed)
```

NOTE

Extension .BA is assumed in the OLD, NEW, NAME, and SAVE commands unless otherwise specified. If no extension is specified in the OLD command, OS/8 Industrial BASIC first tries to load FILE.BA and if unsuccessful tries to locate and load FILE without the extension.

9.5 THE OLD COMMAND

This command retrieves a previously created file from a storage device and places the file in the storage area of the computer. The command is of the form:

```
OLD DEV:FILE.EX
```

For example:

```
          OLD DTA1:SAMPLE.BA
```

retrieves file SAMPLE from DECtape number 1 and places it in the storage area of the computer.

An alternate method of retrieving a file is to type OLD followed by the RETURN key. BASIC responds with:

```
FILE NAME--
```

The user types the device, filename and extension followed by the RETURN key.

```
          OLD
          FILE NAME--DTA1:SAMPLE.BA
```

If the device is omitted, DSK: is assumed.

9.6 THE NAME COMMAND

This command permits the user to rename the program in the storage area of the computer. The command is of the form:

```
          NAME FILE.EX
```

Since this command changes only the name of the storage area of the computer - not its contents, it can be used to create two almost identical versions of the same program. This is accomplished by retrieving the first file, making modifications to it, and then SAVEing (see paragraph 9.7) the modified version under the new name.

9.7 THE SAVE COMMAND

The SAVE command saves on the specified device the file currently in the storage area of the computer. The command is of the form:

```
          SAVE DEV:FILE.EX
```

If device is omitted, DSK: is assumed. If filename and extension are omitted, the current filename and extension are used.

The SAVE command also provides a convenient method for listing large programs quickly on the line printer rather than the terminal. For example:

```
          SAVE LPT:
```

lists the current program on the line printer.

9.8 THE RUN AND RUNNH COMMANDS

After a Industrial BASIC program has been typed and is in core, it is ready to be run. This is accomplished by simply typing the command:

```
        RUN
```

The program heading is printed and the program begins execution. If errors are encountered, appropriate error messages are typed on the keyboard; otherwise, the program runs to completion, printing whatever output was requested. When the END statement is reached, OS/8 Industrial BASIC stops execution and prints:

```
        READY
```

If the program does not run properly, or contains an infinite loop and, hence, will never stop, it may be terminated by typing CTRL/C, which returns control to the OS/8 Industrial BASIC editor (READY).

The RUNNH command suppresses printing of the heading and may be used in place of the RUN command.

NOTE

If a program that is not SAVEd is RUN, and for some reason is not retrievable by LISTing, the program may be retrieved by calling for OLD file BASIC.WS.

9.9 THE BYE COMMAND

The BYE command instructs the computer to exit from OS/8 Industrial BASIC and returns control to the OS/8 Keyboard Monitor. Typing a CTRL/C while in the OS/8 Industrial BASIC editor (READY) mode also returns control to the OS/8 Keyboard Monitor.

NOTE

Never type BYE before SAVEing a newly typed program. Unless the SAVE command is used, the program will be lost.

CHAPTER 10

FILES, FILE STATEMENTS AND CHAINING

10.1 GENERAL INFORMATION ON OS/8 INDUSTRIAL BASIC FILES

10.1.1 Resident Devices

The file capability provided by OS/8 Industrial BASIC allows the user to write information into (PRINT#) or read information from (INPUT#) files. Only I/O to devices which are part of the resident portion of the OS/8 Monitor may be done. The Resident Device handler includes the devices:

```
          SYS: and DTA1: for TD8E's
     and
          SYS: and RKB0: for RK8E's
```

All other current system device handlers contain only SYS:

10.1.2 File Descriptions

Files are referenced symbolically by a name of up to six alphanumeric characters followed, optionally, by a period and an extension of two alphanumeric characters. The extension to a file name is generally used as an aid for remembering the format of a file.

NOTE

All files operating under OS/8 Industrial BASIC are considered sequential access files. That is, they must be read or written sequentially, one item after another, from the beginning of the file.

10.1.2.1 Fixed Length Files

A fixed length file is one which is already in existence. That is, it has been created and CLOSEd.

The length of a fixed length file is equal to the number of blocks in the file and cannot be changed.

10.1.2.2 Variable Length Files

A variable length file is a newly created file. Until the file is CLOSEd, it is equal in length to the largest free space on the device.

When the file is CLOSEd it becomes a fixed length file equal in length to the actual number of blocks it occupies.

Unless the file is CLOSEd, the CHAIN, STOP or END statements will cause a loss of the file.

10.1.2.3 Numeric Files

Data in numeric files are stored as successive three-word floating-point numbers (85 to each OS/8 block) with the last word in each block unused.

10.1.3.4 ASCII Files

Data in ASCII files are stored in standard OS/8 format (three 8-bit characters to every two words). Refer to Chapter 11 of this manual and the OS/8 Software Support Manual (DEC-S8-OSSMA-A-D).

10.2 FILE STATEMENTS

OS/8 Industrial BASIC provides a number of statements for operating on files. They include:

FILE#

PRINT#

INPUT#

RESTORE#

CLOSE#

IF END#

The statements are distinguishable from other OS/8 Industrial BASIC statements by the number sign (#) which is appended to the statement.

10.2.1 The FILE# Statement

This statement is used to open a file and is of the form:

```
(line number) FILEtype numeric expression:
STRING EXPRESSION
```

where:

a. FILEtype is one of four possibilities:

FILEtype	Definition
FILE	Fixed length - ASCII
FILEV	Variable length - ASCII
FILEN	Fixed length - numeric
FILEVN	Variable length - numeric

b. The numeric expression has a value of one or two and represents the internal file number for the file being opened and is used in any other statements referencing this file.

c. The string expression is either a string variable or a string constant which has a value of the form:

```
DEV:FILE.EX
```

For example, the following program:

```
10 LET A$="DTA1:NETSAK.BA"
20 FILEN #1:A$
```

is equivalent to:

```
10 FILEN #1:"DTA1:NETSAK.BA"
```

for opening fixed length numeric file NETSAK on DECtape number 1 and assigning it internal file number 1. If DEV: is missing from the string expression, the device DSK is assumed by default.

NOTE

Only two files, (numbered 1 and 2) besides the terminal (FILE #0) may be open in a program at any time. However, the ability to open and close (CLOSE#) files under program control permits the user to access an unlimited number of files.

When selecting a FILEtype, the user should keep in mind that variable length files (FILEV and FILEVN) are restricted to outputting only. That is, variable length files should be used in conjunction with the PRINT# statement only. An attempt to read (INPUT#) from a variable length file results in error message VR. Only one FILEV or FILEVN may be active per device at a given time.

Fixed length ASCII files (FILE) should be restricted to use with the INPUT# statement, but may be used with the PRINT# statement when the user is certain that the ASCII or numeric data he is PRINTING (i.e., the number of characters) replaces, exactly, the ASCII or numeric data on the file. It is recommended that the use of fixed length ASCII files (FILE) for output be entirely avoided.

10.2.2 The PRINT# Statement

The PRINT# statement writes data into files and is of the form:

```
(line number) PRINT #N: list of expressions and delimiters
```

where N is the numerical expression for a file number. For ASCII files, the expressions in the list can be string or numeric, and the TAB and PNT functions can both be used. The delimiters can be commas or semicolons and have the same meanings that they have in the PRINT statement for the terminal (refer to Chapter 3). For numeric files, the expressions may be only numeric variables separated by commas or semicolons.

```
10 FILEV #1:"RKB0:DATE.DA"
20 LET F=1
30 PRINT #F: TAB(28);DAT$(X)
40 CLOSE #F
50 END
```

This routine prints the date, starting at column 28 on the device RKB0 in a file named "DATE.DA" .

10.2.3 The INPUT# Statement

The INPUT# statement reads data from files and is of the form:

```
(line number) INPUT #N: list of variables
```

where N is the numerical expression for a file number. The INPUT# statement does not expect a line number on each line of data in the file. If one is present, it is read as data:

```
10 DIM N$(19,15)
20 FILE #1:"LARRY"
30 FOR I=1 TO 19
40 INPUT #1: N$(I)
50 NEXT I
60 END
```

The above routine reads 19 strings from file LARRY.

NOTE

DSK: is the system device in this example.

The previous paragraph indicated that numbers may be written into an ASCII file. The reading of numbers from an ASCII file requires some precaution if the data delimiter is other than a comma or semicolon. In line 30 of the example below, the data written into file number 1 will be separated by a carriage-return and line-feed which are both written into the file. The subsequent reading of numbers from the file in line number 80 shows the use of a pair of dummy arguments (C and L) to compensate for the carriage-return and line-feed since they would otherwise be read as numeric data with a value of 0.

```
 10 FILEV #1:A$
 20 FOR I = 1 TO 10
 30 PRINT #1:I
 40 NEXT I
 50 CLOSE #1
 60 FILE #1:A$
 70 FOR I = 1 TO 10
 80 INPUT #1:J,C,L
 90 NEXT I
100 END
```

10.2.4 The RESTORE# Statement

The RESTORE# statement is of the form:

```
          (line number) RESTORE #
```

where N is a numerical expression for the file number to be reset to the beginning. If N is equal to zero, or if #N is missing from the statement, the DATA list in the program is reset to the beginning.

```
          10 FILE #2:"SUSAN"
          20 INPUT #2: A,B,C,D
          25 RESTORE #2
          30 INPUT #2: E,F,G,H
```

This program uses the same values from system file SUSAN for variables A, B, C, and D as it does for variables E, F, G, and H.

10.2.5 The CLOSE# Statement

The CLOSE# statement is of the form:

```
          (line number) CLOSE #N
```

where N is the numerical expression for the file number to be closed. For example:

```
          10 DIM A$(5,10)
          15 FOR I=1 TO 4
          20 LET A$(I) = "SHERRY" & CHR$(I+48)&".BA"
          25 FILE #1:A$(1)
          30 INPUT #1:B$
          35 IN B$="SANDY" THEN 60
          40 CLOSE #1
          45 NEXT I
          50 PRINT "CANNOT FIND SANDY"
          55 STOP
          60 PRINT "FOUND SANDY"
           .
           .
           .
           .
           .
           .
          90 END
```

This program searches through four system files (SHERY1 through SHERY4) for the file that has SANDY as the first entry. If the first entry in the file is not SANDY, the file is closed (statement 40) and the next file is opened.

NOTE

The user must CLOSE# all output files before ending the program in order to prevent the loss of data.

10.2.6 The IF END# Statement

The IF END# statement allows the user to determine whether or not there has been an End-of-File detected for the file in question. The statement has the form:

```
          (line number) IF END #N THEN line number
```

where N is a numerical expression for the file number. The line number must refer to a line in the program.

```
          .
          .
          .
          .
          100 IF END #1 THEN 170
          .
          .
          .
          .
          170 END
```

If the IF END# statement is found true, the last INPUT# or PRINT# (read or write) should be discarded.

The IF END# statement is not used to determine if more data is available, but rather to determine if the last read or write was valid.

The first attempt, in an INPUT# or PRINT# statement, to read or write past an end-of-file causes an abort of the I/O associated with that read or write, and the program passes control to the next operation to be performed. The second, and any subsequent attempts, to read or write past an end-of-file causes an RE or WE runtime error to be printed for each syntactical item in the INPUT# or PRINT# list. To avoid a lengthy list of error messages, avoid using long INPUT# or PRINT# lists in situations which may approach an end-of-file.

10.3 THE CHAIN STATEMENT

The CHAIN statement provides a convenient means for dividing large programs into a series of smaller programs which are written and stored separately, and executed in a chain. The CHAIN statement is of the form:

```
          (line number) CHAIN "DEV:Filename.extension"
```

When BASIC encounters a CHAIN statement in a program, it stops execution of that program, retrieves the program named in the CHAIN statement from the specified device and file, compiles the chained program and begins execution of the program. The use of the CHAIN statement, therefore, is the automatic equivalent of running an OLD program with no header (RUNNH). The file BASIC.WS will contain the original program in the chain and when execution is complete, the BASIC storage area will contain the original program.

Since BASIC removes the program which contains the CHAIN statement from core before retrieving the chained program, the user should make certain to CLOSE# all output files that are opened by FILE statements in the program which contains the CHAIN statement in order to avoid the loss of data generated by the program.

NOTES

1. Control commands OLD, RUNNH, and SAVE are described in Chapter 9.

2. If DEV: is not specified, DSK: is assumed by default.

CHAPTER 11

CREATING ASSEMBLY LANGUAGE FUNCTIONS

11.1 INTRODUCTION

OS/8 Industrial BASIC has a facility which allows experienced PDP-8 assembly language programmers to interface their own assembly language routines to OS/8 Industrial BASIC. This facility permits the user to add functions to OS/8 Industrial BASIC which can operate directly on special purpose peripheral devices. This chapter describes in some detail the organization and internal characteristics of the OS/8 Industrial BASIC Run-time System (INBRTS) and is intended to serve as a programming guide for the creation of such user-coded assembly language functions. This material assumes the user to be familiar with OS/8 and PDP-8 assembly language. For additional information on either subject, see the OS/8 SYSTEM REFERENCE MANUAL.

In addition to this chapter the programmer would find most useful a listing of the OS/8 Industrial BASIC Runtime System (DEC-S8-LBASB-A-LA).

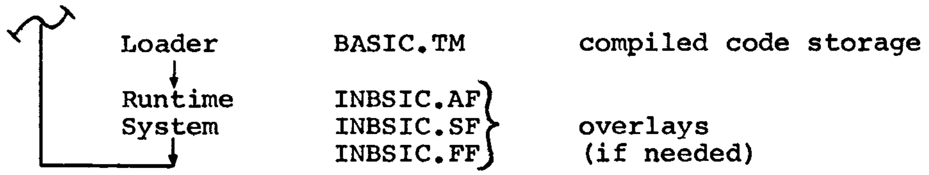
11.2 THE OS/8 INDUSTRIAL BASIC SYSTEM

The OS/8 Industrial BASIC system is divided into the following discrete parts:

1.	The BASIC editor	(INBSIC.SV)
2.	The BASIC compiler	(INBCMP.SV)
3.	The BASIC loader	(INBLDR.SV)
4.	The BASIC runtime system	(INBRTS.SV)
5.	The runtime system overlays	(INBSIC.AF)
		(INBSIC.SF)
		(INBSIC.FF)

The OS/8 Industrial BASIC editor is used to create and edit the source program. On receipt of a RUN command, the editor creates a temporary file called BASIC.WS, stores the source in that file, then chains to the compiler. The compiler compiles the program into a 12-bit pseudo-code which is loaded into core along with the runtime system by the loader. The run time system interprets each pseudo-instruction, calling each of the overlays into core as needed. On completion of the program, the runtime system chains back to the editor, and the cycle is repeated. Following is a diagram showing the files on the systems device associated with or used by each system component.

BASIC Component	Associated Files	Usage
→ Editor ↓	BASIC.WS	program storage
Compiler ↓	BASIC.WS	program storage
	BASIC.TM	compiled code storage



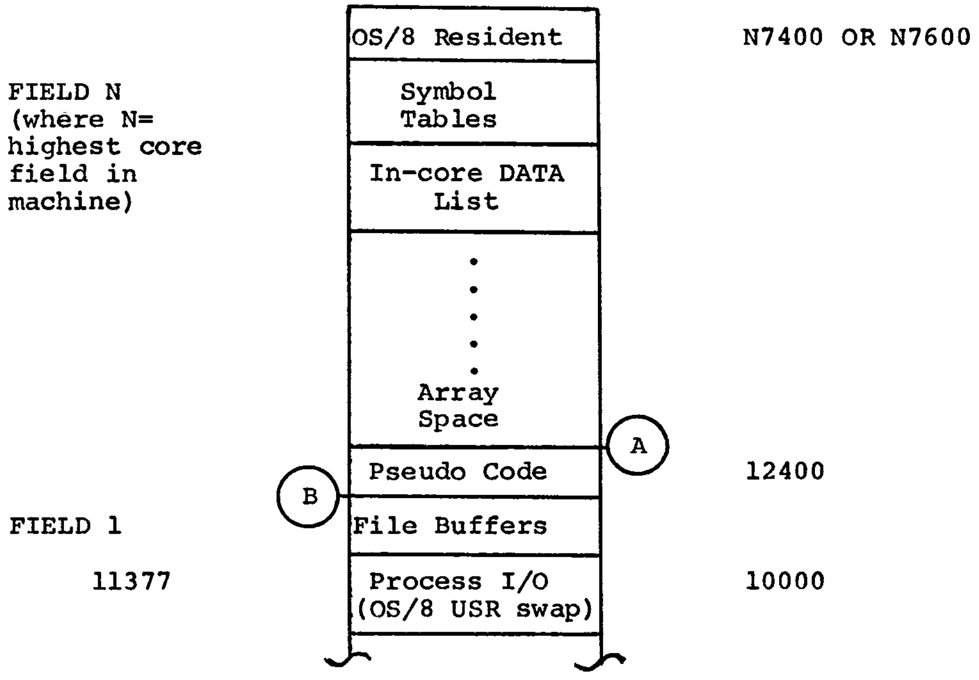
The user must avoid using the filenames above; they are reserved for the OS/8 Industrial BASIC System.

11.3 THE OS/8 INDUSTRIAL BASIC RUNTIME SYSTEM

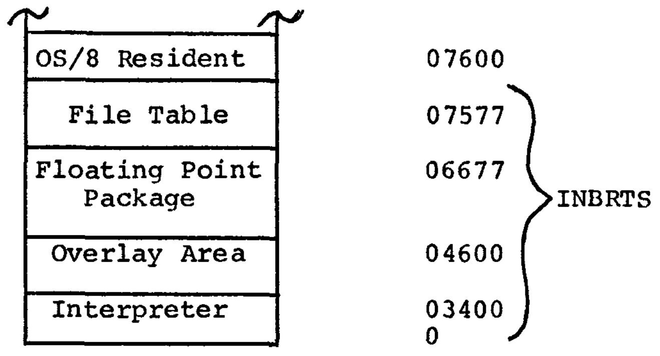
At the time the user's BASIC program is actually being executed, the portion of the BASIC system in control is the Run time System (INBRTS). INBRTS is also in core when user-coded functions are executed, and as such, a knowledge of it is essential to writing an OS/8 BASIC assembly language function. Note that the following sections refer frequently to specific core locations in INBRTS by symbolic names (always capitalized). The actual value of these symbols can be obtained from the symbol table (Appendix H) for the version of BASIC being used. Note that this symbol table is for a non-EAE system; the major routine entry points mentioned in this chapter, however, are the same for both systems. All diagrams in this chapter have the lowest core address at the bottom. This chapter also makes use of the variable names A, A(0,0), A$, and A$(0) to represent the general case. All references in this chapter to "page 0" refer to the INBRTS page 0 (Page 0, Field 0).

11.3.1 INBRTS Core Layout

When executing, INBRTS has the following configuration:



FIELD 0



The highest core field is used for INBRTS symbol tables, storing of the field 1 and field 2 (if non-ROM TD8 /E) resident portions of the OS/8 Monitor, the in-core DATA list (data generated by DATA statements in the program) and pseudo-code (generated by the compiler). The bottom of the array space (marked by line A) can exceed the field boundary and proceed into lower fields, but this will only happen for large programs. Note that if the bottom of the pseudo-code extends below line B (12400), one file buffer space must be sacrificed, with corresponding loss of runtime file capabilities. As the bottom of the pseudo-code approaches 12000, the number of files which may be simultaneously open at runtime approaches 0. At least 400(8) words of buffer must be free for each file opened at runtime. The file buffers are allocated dynamically at runtime in response to FILE commands in the BASIC program, and if not fully used may be used as buffer space by the user function.

11.3.2 INBRTS Overlays

Locations 3400-4577 of field 0 serve as an overlay area, into which the currently needed overlay is read. The overlays consist mainly of functions which are infrequently used, and are constructed as follows:

INBSIC.AF	Arithmetic Functions
	SIN, COS, ATN, EXP, FIX, FLOAT, INT, RND, SGN, SQR, LOG
INBSIC.SR	String Functions
	ASC, CHR$, DAT$, LEN, POS, SEG$, STR$, VAL, Error processing, TRC
INBSIC.FF	File Functions
	CHAIN, CLOSE, FILE

Note that the overlay driver reads in a new overlay only if the overlay currently resident does not contain the function specified in a given function call. If the function call is for a function which is found in the currently resident overlay, no overlay I/O takes place.

NOTE

On TD8E's interrupts are disabled during the I/O for overlays.

11.3.3 INBRTS Symbol Tables

INBRTS locates variables and strings at runtime via four permanently resident symbol tables. These tables, which always reside in the highest core field, are the Scalar Table (for variables like A, B1), the Scalar Array Table (A(1),B(1,1)), the String Symbol Table (A$, A1$), and the String Array Table, (B1$(2)). A more detailed description of the structure of these tables can be found in Section 11.5.

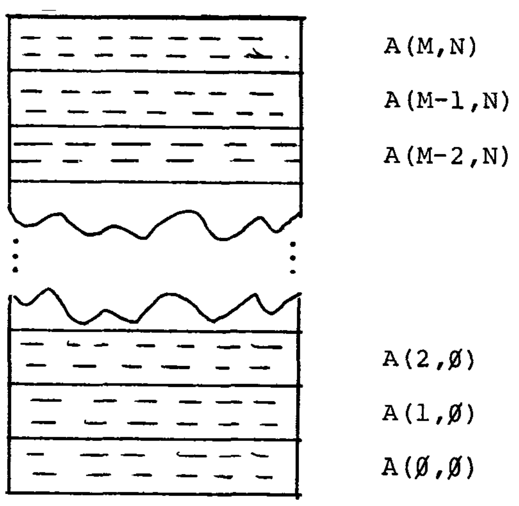
11.4 DATA FORMATS

11.4.1 Variables

Variables are stored in core as standard 3-word floating point numbers. The first word is a signed, 2's complement exponent, while the second and third words represent the signed, 2's complement mantissa.

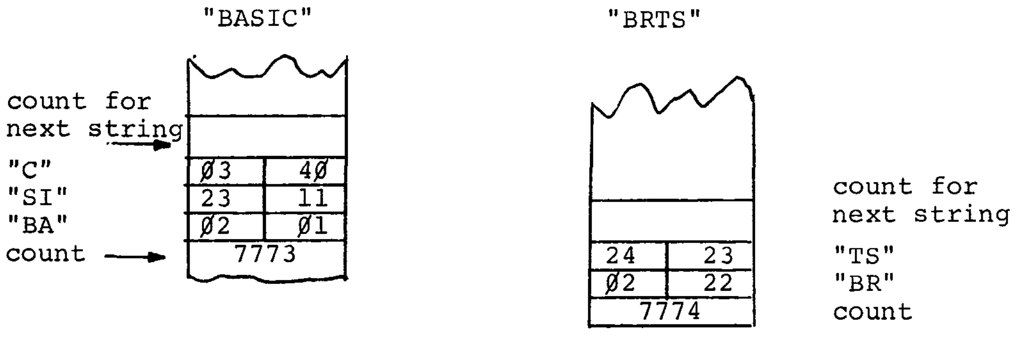
LOW MANTISSA
± HIGH MANTISSA
± exponent

Single variables are stored as 3-word entries in the Scalar Table. Arrays are stored in core as successive 3-word entries, with the first subscript varying the fastest, and A(0,0) occupying the lowest core address. The address and field of A(0,0) are specified in the Scalar Array Table. The structure of the symbol tables is described in Section 11.5.



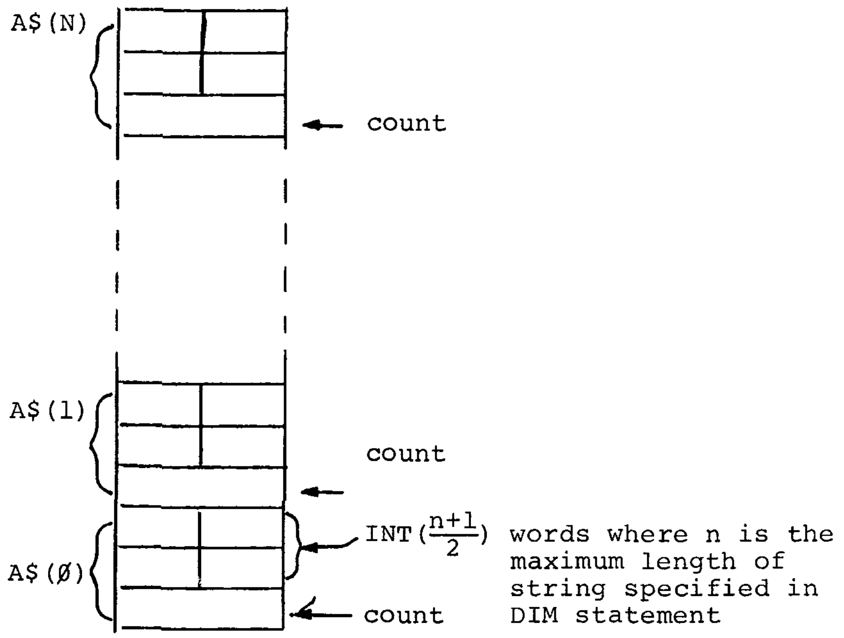
11.4.2 Strings

Strings are stored as 6-bit ASCII characters, with a character count as the first word of the string. The left half of each character word is used first, with unused characters padded with spaces (40(8)). The character count is a signed, 2's complement number representing the actual number of characters in the string, not the number of words devoted to that string. Each string is allocated [INT($\frac{n+1}{2}$)+1] words, where n is the maximum length specified in a DIM statement, whether that many words are actually used or not.



The minimum string is one character long. The address of the count word for each string is pointed to by its entry in the String Symbol Table.

String arrays are stored as successive strings, with A$(0) occupying the lowest core address. Each string is allocated enough space for its maximum length, even though all of this space may not be used.



A$(N)
count
A$(1)
count
A$(Ø)
INT($\frac{n+1}{2}$) words where n is the maximum length of string specified in DIM statement
count

NOTE

For any of the above data types, a field boundary may fall anywhere within any individual item. Routines that use successive words in any data item must be careful to check for a field boundary within that item.

11.4.3 In-core DATA list

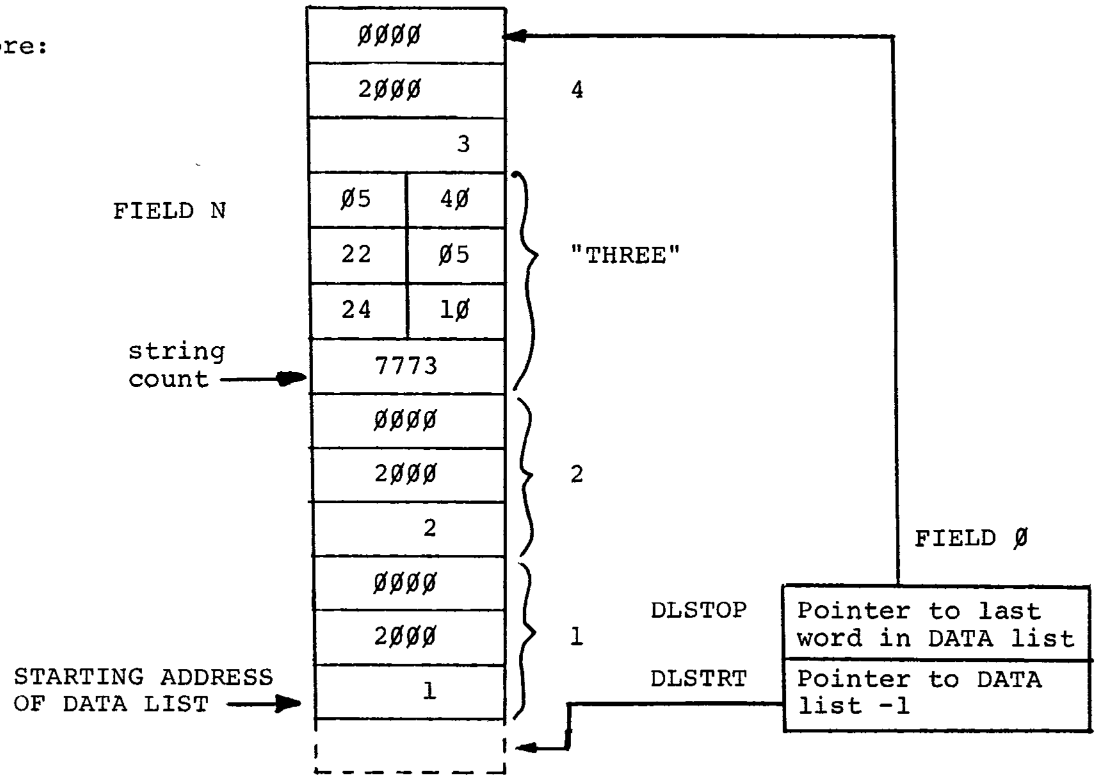
The in-core DATA list is stored as sequential data items in core. Strings again are devoted even numbers of words, and are prefixed by a count. There is no separator or identifier of DATA items and the DATA list is always in the highest core field. A page 0, field 0 pointer to the starting address of the DATA list less 1 is maintained at DLSTRT, and the address of the last word of the list can be found at DLSTOP.

Example:

In BASIC:

```
     DATA    1,2,"THREE", 4
```

In core:



11.4.4 The String Accumulator (SAC)

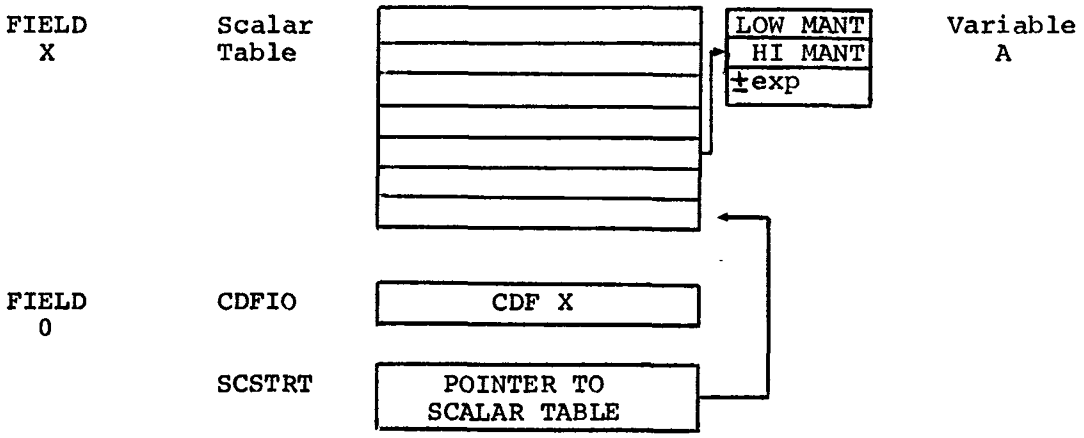
All INBRTS string operations use the String Accumulator (SAC) as one of the operations, and the result is always left in the SAC. The string accumulator is to strings as the hardware AC is to PDP-8 instructions. The SAC starts at location SAC for 36 words (72 characters), and the length of the string currently in the SAC is stored as a negative number in STRLEN. A page 0 pointer to the start of the SAC less 1 is maintained at SACPTR.

11.5 INBRTS SYMBOL TABLE STRUCTURE

The INBRTS symbol tables all reside in the highest core field. A CDF to the symbol table field can be found in location CDFIO of field 0.

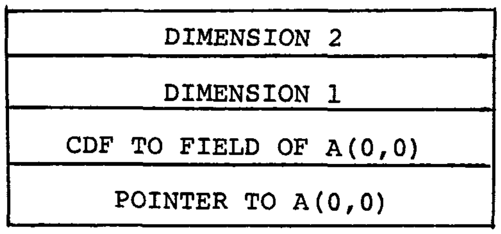
11.5.1 The Scalar Table

The Scalar Table is the highest symbol table in core, and it consists of successive 3-word entries, each entry containing a 3-word floating-point number. One entry exists for each variable used in the program, and a few extra entries are used as temporaries. A pointer to the start of the Scalar Table can be found at location SCSTRT of field 0. The scheme for scalar variables is as follows:



11.5.2 The Array Symbol Table

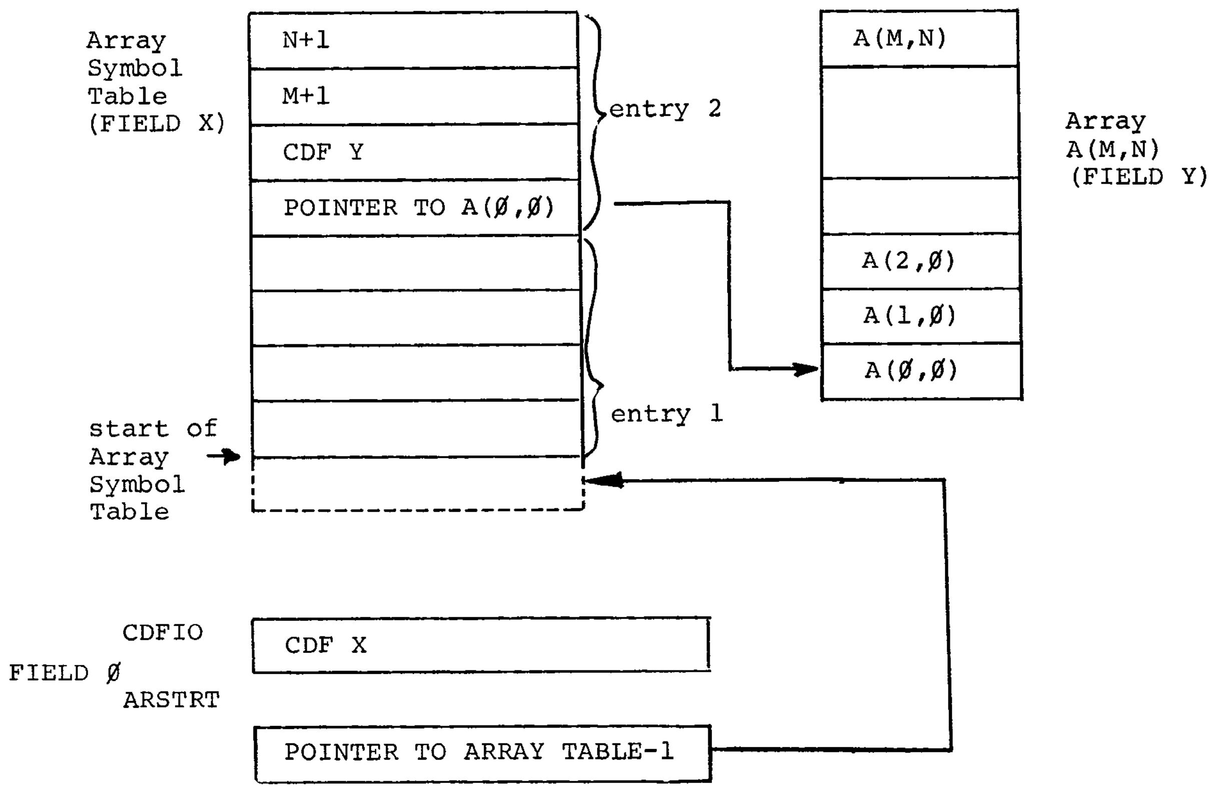
The Array Symbol Table consists of successive 4 - word entries, each entry specifying the location and size of an array used in the program. Each entry is as follows:



The first word of each entry is a 12-bit pointer to the location of the exponent word of the first element in the array. The second word is a CDF N where N is the field for the pointer in the first word. The third word is the first dimension of the array (obtained by adding 1 to the M in a DIM A(M,N) statement because the first subscript is always 0), and the last word is the second dimension of the array (obtained by adding 1 to the N in the aforementioned DIM statement for the same reason). If the array is uni-dimensional, the second dimension is zero. To locate the nth element in the array, INBRTS performs the following calculation:

Addr of A(M,N)=3*[M+(DIM1+1)*N]+Addr of A(0,0)

A pointer to the start of the Array Symbol Table less 1 (for use in an index register) can be found in field 0 at location ARSTRT. The scheme for arrays is:



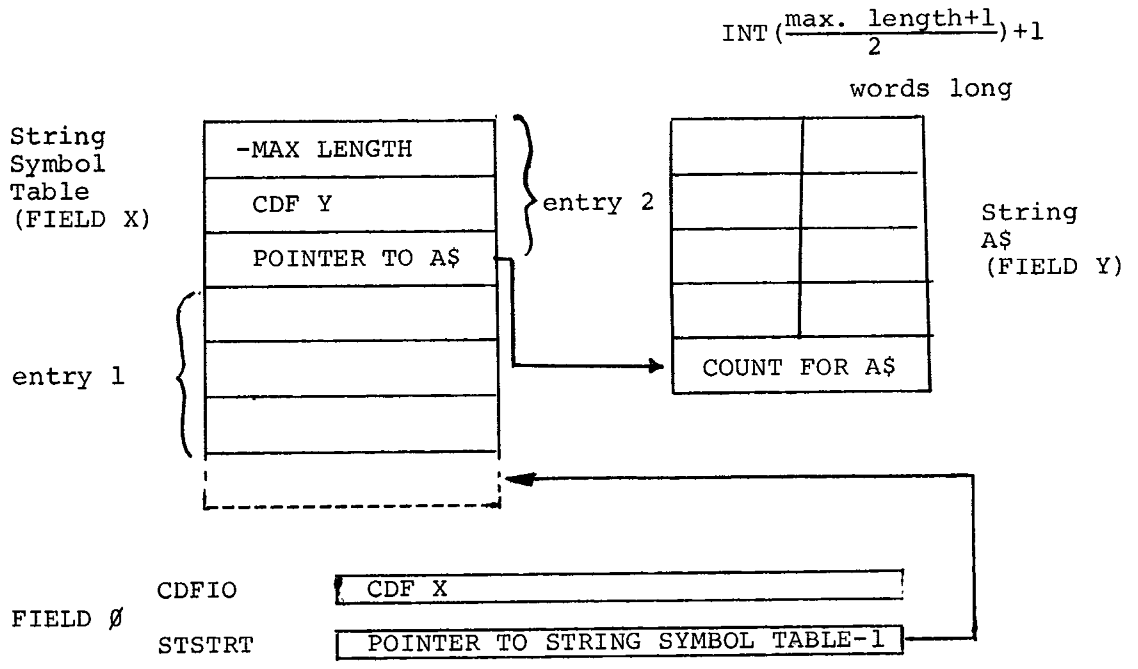
11.5.3 The String Symbol Table

The String Symbol Table has successive three-word entries as follows:

-MAXIMUM # OF CHARS IN STRING
CDF FOR STRING
POINTER TO STRING

The first word is a 12-bit pointer to the count word of the string. The second word of each entry is a CDF for that count word, and the third word of the entry is the maximum length of the strinq (in number of characters) stored as a 2's complement negative number. A pointer to the start of the String Symbol Table (less 1) can be found in field 0 location STSTRT. Note that the maximum number of characters in the string represents the amount of space allocated for the string; the amount of space actually used is represented by the count word which is stored with the string.

The scheme for simple strings is:



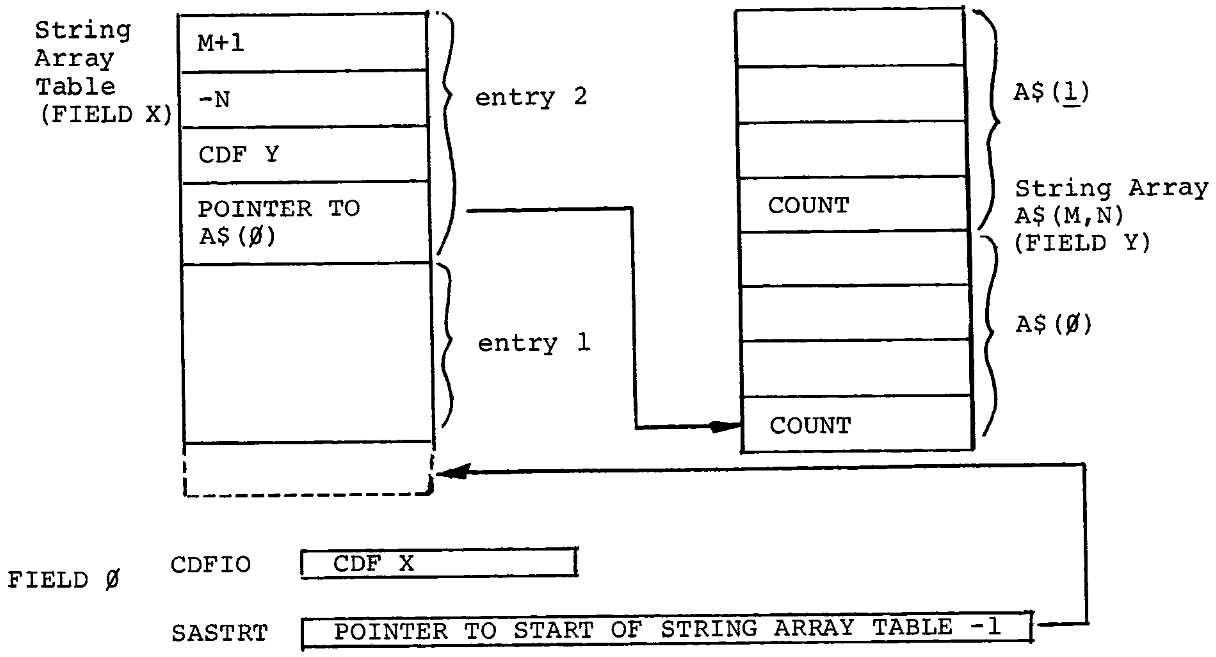
11.5.4 The String Array Table

The String Array Table consists of consecutive 4-word entries, with each entry as follows:

DIMENSION OF A$(0)
-MAXIMUM #OF CHARS IN A$(0)
CDF FOR A$(0)
POINTER TO A$(0)

The first word contains a pointer to the count word of string A$(0), and the second word contains a CDF for this count. The third word has the maximum length (in characters) of each element in the array stored as a 2's complement negative number. The last word contains the dimension of the string array, obtained by adding 1 to the M in a DIM statement of the form DIM A$(M,N) because the first element is always A$(0). A pointer to the start of the String Array Table less 1 can be found in field 0 at location SASTRT.

The scheme for string arrays is:



String Array Table (FIELD X)
M+1
-N
CDF Y
POINTER TO A$(Ø)
entry 2
entry 1
COUNT
COUNT
A$(1)
String Array A$(M,N) (FIELD Y)
A$(Ø)
FIELD Ø
CDFIO
CDF X
SASTRT
POINTER TO START OF STRING ARRAY TABLE -1

To locate the nth element of the string array, INBRTS performs the following calculation:

$$\text{addr of A\$(N)} = \text{addr of A\$(0)} + \left(\text{INT}\frac{(\text{ABS}(Z)+1)}{N}+1\right)*N$$

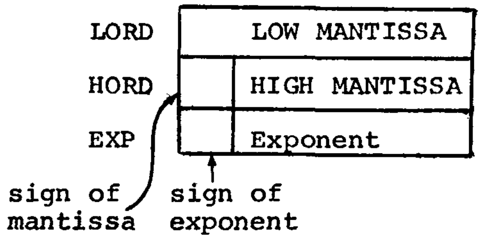
where Z = individual element length.

11.6 FLOATING-POINT PACKAGE

The INBRTS floating-point package is permanently resident, and as such it is readily available for use by assembly language routines for floating-point calculations.

11.6.1 Floating-Point Accumulator

One of the operands of every floating-point operation is the Floating Accumulator (FAC), and the result of all floating-point operations (except FPUT) is always left in the FAC. The FAC is found at EXP HORD, and LORD on page 0 with standard PDP-8 23-bit floating-point format:



The floating-point accumulator is to floating-point instructions what the hardware accumulator is to PDP-8 machine language instructions.

11.6.2 Floating-Point Routines

The following floating-point routines are available for user subroutine use:

Function	Starting Address	Operation
ADD	FFADD	FAC←FAC+OPERAND
SUBTRACT	FFSUB	FAC←FAC-OPERAND
MULTIPLY	FFMPY	FAC←FAC*OPERAND
DIVIDE	FFDIV	FAC←FAC/OPERAND
INVERSE SUBTRACT	FFSUB1	FAC←OPERAND-FAC
INVERSE DIVIDE	FFDIV1	FAC←OPERAND/FAC
LOAD FAC	FFGET	FAC←OPERAND
STORE FAC	FFPUT	OPERAND←FAC

The symbol "←" means "is replaced by".

Note that the store function (FFPUT) is the only operation in which the result is not left in the FAC Note also that FFPUT is a non-destructive store, i.e., the FAC is the same after the store operation as before.

There are two calling sequences for the floating-point routines, each with a different method for passing the address of the operand. Mode 1 is the most efficient, and can be used whenever the operand is in field 0. Mode 2 is the field independent call, but is more core expensive than mode 1.

The mode being used is determined as follows:

1. If the contents of the AC is non-zero on entry, the mode used is mode 2.
2. If the contents of the AC is zero on entry, the location FF is examined. If FF is also zero, mode 1 is the calling mode. If FF is non-zero, mode 2 is used.

The calling modes are as follows:

Mode 1 - address of operand follows call to floating-point routine.

```
        CLA
        DCA FF                /SWITCH FF=0 FOR MODE 1
        JMS I POINTER           /JUMP TO FLOATING-POINT ROUTINE
        (operand address)       /12 BIT ADDRESS OF OPERAND
              .
              .
              .
    POINTER,(starting address)
                                /FLOATING-POINT ROUTINE
                                /STARTING ADDRESS.
```

Mode 2 - address of operand in AC on call to floating-point routine.

```
        CLA IAC
        DCA FF                  /FF SWITCH NOT EQUAL TO 0 FOR
                                /MODE 2
        CDF N                   /DF TO FIELD OF OPERAND
        TAD OPADDR              /ADDRESS OF OPERAND
        JMS I POINTER           /JUMP TO FLOATING-POINT ROUTINE
        (unused)                /THIS LOCATION UNUSED
                                /RETURNS HERE.

    POINTER, (starting address) /ADDRESS OF FLOATING-POINT ROUTINE
    OPADDR, (operand)           /ADDRESS OF OPERAND
```

Both modes return with a clear AC and the data field set to 0. Note that the switch FF is not altered by the routines themselves, hence it is only necessary to set it when desired to change modes, not before every call.

The mode 2 call always returns to the second instruction following the JMS call, skipping the word following the JMS. Since this word is completely unused, it is a good location for constant storage.

The FF switch is necessitated by the special case when it is desired to reference an operand located at location 0 in a field other than field 0. If the FF switch were not present, the floating-point package would examine the AC, find it empty, and use the address in the word following the call, since there is no way of distinguishing an empty AC from an operand address of 0 loaded into the AC. The FF switch, then, is used to tell the floating-point package whether the zero AC means "mode 1 call" or "operand at 0".

INBRTS maintains links for FGET and FPUT on page 0 of field 0, providing convenient access to these frequently used routines.

Page 0 Link Name	Routine Linked
FGETL	FFGET
FPUTL	FFPUT

Examples:

Some examples of INBRTS floating-point code:

1. Routine to calculate X↑2+2X+1

```
              .
              .
              .
             CLA
```

```
                DCA FF              /OPERAND ADDRESS WILL
                                    /FOLLOW CALLS (MODE 1)
                JMS I FGETL         /LINK IS ON PAGE 0
                  X
                JMS I FMPYLK        /X * X
                  X
                JMS I FPUTL         /SAVE X↑2
                  Y
                JMS I FGETL         /LOAD X AGAIN
                  X
                JMS I FMPYLK        /2X
                  TWO
                JMS I FADDLK        /2X+1
                  ONE
                JMS I FADDLK        /X↑2+2X+1
                  Y
                  .
                  .                 /RESULT NOW IN FAC
                  .
FADDLK,       FFADD                 /LINK TO ADD ROUTINE
FMPYLK,       FFMPY                 /LINK TO FLOATING MULTIPLY
TWO,          0002                  /FLOATING POINT CONSTANT
              2000                  /2.0
              0000
ONE,          0001                  /FLOATING POINT CONSTANT
              2000                  /1.0
              0000
X,            ...                   /VARIABLE
              ...
              ...
Y              0                    /FLOATING-POINT TEMPORARY
               0
               0
```

2. Routine to add 5 successive floating-point numbers starting at location 0 of field 2.

```
START,        CLA                   /
              DCA OPADDR            /FIRST OPERAND AT LOCATION 0
              JMS I FCLR            /ZERO FAC
              IAC
              DCA FF                /CALLS ARE MODE 2
ALOOP,        CDF 20
              TAD OPADDR            /OPERAND ADDR IN AC
              JMS I FADDLK          /CALL ADD ROUTINE
MINUS5,          -5                 /LOCATION UNUSED, SO WE USE
              TAD OPADDR            /IT AS A COUNTER
              TAD K3                /UPDATE OPERAND ADDRESS
              DCA OPADDR
              IAZ MINUS5            /DONE?
              JMP ALOOP             /NO
              HLT                   /YES-ANSWER IN FAC.
FADDLK,       FFADD                 POINTER TO ADD ROUTINE
OPADDR,         0                   /POINTER TO OPERAND
K3,             3                   /EACH OPERAND IS 3 WORDS LONG.
```

11.6.3 Floating-Point Operations

There are also four simple floating-point operations that operate on the FAC and are available to user subroutines.

Function	Starting Address	Operation
NEGATE	FFNEG	FAC←-FAC
NORMALIZE	FFNOR	NORMALIZE←FAC
SQUARE	FFSQ	FAC←FAC*FAC
CLEAR	FACCLR	FAC←0

These functions are all called by simple JMS, and return with the hardware AC=0. Page 0 Links are maintained for negate, normalize, and clear.

Page 0 Link	Routine
FNEGL	FFNEG
FNORL	FFNOR
FCLR	FACCLR

11.7 INBRTS SUBROUTINES

There are several subroutines in INBRTS which can be useful to assembly language functions. A discussion of each of these routines follows. They are identified in the discussion by the tag for their starting address, and all tags referred to can be found in the symbol table.

11.7.1 Subroutine ARGPRE

Subroutine ARGPRE is used to locate scalar Table. When called, it uses the rightmost 8 bits (0-225 decimal) of location INSAV as the entry number to be found, and on return, the data field is set to the field of the variable and the AC points to the exponent word of the variable. ARGPRE is called via a JMS, and is used most often in passing arguments to and from the user subroutine. (See Section 11.8)

Example: Load the FAC with the third variable in the Scalar Table.

```
            CLA
            TAD C2              /WE WANT ENTRY #3, BUT
                                /SINCE THE FIRST ONE IS 0,
                                /LOAD INSAV WITH 2
            DCA INSAV
            IAC
            DCA FF              /SET FF SWITCH
            JMS I ARGPRL        /CALL ARGPRE
            JMS I FGETL         /THE AC AND DATA FIELD
            (unused)            /ARE SET, SO THIS IS A
            HLT                 /MODE 2 CALL.

C2,         2
ARGPRL,     ARGPRE
```

11.7.2 Subroutine XPUTCH

Subroutine XPUTCH is used to put ASCII characters into the terminal ring buffer. When called, the 8-bit ASCII character is in the rightmost 8 bits of the AC. On return, the AC is cleared. Note that unless the ring buffer is empty, XPUTCH does not cause any characters to be printed; it merely places the character in the terminal ring buffer. If the ring buffer is full the system will wait until it can place the character in the ring buffer. A page 0 link to XPUTCH is maintained at location XPUT.

Example: Put a carriage return/line feed combination in the terminal buffer.

```
                CLA             /LOAD CR INTO AC
                TAD D215        /CALL XPUTCH VIA PAGE 0 LINK
                JMS I XPUT      /LOAD LINE FEED INTO AC
                TAD K212        /PUT IN BUFFER
                JMS I XPUT
                 .
                 .
                 .
    K215,       215             /ASCII CODE FOR CR
    K212,       212             /ASCII CODE FOR LF
```

11.7.3 Subroutine PSWAP

Under normal conditions, INBRTS runs with the OS/8 page 17600 portion of the resident monitor moved to the highest page of core (second highest page if TD8/E system). PSWAP is used to swap this page back and forth prior to doing any operations with OS/8. Prior to calling OS/8, PSWAP should be used to restore the page 17600 resident to 17600, and when OS/8 operations are complete, PSWAP should be called again to swap the 17600 resident back up to high core. A page 0 link to PSWAP is maintained at location P1SWAP.

Example: The following code uses the USR in OS/8 to perform a LOOKUP on the file BASIC.DA on the systems device.

```
        .
        .
        .
        CLA             /AC SHOULD BE 0 ON CALL
        JMS I P1SWAP    /RESTORE OS/8 PAGE 17600 RESIDENT
        CLA IAC         /DEVICE # FOR SYS: IS 1
        CIF 10
        JMS I K7700     /CALL USR
        2               /LOOKUP
        FNAME           /POINTER TO FILE NAME
        0               /CONTAINS LENGTH ON RETURN
        HLT             /ERROR RETURN
        JMS I P1SWAP    /SWAP OS/8 RESIDENT BACK
        .               /TO HIGH CORE
        .
        .
        .
```

NOTE

If PSWAP is used, it must be executed an even number of times. When the assembly language function is called, the page 17600 resident is at high core; when the function returns to INBRTS, the 17600 resident must be back in high core. On TD8E systems interrupts should be disabled before calling PSWAP and reenabled after last call.

11.7.4 Subroutine UNSFIX

Subroutine UNSFIX is used to fix a positive, 12-bit, magnitude only integer from the FAC and return with the result in the hardware AC. The range of the fixed integer is 0-4095; an attempt to fix a number larger than 4095 or a negative number will cause an "FO" or "FM" error, respectively. UNSFIX is called via simple JMS, and a page 0 link to UNSFIX is maintained, called INTL. UNSFIX destroys the contents of the FAC.

Example: The following code uses the FAC1 as a count of the number of times to ring the bell on the terminal.

```
              .
              .
              .
              CLA
              JMS I INTL          /FIX THE FAC TO 12-BIT INTEGER
              CIA                 /NEGATE THE INTEGER
              DCA COUNTR          /AND STORE AS COUNT
BELLOP,       TAD K207            /ASCII FOR BELL
              JMS I XPUT          /PUT IN RING BUFFER
              ISZ COUNTR          /RIGHT NUMBER YET?
              JMP BILLOP          /NO-RING ANOTHER BELL
              .
              .
              .
K207,         207
```

11.7.5 Subroutine STFIND

Subroutine STFIND is used to locate a string variable or the first element of a string array. When called, if the link is non-zero, STFIND looks for an entry in the String Array Table. If the link is zero, STFIND uses the String Symbol Table. For standard string variables, the rightmost 8 bits of location INSAV are used as the number of the entry to be obtained; for string array variables the last 5 bits are used. On returns from STFIND, the AC contains a CDF to the field of the string specified, location STRPTR points to the first word (count word) of the string, location STRMAX holds the maximum length of the string (as a negative number), and location STRCNT contains the actual number of characters in the string (as a negative number). STFIND is used most often in passing arguments to and from user functions.

Examples:

1. To find string number 7

```
                TAD K6          /THE NUMBERING STARTS WITH 0
                DCA INSAV       /SET UP STFIND POINTER
                CLL             /WE WANT SIMPLE STRING
                JMS I STFINL    /CALL STFIND
                .
                .
K6,             6
STFINL,         STFIND
```

2. To find the first element of string array number 2.

```
                TAD K1          /THE SECOND ENTRY
                DCA INSAV
                CLL CML         /WE WANT STRING ARRAY
                JMS I STFINL    /CALL STFIND
                .
                .
K1,             1
STFINL,         STFIND
```

11.7.6 Subroutine BSW

Subroutine BSW is used to swap the two halves of the hardware AC. BSW is called by a simple JMS, and a page 0 link called BSWL is maintained.

11.7.7 Subroutine MPY

Subroutine MPY is a 12-by-12-bit binary multiply routine. The AC is multiplied by the contents of location TEMP3 (both numbers are treated as 12-bit, unsigned integers), and on return, the high-order bits of the result are in TEMP6, and the low-order bits of the result are in the AC. The page 0 line to MPY is MPYLNK.

11.7.8 Subroutine DLREAD

Subroutine DLREAD is used to read the next word of the incore DATA list into the AC. If there is no more data in the DATA list, a DA error message results.

Example: Read the next number from the DATA list into the FAC.

```
                CLA
                JMS I DLREAL    /READ EXPONENT WORD INTO AC
                DCA EXP         /STORE IN FAC
                JMS I DLREAL    /READ HIGH MANTISSA FROM LIST
                DCA HORD        /STORE HIGH MANTISSA WORD
                JMS I DLREAL    /READ LOW MANTISSA FROM LIST
                DCA LOSR        /STORE LOW MANTISSA WORD
                .
                .
                .
DLREAL,         DLREAD
```

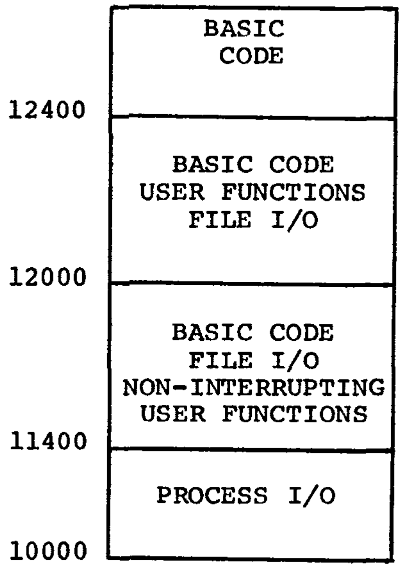
11.7.9 Subroutine ABSVAL

Subroutine ABSVAL is used to take the absolute value of the FAC. If the FAC is positive, ABSVAL is essentially a NOP; if the FAC is negative, it is negated before return.

11.8 PASSING ARGUMENTS TO THE USER FUNCTION

INBRTS calls the user assembly language function with a JMP instruction. Prior to executing that JMP, it places the first numeric argument in the FAC, the second in Scalar Table entry 0, the third in Scalar Table entry 1, etc., until the argument list is satisfied. If any string arguments are used, the first is found in the SAC and the second is pointed to by String Table entry 0. The user function obtains these arguments as needed by calling the routines ARGPRE and STFND appropriately. All user functions occur in FIELD 1.

FIELD 1 core usage is as follows:


BASIC CODE
12400
BASIC CODE USER FUNCTIONS FILE I/O
12000
BASIC CODE FILE I/O NON-INTERRUPTING USER FUNCTIONS
11400
PROCESS I/O
10000

The process I/O area which extends up to 1400 in field 1 is used by the real-time functions. The area from 1400 to 1777 in field 1 is a sharable area: If any file I/O is done, this area will be allocated first for the buffer space; if no file I/O is done, this area may be used for non-interrupting user functions. Finally, if no file I/O or non-interrupting user functions are being used, BASIC interpretive code can extend into these locations.

The area from 2000 to 2377 in field 1 is another sharable area: If 2 files are needed at runtime, this will be the second area allocated; if this area is not needed for 2 files, large BASIC programs can use this area for extended core, or it may be used as a resident area for any types of user functions.

Users adding their own functions to the BASIC Run-Time System have to provide information to INBRTS so it may handle core correctly. This will be detailed later.

11.8.1 Interfacing FIELD 1 Code with FIELD 0 Subroutines

For convenience a routine resides in FIELD 0 (CALLF0) which will accept the word following the JMS as the FIELD 0 subroutine to call. An argument can be passed to the subroutine in the AC. Return to FIELD 1 occurs at argument +1.

The sequence looks as follows:

In FIELD 1

```
          TAD AC
          CIF 0
          JMS I [CALLF0
          subroutine to call
          return occurs here with Data FIELD=1
```

The following function takes the first two numeric arguments and performs the operation on them specified in A$:

```
          UDEF EXM(X,A$,Y)
          LET Z=EXM (2,"PLUS",1)
```

Legal values for A$ are strings beginning with "PL" for "PLUS" and "MI" for "MINUS".

If the function is to return any value, that value should be left in the FAC on return. The user function must always return by a JMP I ILOOP in FIELD 0 with the data FIELD 0.

To generate a fatal IA (illegal argument) error message, perform a JMP to location IA in INBRTS.

```
EXM,                               /ENTRY POINT, DATA FIELD=0

        DCA I (INSAV               /INITIALIZE FOR ARG 0
        TAD I F1SACP               /GET FIRST 2 CHARS OF A$ FROM SAC
        TAD PL
        SZA CLA
        JMP EMINUS
        CIF 0
        JMS I (CALLF0
        ARGPRE
        CIF0
        JMS I (CALLFO
        FFADD
RETURN, CDF CIF 0
        JMP I (ILOOP

EMINUS, TAD I F1SACP
        TAD MI
        SNA CLA
        JMP ISMINS
        CIF CDF 0
        JMP I (IA
ISMINS, CIF 0
        JMS I (CALLF0
        ARGPRE
        CIF 0
        JMS I (CALLFO
        FFSUB
        JMP RETURN

PL,     -2014
MI,     -1511
```

11.8.2 Using the USE Statement

If the assembly language function needs to know the location of an array (for buffer space, multiple argument passing, array argument), the USE statement is necessary. The USE statement places the octal number for the array specified into location USECON. By using this value as an index into the Array Symbol Table, the array specified can be located and used by the assembly language function, as necessary.

For example: The hypothetical assembly language function PLT requires a 100 - word buffer. To assure allocation of this buffer, the BASIC user of PLT is instructed to dimension a 34-element array and use it in a USE statment before calling the PLT function.

In BASIC:

```
10  REM DEFINE THE USER FUNCTION
20  UDEF PLT (X,Y)
30  REM ALLOCATE A 34 ELEMENT (102 WORDS) ARRAY FOR A BUFFER
40  DIM B(34)
        .
        .
        .
100 USE B
110 Y=PLT(3,2.8)
        .
        .
        .
```

The function PLT finds B as follows:

```
PLT,    TAD USECON      /GET ENTRY NUMBER OF B
        CLL RTL         /MULTIPLY BY 4 (EACH ARRAY TABLE
                        /ENTRY IS 4 WORDS LONG)
        TAD ARSTRT      /MAKE POINTER INTO ARRAY TABLE
        DCA XR5         /AND SAVE IT
        TAD CDFIO       /GET CDF TO SYMBOL TABLE FIELD
        DCA .+1         /PUT INTO LINE
        .               /CHANGE DF TO SYMBOL TABLE FIELD
        TAD I XR5       /GET POINTER TO B(0)
        DCA BPTR        /SAVE FOR LATER
        TAD I XR5       /GET CDF FOR B(0)
        DCA BCDF        /SAVE FOR LATER
        TAD I XR5       /GET ARRAY DIMENSION 1
        DCA DIM1
        TAD I XR5       /GET ARRAY DIMENSION 2
        DCA DIM2
            .
            .
```

Note that the USE statement merely passes an entry number to the assembly language function; all actual parameters must be obtained from the Array Symbol Table using that entry number as an index. Note also that the physical location of arrays passed in such a fashion can be almost anywhere in core, and a field boundary may fall within the array.

11.9 INBRTS I/O

11.9.1 Terminal I/O

INBRTS drives the terminal asynchronously by maintaining a character terminal output buffer and using interrupts. The procedure is as follows:

1. Characters are inserted into the terminal ring buffer by calling subroutine XPUT. If the ring buffer is full, XPUT waits until a character is printed and a slot is free.

2. INBRTS prints a character from the ring buffer whenever the flag comes up.

Assembly language functions are free to use the ring buffer via XPUT.

11.9.2 INBRTS File Formats

BASIC files are formatted as follows:

1. Numeric files - Numeric files are formatted as consecutive 3-word floating-point numbers,85 to each 256-word OS/8 block. The last word in each block is unused. There is no end-of-file marker.

2. ASCII Files - ASCII files are stored in OS/8 ASCII format, that is, three 8-bit characters packed to every two words as follows:

0 3	4 11
HI ORDER CHAR 3	CHAR 1
LO ORDER CHAR 3	CHAR 2

The end of the file is marked with a CTRL/Z character.

11.9.3 INBRTS Buffer Space

Locations 11400-12377 in INBRTS are devoted to file buffer space. Buffers are allocated as they are needed, the lowest free buffer always being allocated first. A map of currently allocated buffers is maintained on page 0, called BMAP. Bits in the map are on if the buffer is allocated, off if the buffer is free. Bit 11 represents the buffer from 11400-11777, bit 10 for 12000-12377. If any of the buffers are not available because the pseudo-code or variable space extends below 12400, the corresponding BMAP bits are set when INBRTS is started.

11.9.4 INBRTS Device Driver Space

The only available device drivers are the resident device drivers. No other space is allocated for device drivers. On TD8E systems units 0 and 1 are resident, on RK8E systems all of unit 0 is resident.

Note that assembly language functions that are used in programs which do not require more than one file open at once may wish to use some of the buffer space for their own purposes. This space can be allocated by setting appropriate bits in BMAP, by modifying INBRTS initialize code (follows TAG TTYBUF). After the bits are set, INBRTS will not use this space in subsequent FILE commands.

11.9.5 The INBRTS I/O Table

INBRTS keeps track of the status of each of the files which may be open simultaneously by means of the I/O table. Starting at FILE1, it has two 13-word entries, labeled FILE1 and FILE2. Each name corresponds to the number specified in the file statement which opened that file, and the format of each entry is as follows:

```
HEADER WORD
POSITION OF PRINT HEAD (FOR COLUMN FORMATTING) OR DEVICE NAME
MAXIMUM FILE LENGTH OR DEVICE NAME
FILE NAME
FILE NAME
FILE NAME
FILE NAME
STARTING ADDRESS OF BUFFER (IN FIELD 1)
CURRENT BLOCK IN BUFFER
READ/WRITE POINTER INTO BUFFER
HANDLER ENTRY POINT
STARTING BLOCK NUMBER FOR FILE
ACTUAL FILE LENGTH
```

The header word bits have significance as follows:

Bit Positions	Meaning
0-3	OS/8 number for device
4-5	Current character number for unpacking ASCII files
6	0 if the current buffer load has not been changed 1 if the current buffer load has been altered

7	0 if device is file structured
	1 if device is read/write only
8	0 if the handler is 1 page long
	1 if it is a 2-page handler
9	0 if file is fixed length
	1 if variable length
10	0 if more data in file
	1 if EOF has been seen
11	0 if file numeric
	1 if file ASCII

11.10 INTERFACING THE ASSEMBLY LANGUAGE FUNCTION TO INBRTS

All assembly language functions are routines, called by a JMP through the User Function Table. This table, which begins at location 1560 in INBRTS, contains absolute pointers to the starting addresses of each of the user assembly language functions. User functions must be origined to run in FIELD 1, and must return to INBRTS via a JMP I ILOOPL in FIELD 0. To interface a set of user functions to INBRTS, perform the following operations:

1. Assemble all the user assembly language functions; the entry to the functions must be in FIELD 1. Entry is with the data FIELD set to 1.

   FIELD 1 PAGE 0 scratch locations and useful FIELD 0 pointers are given in Section 11.11.3.

   ```
   .R PAL8
   *NAME←NAME
   ```

2. Load the user functions and INBRTS.BN into core with the Absolute Loader, and save the core image.

   ```
   .R ABSLDR
   *INBRTS,USER$
   ```

3. Using OS/8 ODT, modify the User Function Table in INBRTS which starts at 1574, entering pointers to the user assembly language functions. Unmodified table entries are 236(8); replace these entries with the starting addresses (pointers)to the user assembly language function. Starting at location 1574, enter the pointers in the table in the exact corresponding order in which the functions appear in the UDEF statement which defines them.

```
        .ODT

        1574/236 2000 (LF)

        1575/236 2010
        ↑C
        .SAVE SYS INBRTS 0-7577,10000-1XXXX;7605
```

where XXXX represents the high core address of the user functions.

In the procedure above two functions are interfaced which start at locations 2000 and 2010 respectively . LF indicates pressing the LINE FEED key.

Example: There are three assembly language functions in our package, called PLT, HI, and LO. The BASIC user is instructed that when he uses this function package, PLT, HI and LO must be defined in that order. The function files, then, look like:

Function Source (USER.PA)

```
        *2000
HI,                                   /ENTRY POINT FOR HI
        .                             /ORDER OF ENTRY POINTS IS
        .                             /NOT CRITICAL
        .
        .
        CIF CDF0
        JMP I [ILOOP
PLT,                                  /ENTRY FOR PLT
        .
        .
        .
        .
        CIF CDF0
        JMP I [ILOOP
LO,                                   /ENTRY FOR LO
        .
        .
        .
        .
        CIF CDF0
        JMP I [ILOOP
```

To enter these three functions into the user function table in INBRTS, the procedure is:

```
        .GET SYS:INBRTS

        .ODT

        1574/236  PPPP (LF)
        1575/236  HHHH (LF)
        1576/236  LLLL
        ↑C
        .SAVE SYS:INBRTS
```

where PPPP, HHHH, and LLLL represent octal starting addresses for PLT, HI, and LO respectively. LF indicates pressing the LINE FEED key.

NOTE

> INBRTS establishes calls to the user function by setting up a one-to-one correspondence between the pointers at 1574 and the function names present in a UDEF statement. Therefore, the order of the pointers must exactly correspond to the order of the function definitions in UDEF. If the BASIC user wants to use only the n(th) function in a given user package, he must still define n functions in the UDEF statement, though the first n-1 may be dummies.
>
> For example: A package of four assembly language functions that use arguments as follows.
>
> ```
> ONE (X)
> TWO (X,Y)
> THR (X,Y,Z)
> FOU (X,Y,Z,A)
> ```
>
> If a BASIC user wishes to use only one function ONE, the UDEF would look like:
>
> ```
> 10 UDEF ONE (X)
> ```
>
> If the BASIC user wants to use functions ONE and FOU, the UDEF would look like:
>
> ```
> 10 UDEF ONE(X),DUA(D),DUB(D),FOU(D)
> ```
>
> For this user, DUA and DUB are dummy user function names which are never called; they merely set up the right correspondence between names and pointers.
>
> The easiest way to assure that the pointers are established correctly is to provide the user of an assembly language function package with a set of complete UDEF statements that define all functions correctly, and instruct him to use the complete set of UDEF's each time.

11.11 GENERAL CONSIDERATIONS AND HINTS

11.11.1 Routines Unusable by Assembly Language Functions

Because only one overlay is resident at any time, assembly language functions can only call routines in the overlay which is resident when executing, they cannot use any routines that reside in any of the two other overlays. Following is a list of the INBRTS functions and routines grouped by overlay.

Routine Name	Function
Arithmetic Overlay	
FFATN	Arctangent Function
FFCOS	Cosine Function
FFEXP	Exponential Function (e^X)
EXPON	Power Function (A^B)
INT	Signed integer Function
FFLOG	Naperian log Function
SGN	Sign Function
FFSIN	Trigonometric Sine Function
RND	Random Number generator
FROOT	Square root Function
String Overlay	
ASC	String Function ASC
CHR	CHR$ Function
DAT	DAT$ Function
LEN	String length Function
POS	String search Function
SEG	String segmenting Function
STR	STR$ Function
VAL	VAL Function
TRC	Trace Function
File Overlay	
CHAIN	File manipulation routines
CLOSE	File manipulation routines
OPENAF	File manipulation routines
OPENAV	File manipulation routines
OPENNF	File manipulation routines
OPENNV	File manipulation routines

11.11.2 Using OS/8

So long as the assembly language function is carefully designed to protect all core areas being used by INBRTS, there are no restrictions on the function's use of OS/8. Once the page 17600 resident monitor has been restored, the OS/8 User Service Routine (USR) may be called at will, and files may be located, used, and closed again. If the user's BASIC program does not need full file capabilities, the assembly language function is free to use the buffer space from 11400-12377. The assembly language function should be careful, however, to check the bit maps and status words on page 0 to make certain a given area is free before using it. Note that the system device driver may be used without restoring the page 17600 resident: restoration is only required when it is desired to use the USR.

11.11.3 Page 0 Usage

Following is a map of the INBRTS page 0 usage. Locations marked with an * may be used by the assembly language function without saving the contents.

FIELD 0, PAGE 0

0-2	Interrupt vector
3-7	System parameters and temps
10-15 *	Index registers
16-17	System pointers
20-30	Compiler-INBRTS communication
30-36	System registers
37-62	Floating-point package area
63-67	System registers
73-107	Constants
110-161	Links to INBRTS subroutines
162-177	I/O Table pointers

FIELD 1, PAGE 0

0-17	Unused (available for user)
20-33	Scratch Area
34-40	Pointers for FIELD 0 routines

Assembly language functions are free, of course, to use any of the pointers or constants, but they must be intact when control is returned to INBRTS.

11.12 ASSEMBLY LANGUAGE FUNCTION EXAMPLE

To illustrate the material in the previous sections, an example of a complete assembly language function follows. Note that this example is tutorial in nature; it is meant to illustrate argument passing and getting along with INBRTS rather than a typical use for assembly language functions.

The example consists of two user functions and an addition to the interrupt skip chain. The function performed is reading a string from a second terminal and storing it in a string variable; the string will be printed on the main console terminal if a carriage return is entered.

```
 100  REM  THIS IS A DEMONSTRATION INDUSTRIAL
 105  REM  BASIC PROGRAM
 110  REM  WHICH WILL RUN COMPUTE BOUND AND
 120  REM  ACCEPT CHARACTERS FROM A SECOND TELETYPE
 130  REM  USING THE INTERRUPT FACILITY.
 140  REM
 150  REM  THE RUN TIME SYSTEM HAS BEEN MODIFIED
 160  REM  TO ADD CODE TO THE INTERRUPT SKIP CHAIN
 170  REM  AND INSERT THE ADDRESSES OF THE USER
 180  REM  FUNCTIONS
 200  DIM  A$(40)
 210  UDEF  TT1(X),TT2$(A$)
 220  PRINT  "I'M RUNNING"
 230  TIMER .1 THEN 1000
 300  FOR I = 1 TO 5E5
 310  J = J+1
 320  IF J 500 THEN 350
 330  PRINT I
 340  J = 0
 350  NEXT I
 360  TIMER 0 THEN 1000
 370  PRINT "DONE"
 380  STOP
1000  X = TT1(0)
1010  IF X=0 THEN 1040
1020  IF X 0 THEN 1050
1030  PRINT "*****"; A$
1040  DISMISS
1050  A$ = TT2$(A$)
1060  DISMISS
5000  END
```

11.13 LINKING INTO THE INTERRUPT SKIP CHAIN

If the user desires to process I/O from an I/O device which is currently not supported by INBRTS and the interrupt facility must be used, the user must add code to the interrupt skip chain.

The user may allocate the second file I/O buffer at location 12000 for storage of the interrupt driven code and the skip chain. If this is done, remember to mark this buffer as "in use" so that INBRTS will not try to allocate it. File I/O buffer 1 may not be used for interrupting code, but can be used for non-interrupting user functions because it is swapped by the OS/8 system when INBRTS calls upon the USR to perform services.

NOTE

> INBRTS gets an interrupt regularly once every 1/50 second for clock services. Obviously interrupts are off in the skip chain, therefore no computations of any type should be performed.

Actual linking into the skip chain is performed as follows:

```
.R ABSLDR
*INBRTS
*UFUNCS$                 (UFUNCS are any user functions)
.ODT
6773/0000 addr           addr is the FIELD 1 address of the
                         continuation of the skip chain
1041/6764 6773
↑C
.SAVE SYS INBRTS....     save arguments as needed by the user
                         functions and INBRTS.
```

NOTE

> Entries in the user function table may also be modified at this time. Location 6773 in FIELD 0 is a CIF CDF 10. If the user wants the data field to be zero upon entry to the FIELD 1 portion of the skip, he must change the CIF CDF 10 to CIF 10.

```
                    FIELD 1

      /EXAMPLE OF ASSEMBLY LANGUAGE INTERFACE
      /            TO INBRTS


      /DEFINE USED IOTS
           ASKF = 6331
           AKRB = 6336
           ATSF = 6341
           ATLS = 6346
           ATCF = 6342


      /DEFINE REFERENCED LOCATIONS IN INBRTS
           FACCLR =
           STRLEN =
           SACPTR =
           CALLF0 =
           INTRET =
           ILOOP =
           HORD =

      /THE ADDITION TO THE INTERRUPT SKIP CHAIN
         *2000

TTYSCN,    AKSF           /IS READER FLAG UP
           JMP TESTOUT    /NO...
           AKRB           /YES...GET CHARACTER
           DCA CHAR       /SAVE IT
TESTOUT,   ATSF           /IS PRINTER FLAG UP
           SKP            /NO...
           ATCF           /YES...CLEAR IT
           CIF CDF 0      /RETURN BACK TO FIELD 0
           JMP I (INTRET


      /ROUTINE TO DETERMINE FUNCTION DESIRED BY
      /ALTERNATE TELETYPE

      /IF FAC =0, NO ACTIVITY, IF FAC>0 ADD CHARACTER TO
      /CURRENT STRING, AND IF FAC<0 PRINT STRING
TT1,       TAD CHAR       /GET CHARACTER
           SNA            /IS ONE REALLY THERE?
           JMP NOCHAR     /NO...NOTHING INPUT RECENTLY
           ATLS           /YES...ECHO IT
           AND (77        /MAKE IT 6-BIT FOR INTERNAL FORMAT
           DCA SVCHAR     /SAVE CHARACTER FOR STRING FUNCTION
           DCA CHAR       /CLEAR LAST TYPED FLAG
           CIF 0          /USE CALLF0 INTERFACE FUNCTION
           JMS I (CALLF0  /TO CALL ROUTINE WHICH CLEARS
           FACCLR         /FAC
           TAD SVCHAR     /SEE IF CHARACTER
           TAD (-15       /WAS A CARRIAGE RETURN
           SNA CLA        /SKIP IF NOT
           CLL CML RAR    /YES...MAKE AC NEGATIVE
           IAC            /SET AC TO 1 OR 4001
NOCHAR,    CDF 0          /PLACE RETURN ARGUMENT IN
```

```
        DCA I (HORD     /FAC HIGH ORDER WORD
        CIF CDF 0       /RETURN
        JMP I (ILOOP    /TO INTERPRETER


    /ROUTINE TO CONCATENATE LAST TYPED CHARACTER
    /ON ALTERNATE TELETYPE TO STRING IN THE STRING
    /ACCUMULATOR

    /SHOULD ONLY BE CALLED IF TT1 RETURNED A
    /POSITIVE NUMBER

TT2,    CDF 0           /INBRTS INFO. IS IN FIELD 0
        TAD I (STRLEN   /GET CURRENT STRING LENGTH
        CIA             /MAKE IT POSITIVE
        CLL RAR         /CONVERT TO WORD OFFSET FROM CHAR OFFSET
        TAD I (SACPTR   /POINTS TO SAC-1
        DCA PTR         /SAVE IT
        SNL             /LINK=1 MEANS 2ND CHAR IN WORD
        JMP HIGHPT      /WILL BE FIRST CHARACTER IN WORD
        TAD I PTR       /GET UPPER HALF OF LAST CHARACTER
        TAD SVCHAR      /ADD IN NEW CHARACTER
PUTIN,  DCA I PTR       /PLACE BACK INTO SAC
        CMA             /BUMP LENGTH UP BY ONE, BUT IT IS
        TAD I (STRLEN   /NEGATIVE
        DCA I (STRLEN
        CIF CDF 0       /RETURN
        JMP I (ILOOP    /TO INTERPRETER

HIGHPT, TAD SVCHAR      /CHARACTER MUST GO INTO HIGH
        CLL RTL         /SIX BITS OF WORD
        RTL
        RTL
        JMP PUTIN       /FINISH PUTTING IT IN


SVCHAR, 0
CHAR,   0
PTR,    0

        $
```

APPENDIX A

OS/8 BASIC STATEMENT, COMMAND, AND FUNCTION SUMMARY

A.1 ELEMENTARY OS/8 BASIC STATEMENTS

Statement	Example of Form	Explanation
CHAIN	CHAIN DEV:Filemane.ext	Stops execution of the current program, retrieves the program named in the CHAIN statement from the specified device and file, compiles the chained program and begins execution of the program.
DATA	DATA x1,x2,...,xn	Values x1 through xn are to be associated with corresponding variables in a READ statement. The values may be either numerics or strings. Strings must be enclosed by quotation marks.
DEF	DEF FNB(x) = f(x) DEF FNB(x,y) = f(x,y)	The user may define his own function to be called within his program by putting a DEF statement at the beginning of a program. The function name begins with FN and must have three letters. The argument list in the function may contain as many as 4 numeric and 2 string arguments.
DIM	For numerics: DIM v1(n1), v2(n1,m2)	Enables the user to create a table or array with the specified number of elements where v is the variable name and n and m are the maximum subscript values. Any number of arrays can be dimensioned in a single DIM statement.
	For strings: DIM v1$(I),v2$(K,L)	I is the length of string variable v1$, K is the number of strings and L is the length of strings of string variable v2$. Strings longer than

		8 characters must be dimensioned. String tables are illegal.
END	END	Last statement in the program. Signals completion of the program.
FOR-TO-STEP	FOR v=x1 TO x2 STEP x3	Used to implement loops; the variable v is set equal to the expression x1. From this point the loop cycle is completed following which v is incremented after each cycle by x3 until its value is greater than x2. If STEP x3 is omitted, x3 is assumed to +1. x3 may also be negative. If x3 is positive and x1>x2, the loop is never executed.
GOSUB	GOSUB x	Allows the user to enter a subroutine at several points in the program. Control transfers to line x.
GO TO or GOTO	GO TO n or GOTO n	Transfers control to line n and continues exection from there.
IF-GOTO	IF f1 r f2 GOTO n	Same as IF-THEN.
IF-THEN	IF f1 r f2 THEN n	If the relationship r between the expressions f1 and f2 is true, transfers control to line n; if not, continue in regular sequence. If f1 and f2 are string they are compared on the basis of the ASCII numeric value of each character in the string. See paragraph 7.1.4 for details.
INPUT	INPUT v1,v2,...,vn	Causes typeout of a ? to the user and waits for the user to supply the values of the variables v1 through vn. See paragraph 3.5.1 for INPUT rules.
LET	LET v = f	Assigns the value of the expression f to the variable v. The word LET is optional.

NEXT	NEXT v	Used to tell the computer to return to the FOR statement and execute the loop again until v is greater than or equal to terminal value in FOR statement (or v is ≤ terminal value if increment <0).
PRINT	PRINT a1,a2,...,an	Prints the values of the specified arguments, which may be variables, text, or format control characters (, or ;). See paragraph 3.5.2 for format control and printing rules.
RANDOMIZE	RANDOMIZE	Generates new sets of random numbers.
READ	READ v1,v2,...,vn	Variables v1 through vn are assigned the value of the corresponding numbers or strings in the DATA list.
REM	REM	When typed as the first three letters of a line everything between REM and end of line is ignored to allow typing of remarks within the program.
RESTORE	RESTORE	Sets pointer back to the beginning of the list of DATA values.
RETURN	RETURN	Transfers control to the statement following the last GOSUB.
STOP	STOP	When encountered in the program the STOP statement terminates execution.
UDEF	UDEF function name (arguments)	The UDEF statement is used to define the syntax of a call to a user-coded machine language function (function name) with its associated arguments.

USE	USE v1,v2,...,vn	The USE statement identifies the lists and arrays referenced by a user-coded machine language function.
CONTACT-THEN	CONTACT v THEN n	Define the line number (n) to schedule when a contact interrupt occurs on point v.
COUNTER-THEN	COUNTER v THEN n	Define the line number (n) to schedule when the counter point v reaches zero.
TIMER-THEN	TIMER v THEN n	Define the line number (n) to schedule when the time v has elapsed. Continue to cycle until timer disabled.
DISMISS	DISMISS	Exit from user process interrupt mode - resume main line code.

A.2 OS/8 BASIC FILE STATEMENTS

See Chapter 10 of this manual for details regarding the use of OS/8 BASIC file statements and file descriptions.

Statement	Example of Form	Explanation
CLOSE #	CLOSE #N:	Closes a file N previously opened by a FILE #N statement where N is the numerical expression for the file number.
FILE # FILEV # FILEN # FILEVN #	FILE #n:s FILEV #n:s FILEN #n:s FILEVN #n:s	These statements, open, respectively, a fixed length ASCII, variable length ASCII, fixed length numeric, and variable length numeric file, where n has a value of 1 or 2 and s is a string expression with a value of DEV:FILE.EX. DEV is system handler.
INPUT #	INPUT #N: v1,v2,...,vn	Reads v1 through vn from file number N.
IF END #	IF END #N THEN n	If an attempt has been made to read or write

		beyond the last datum in file number N, control passes to line number n.
PRINT #	PRINT #N: a1,a2,...,an	Writes the values of the arguments into file number N.
RESTORE #	RESTORE #N	Sets pointer back to beginning of file number N.

A.3 OS/8 BASIC CONTROL COMMANDS

Command	Example of Form	Explanation
BYE	BYE	Exits from BASIC and returns control to Keyboard Monitor.
CTRL/C	CTRL/C	Stops execution of program and returns control to OS/8 Industrial BASIC editor. In editor mode returns control to OS/8 Keyboard Monitor.
CTRL/O	CTRL/O	Stops the listing of text and returns control to BASIC editor.
LIST LI	LIST LI	Lists program with heading.
LIST n LI n	LIST n LI n	Lists program starting from line n, with heading.
LISTNH LISTNH n	LISTNH LISTNH n	Same as LIST and LIST n, but heading suppressed.
NAME NA	NAME FILE.EX NA FILE.EX	This statement renames the current program in user core.
NEW	NEW FILE.EX	Used to name a program to be created.
NE	NE FILE.EX	Performs an inherent SCRATCH.
OLD OL	OLD DEV: FILE.EX OL DEV: FILE.EX	Performs inherent SCRATCH and retrieves a previously created file from the device specified.
RUN RU	RUN RU	Compiles and executes the program currently in core, with heading.
RUNNH	RUNNH	Compiles and executes the program currently in core, with heading suppressed.
SAVE SA	SAVE DEV:FILE.EX SA DEV:FILE.EX	Saves the current program on the device specified.

SCRATCH SC	SCRATCH SC	Deletes all program statements from user core.

A.4 OS/8 BASIC FUNCTIONS

Function	Explanation
ABS(X)	This function returns the absolute value of the argument X.
ASC(X)	This function returns the decimal ASCII number (see Appendix D) corresponding to the character X.
ATN(X)	This function calculates the angle (in radians) whose tangent is given by the argument X.
CHR$(X)	X is a numeric expression (modulo 64) which is truncated to an integer. The decimal integer is converted to its equivalent ASCII character (see Appendix D).
COS(X)	The cosine function is used to calculate the cosine of an angle specified in radians.
DAT$(X)	This function returns the data in the form MM/DD/YY. The argument X is a dummy argument.
EXP(X)	This function calculates the value of e(2.71828) raised the the X power.
FNA(X)	Used with a DEF statement to define a user function. Thereafter used as any other function.
INT(X)	This function returns the greatest integer less than the value of the argument X.
LEN(X$)	This function returns the number of characters in string X$.
LOG(X)	The LOG(X) function calculates the natural logarithm of X.
PNT(X)	This function, which can be used only in a PRINT statement, outputs the character whose decimal ASCII value is X. This function is useful for outputting non-printing characters.
POS(X$,Y$,Z)	This function returns the location in string X$ of the first occurrence of string Y$ starting with the Zth character in string X$. See paragraph 7.2.4 for POS function rules.
RND(X)	This function returns a random number between 0 and 1.
SEG$(X$,Y,Z)	This function returns the substring of X$ which is between positions Y and Z inclusively. See paragraph 7.2.5 for SEG$ function rules.

SGN(X)	The sign function returns the value 1 if X is any positive number, 0 if zero, and -1 if any negative number.
SIN(X)	This function is used to calculate the sine of an angle specified in radians.
SQR(X)	The square root function computes the square root of the absolute value of an expression.
STR$(X)	This function converts the numeric value of X to a string which is its decimal representation.
TAB(X)	This function which can only be used in a PRINT statement, moves the print head to position X.
TRC(X)	This function turns on the trace feature if x=1 and turns off the trace feature if x=0. When the trace feature is on, line numbers are printed between percent signs as the lines are encountered in the program. The feature is useful when debugging programs.
VAL(X$)	This function returns the number represented by the string X$ which is the decimal representation of a number.

A.5 INDUSTRIAL FUNCTIONS

Function	Explanation
ANI(P,G)	This function reads analog input point P at gain G and return the value in volts.
ANO(P,V)	This function loads the D/A point P with the value V. 1023=full scale 0=minimum
CLK(X)	If the value of X is zero or negative this function returns the time of day clock in seconds. If the value of X is positive the time of day clock is set to the value of X $1 \leq x \leq 86400$
CNI(P)	This function returns the number of counts remaining until zero in counter P.
CNO(P,V)	This function loads the counter (P) such that V items will be counted before the counter (P) interrupts (overflows).
RDI(P,N)	This function returns the value of point(s) P through N.
SDO(P,N,V)	This function loads the value V, in binary, into point(s) P through N (right justified).

RDO(P,N)	This function return the results of the last data sent to point(s) P thru N.

LNE(X)
STA(X)
CNT(X)
	These are UPIR Identification Functions.
	(Refer to chapter 8, section 8.4).
	Arguments are dummies.

APPENDIX B

COMPILE-TIME DIAGNOSTICS

Compile-time diagnostic messages typed out by OS/8 BASIC are in the form:

```
        XX YY
```

where XX is the diagnostic code and YY is the line number at which the error occurred.

Diagnostic Code	Explanation
CH	Error in CHAIN statement.
DE	Error in DEF statement.
DI	Error in DIM statement syntax or string dimension greater than 72, or array dimensioned twice.
FN	Error in file number or filename designation.
FP	Incorrect FOR loop parameters or FOR loop syntax.
FR	Error in function arguments or function not defined.
IF	THEN or GOTO missing from IF statement, or bad relational operator.
IO	I/O error.
LS	Missing equal sign in LET statement.
LT	Statement too long (greater than 80 characters).
MD	Line number defined more than once. YY equals line number before line in error.
ME	Missing END statement.
MO	Operand expected, not found.
MP	Missing parenthesis or error in expression within parentheses.
MT	Operand of mixed type or operator does not match operands (e.g., A$=1 and A&2 are both incorrect).
NF	NEXT statement without corresponding FOR statement.
NM	Line number missing after GOTO, GOSUB, or THEN.
OF	Output file error.

PD	Pushdown stack overflow. Result of either too complex a statement (or statements) or too many nested FOR-NEXT loops.
QS	String literal too long or does not end in quote.
SS	Subscript or function argument error.
ST	Symbol table overflow due to too many variables, line numbers, or literals. Combine lines using backslash (\) to condense program.
SY	System incomplete. System files such as INBSIC.SV, INBCMP.SV, and INBRTS.SV missing.
TB	Program too big. Condense or CHAIN.
TD	Too much data in program.
TS	Too many total characters in the string literals.
RT	Incorrect real time statements.
UD	Error in UDEF statement.
UF	FOR loop without corresponding NEXT statement.
US	Undefined statement number. (i.e., statement number mentioned in statement is not in program.)
UU	Incorrect or missing array designator in USE statement.
XC	Extra characters after the logical and of line. (e.g., LET A=B.D -- the dot after the B suggests that B is the end of the line and the character D appears extraneous.)

APPENDIX C

RUNTIME DIAGNOSTICS

Runtime diagnostic messages typed out by OS/8 BASIC are in the form:

```
XX AT LINE YYYYY
```

where XX is the diagnostic code and YYY is the line number at which the error occurred. Most runtime errors stop execution of the program. Those errors which do not stop the program are termed non-fatal (NF) and are indicated below.

Diagnostic Code	Explanation
BO	No more file buffers available.
CI	Inquire failure in CHAIN. Device not found.
CL	Lookup failure in CHAIN. Filename not found.
DA	Attempt to read past end of data list.
DE	Device driver error. Caused by hardware I/O failure.
DO	No more room for drivers. Too many different devices used in file commands.
DV	Attempt to divide by 0. Result is set to zero. (NF)
EF	Logical end of file. Usually caused when I/O device runs out of medium.
EM	Attempt to exponentiate a negative number to a power.
EN	Enter error in opening file. Device is read only or there is already one variable file open on that device or file not found.
FB	FILE busy. Attempt to use a file already in use.
FC	OS/8 error while closing variable file. Device is read only on file closed already.
FI	Attempt to close or use unopened file.
FM	Attempt to fix minus number. Usually caused by negative subscripts or file numbers.
FN	Illegal file number. Only 0,1 and 2 are legal.
FO	Attempt to fix number greater than 4095. Usually caused by negative subscripts of file numbers.

GR	RETURN without a GOSUB.
GS	Too many nested GOSUBs. The limit is 10.
IA	Illegal argument in UDEF function call.
IF	Illegal DEV:filename specification.
IN	Inquire failure in opening file. Device not found.
LM	Attempt to take log of negative number or 0.
OE	Driver error while overlaying. Caused by SYS device hardware error.
OV	Numeric or input overflow.
PA	Illegal argument in POS function.
RE	Attempt to read past end of file. (NF)
SC	String too long (greater than 72 characters) after concatenating.
SL	String too long or undefined.
SR	Attempt to read string from numeric file.
ST	String truncation on input. Stores maximum length allowed. (NF)
SU	Subscript out of DIM statement range.
SW	Attempt to write string into numeric file.
VR	Attempt to read variable length file.
WE	Attempt to write past end of file (NF).

Added Run-Time Process I/O Errors

BE	Width crosses module boundary.
DR	DISMISS executed while not in user interrupt routine.
GE	Module is not the correct generic code (Type).
IG	Invalid gain specified for analog input.
GC	Illegal generic code (Module not supported).
IV	Data item out of valid range.
SG	Channel not defined at system generation.

IC	Illegal channel (channel not in valid range).
QW	Zero width or width not wide enough.
TM	Attempt to start too many timers, counters or contacts.
TE	Time too large ($\geqslant$ 86400 for clock or timer.

APPENDIX D

ASCII CONVERSION TABLE

Character	6-Bit Octal	Decimal	Character	6-Bit Octal	Decimal
A	01	1	6	66	54
B	02	2	7	67	55
C	03	3	8	70	56
D	04	4	9	71	57
E	05	5	space	40	32
F	06	6	!	41	33
G	07	7	"	42	34
H	10	8	#	43	35
I	11	9	$	44	36
J	12	10	%	45	37
K	13	11	&	46	38
L	14	12	' (apostr.)	47	39
M	15	13	(	50	40
N	16	14	)	51	41
O	17	15	*	52	42
P	20	16	+	53	43
Q	21	17	, (comma)	54	44
R	22	18	-	55	45
S	23	19	.	56	46
T	24	20	/	57	47
U	25	21	:	72	58
V	26	22	;	73	59
W	27	23	<	74	60
X	30	24	=	75	61
Y	31	25	>	76	62
Z	32	26	?	77	63
0	60	48	@	00	0
1	61	49	[	33	27
2	62	50	\	34	28
3	63	51	]	35	29
4	64	52	↑	36	30
5	65	53	←	37	31

APPENDIX E

OS/8 INDUSTRIAL BASIC SYSTEM BUILD INSTRUCTIONS

OS/8 Industrial BASIC is distributed on DECtape. The DECtape version of OS/8 Industrial BASIC contains SAVE images (ready-to-run) for each of the OS/8 Industrial BASIC system components as well as binaries for each system component. OS/8 Industrial BASIC, then, is distributed as the following files:

File	Component	Distributed on:
INBSIC.BN	Binary for editor	DECtape
INBSIC.SV	Editor save image	DECtape
INBCMP.BN	Compiler binary	DECtape
INBCMP.SV	Compiler save image	DECtape
INBLDR.BN	Loader binary	DECtape
INBLDR.SV	Loader save image	DECtape
INBRTS.BN	Runtime system binary	DECtape
EAEOVR.BN	Overlay for KE-8/E EAE (8/E with KE-8/E EAE)	DECtape
INBRTS.SV	Runtime system save image (from INBRTS.BN)	DECtape
INBSIC.AF	Arithmetic function overlay	DECtape
INBSIC.SF	String function overlay	DECtape
INBSIC.FF	File manipulation overlay	DECtape

Assembling the OS/8 Industrial BASIC sources

Instructions for assembling each of the OS/8 Industrial BASIC sources follow. PAL-8 (under OS/8) is used, and the descriptions represent OS/8 commands. To assemble OS/8 Industrial BASIC, a 12K machine is required.

The OS/8 BASIC sources are named as follows:

```
NAME.MM
```

where NN represents the version number. For the first release, the files are named:

Name	Component
INBSIC.01	Editor Source
INBCMP.01	Compiler Source
INBLDR.01	Loader Source
INBRTS.01	Runtime System Source

1. To assemble the editor:

```
.R PAL8
*DEV:INBSIC.BN DEV:INBSIC.01
```

2. To assemble the compiler:

```
.R PAL8
*DEV:INBCMP.BN<DEV:INBCMP.01
```

3. The runtime system source is conditionalized for PDP-8/E with EAE. Assembly instructions for each of the supported configurations follow.

   To assemble for PDP-8E without EAE:

```
.R PAL8
*DEV:INBRTS.BN<DEV:INBRTS.01/K
```

   To assemble the run-time system overlay for PDP-8E, PDP-8F or PDP-8/M with KE-8/E EAE option, prepare a file called EAE.PA that looks as follows:

```
EAE=1
PDP8E=1
PAUSE
```

   Then:

```
.R PAL8
*DEV:EAEOVR.BN-DEV:EAE.PA,DEV:INBRTS.01/K
```

Making SAVE Images from Binary Files:

To create SAVE images of each of the OS/8 Industrial BASIC binaries, perform the following OS/8 commands.

1. For the editor:

```
.R ABSLDR
*DEV:BASIC.BN$
.SAVE SYS:BASIC;3011
```

2. For the compiler:

```
.R ABSLDR
*DEV:BCOMP.BN$
.SAVE SYS:BCOMP;7000
```

3. For the loader:

```
.R ABSLDR
*DEV:INBLDR.BN$
.SAVE SYS:BLOAD;7605
```

4. For the runtime system:

```
.R ABSLDR
*DEV:INBRTS.BN$  (if you have no KE-8/E EAE option)

          or
```

```
*DEV:INBRTS.BN,DEV:EAEOVR.BN$
     (if you have PDP-8/E, PDP-8M or
     PDP-8F with KE-8/E EAE)

.SAVE SYS INBRTS 0-7577, 10000-11377; 7605

.SAVE SYS INBSIC.AF 3400-4577

.SAVE SYS INBSIC.SF 12000-13177

.SAVE SYS INBSIC.FF 13400-14577
```

NOTE

All BASIC system files must reside on the systems device (SYS:).

5. At this point, Industrial BASIC is ready to run.

APPENDIX F

INDUSTRIAL BASIC SYSTEM GENERATION

Each functional I/O module (analog input, analog output, digital input, digital output) operates on a continuous range of logical addresses. The mapping between the logical address and the corresponding physical address of the UDC module and subchannel is performed within the system, based on tables which define the UDC configuration and which the user generates during "SYSTEM GENERATION".

System generation is the process of setting the functional module tables for analog input, analog output, digital input, digital output, specifing the base of the contact modules, and the base of counter modules.

The first step is to determine the classification of each physical channel. The classification groups are contacts(W742), counter(W734), analog input(ADU01), analog output(BA633), digital input (W740) and digital output. Once the channels are classified the I/O table in INBRTS may be generated. The following is a sample system generation of this set of modules.

Slot	Module	Class
0	BW742	contact
1	BW742	contact
2	BW740	digital input
3	BW740	digital input
4	BM684	digital output
5	BM686	digital output
6	BM686	digital output
7	BW734	counter
8	ADU01	analog input

The two interrupting module types (contacts and counters) require only the base address of the initial contiguous physical channel of each module type. The two module types do not have to be adjacent to each other, but each module (contacts and counters) MUST be in contiguous channels.

Logical channels start at the high order bit (or subchannel zero) of the first physical channel of a given type of functional I/O module and continue sequentially in the order the physical channels are specified in the functional I/O table.

If channels 5,7,9 are digital input and the table entries are in that order, logical channels 1 through 12 are on physical channel 5, while 25 through 36 would be on physical channel 9 (Note - the breaks are dependent on the number of I/O points or subchannels per module).

The tables are most simply altered via OS/8 ODT; the UDC physical address should be converted to octal addresses.

The table addresses are:

```
DIGITAL INPUT -  10600

DIGITAL OUTPUT -  10610

ANALOG INPUT -  10620

ANALOG OUTPUT -  10630

CONTACT -  7310

COUNTER -  7276
```

The sample system table would be altered as follows:

```
.GET SYS INBRTS

.ODT

10600/7777  0203(CR)

10610/7777  0405(LF)
10611/7777  0677(CR)

106201/7777  1077(CR)

7276/7777  0007(CR)

7310/0000  0(CR)
↑C
.SAVE SYS INBRTS
```

Notice that addresses are entered in octal numbers and if an odd number of addresses exists the last entry is terminated by 77. A line feed gives the next entry in the current table. The table generation is terminated via CTRL/C and the SAVE command. The table may be altered to reflect alteration in hardware configurations; the sequence of channel entries determines the mapping into logical channels.

APPENDIX G

OPTIMIZING SYSTEM PERFORMANCE

There are several steps the OS/8 Industrial BASIC user can take to speed up BASIC execution and compilation times, thus speeding up OS/8 Industrial BASIC throughput rates. This appendix contains suggestions for improving system performance.

1. Bypassing the Editor

The OS/8 Industrial BASIC compiler is constructed such that it will accept any source file for input. Thus, it is possible to execute an already existing BASIC program directly, saving the overhead of an OLD and RUN command to the editor. The format is as follows:

```
.R INBCMP
*DEV:FILE.BA
```

If OS/8 Industrial BASIC is used in this fashion, it returns to the OS/8 Monitor on completion, rather than the OS/8 Industrial BASIC editor.

```
Normal Usage            Faster Equivalent

.R INBSIC               .R INBCMP
NEW OR OLD--FILE        *FILE

READY

RUNNH

READY
```

In general, use R INBSIC when:

a. Creating new programs or modifying old programs
b. Debugging old programs

Use R INBCMP:
a. To run existing programs
b. In BATCH stream to run BASIC programs

Source files for use by INBCMP must conform to the following rules:

a. There should be no blank lines.

b. Statements must be in the order in which the are to be executed.

c. Line numbers are required only for statements that are referenced in IF, GOSUB, and GOTO statements. In other words, if the only way a statement may be reached is for the preceding statement to be executed, it does not require a line number. In the following example, there are no unnecessary line numbers.

```
          FOR I=1 TO 10
          IF I=2 THEN 400
          PRINT I
          GO TO 410
      400 PRINT "TWO"
      410 NEXT I
          END
```

Note that the source file can be created in one of two ways: it may be created in the normal fashion with the OS/8 Industrial BASIC editor and saved (in which case all lines will contain line numbers), or it may be prepared using any of the other OS/8 editors (EDIT, TECO). In this second case, the user can take advantage of the extra features supported by these sophisticated editors over the OS/8 Industrial BASIC editor.

2. Placement of BASIC Overlays on SYS

For DECtape system users, the performance of the system can be improved by two simple steps:

a. Use a DECtape drive other than DTA0 for DSK: (via the ASSIGN statement). TD8E system may only use drives 0 and 1.

b. Place the OS/8 Industrial BASIC system files as close together on the SYS tape as possible. The best approach is to make a "BASIC tape" containing only the OS/8 system, PIP, and the BASIC system image files.

Both actions have the effect of speeding up OS/8 BASIC by the simple reduction of the tape motion required for overlaying and compiling.

3. Placement of Function Calls Within BASIC Programs

Most of the OS/8 BASIC functions and file operations reside in one of the three system overlays. Since the system overlay driver reads in an overlay only if the function desired is not present in the currently resident overlay, overlaying overhead can be reduced by the simple mechanism of placing calls to functions that reside in the same overlay as close as possible in the BASIC program. For example:

```
10 INPUT A$
20 Z$= SEG$(A$,1,6)
30 FILEN #1: Z$
40 INPUT A$
50 Z$= SEG$(A$,1,6)
60 FILEN #2:Z$
```

The above BASIC program uses the first six characters of a string typed by the user as a file name to open a BASIC file. It uses the SEG$ function, a File command, the SEG$ function, and the File command again. Since SEG$ and FILE are in different overlays, the overlayer will be used four times. A faster way to accomplish the same operations follows:

```
10 INPUT A$,B$
20 Z$=SEG$(A$,1,6)
30 X$=SEG$(B$,1,6)
40 FILEN #1: Z$
```

```
        50 FILEN #2: X$
```

The above only overlays twice, saving considerable time in the program execution. The functions are grouped in the overlays as follows:

```
     Overlay 1 (INBSIC.AF): SIN,COS,ATN,LOG,EXP,RND,SQR,SGN,POWER(A B)

     Overlay 2 (INBSIC.SF): ASC,CHR$,DAT$,LEN,POS,SEG$,STR$,VAL

     Overlay 3 (INBSIC.FF): CLOSE, FILE, FILEN, FILEV,FILEVN
```

APPENDIX H

RUN-TIME

SYMBOL TABLE

```
/OS/8 INDUSTRIAL BASIC RUNTIME    PAL8-V7  9/20/73   PAGE 148-1

A       6347      ATAN    4200      CHAIN   3600      COUNCK  1363
ABSV    3575      ATANA1  4430      CHAR    0053      COUNTR  1236
ABSVAL  2363      ATANA2  4436      CHARNO  6677      CPLOOP  3233
ACH     0045      ATANA3  4444      CHAR3P  2746      CR      6432
ACL     0046      ATANB0  4425      CHAR3U  3064      CREAD   3665
ACLO    0046      ATANB1  4433      CHKB3   6604      CRETN   3456
ACSAVE  6746      ATANB2  4441      CHKB4   6613      CRFUNC  0765
ACSR    6072      ATANB3  4447      CHR     3400      CRLF    0153
ACSRPT  4704      A999    2507      CHRNOL  0151      CRLFR   2346
ACX     0044      BADCHN  0055      CI      3624      CRREP   1360
AC0     0040      BADGC   0044      CL      3640      CSFN    2001
AC1     0041      BAS     6617      CLEANP  4243      CSMOVE  4051
AC2     0042      BCGET   3035      CLENG   3442      CSTA    4070
ADCALC  0636      BCGETL  2776      CLF     3570      CSTAC   4071
ADFC    0705      BCPUT   0745      CLN     0030      C1LNL   7400
ADFW    4747      BCPUTL  3034      CLOCK   7000      C13     4531
AIROR   0043      BE      0047      CLOSE   3403      C20     4320
AJT     0712      BEND    5572      CLOSED  3445      C2000   6670
AL1     6057      BKHERE  7347      CLOSER  3405      C2400   6671
AL1K    3774      BLINIT  3352      CLPY    1115      C3      4070
AL1P    6240      BLREAD  3336      CLPY1   1122      C4      4047
AL1PP   5140      BMAP    0036      CLSTAG  3544      C770    6676
AL1PT   4705      BO      4210      CMMA    6434      C7760   6675
AL1PTR  5642      BRTSB   1216      CM210   6674      DA      2302
AMODE   2310      BSHFT   0427      CNOBMK  2765      DADP    0060
ANDLST  3401      BSTRT   5547      CNOBML  5573      DATABA  3675
ANDPTR  3400      BSWL    0144      CNOBMP  3104      DATCOM  3613
ANITBL  0620      BSWP    6361      CNOCLL  0146      DATE    3600
ANOTBL  0630      BUFASS  4222      CNOCLR  3016      DATTAB  4525
ANOUT   0656      BUFCHK  2706      CNTBAS  7276      DAY     3742
AN1     3775      BUFCHL  0147      CNTCBP  6666      DAYTBL  4564
AN2     3776      B1      4216      CNTCBS  7310      DBAD    6115
AP0001  2374      B2      4211      CNTCOM  1254      DBAD1   5534
ARGET   6200      CAD     4506      CNTCST  1337      DBAD1P  6175
ARGETK  5503      CAF     6007      CNTCT   1106      DCNT    5137
ARGETL  5444      CALFFP  7243      CNTCTI  7122      DCNTP   4726
ARGETP  6127      CALLBE  0046      CNTERI  1110      DCOS    7520
ARGPLK  4312      CALLF0  7221      CNTFNC  0527      DCOSP   6665
ARGPLL  3511      CALLQW  0061      CNTFUC  7170      DE      3366
ARGPOL  4220      CBLK    3672      CNTIN   1314      DECNV   5213
ARGPRE  0304      CC16    4414      CNTLC   7215      DECON   5214
ARGPRL  1415      CC3     4316      CNTOUT  1277      DECONV  5207
ARGSET  0071      CC4     4315      CNTRCT  1107      DECON1  5335
ARITHA  6373      CDFINL  0534      CNTSW   1077      DEVCAL  0124
ARJMP   0707      CDFIO   0020      CNTSW2  1104      DEVNAL  4317
ARRAYI  0600      CDFPS   0025      COMBNE  2766      DEVNA1  4030
ARSTRT  0022      CDFPSL  0115      COMLOP  2527      DEVNA2  4031
ARTRAP  4366      CDFPSU  0206      COMMA   2512      DGTYP   5076
ASC     3407      CDFXX   2140      COMMAS  2551      DGTYPP  4725
ASCNDE  3243      CDF0    0210      COMMON  0200      DIGOUT  0400
ASCOLK  3457      CDF000  4341      COMONP  0041      DIG1    4125
ASCON   3667      CDF10   6307      COMXIT  0475      DIG1A   3571
ASCOUT  1260      CDF20   4565      CONTAC  1327      DIG2    4126
ATABDF  0610      CDIN    3623      CONTXT  7125      DIG3    4127
ATABDL  1162      CFETCH  1322      COS     4053      DIG4    4130
```

```
DIG5    4131     EF      3367     FD      5527     FIGO2   5241
DISIN   3630     EFATAL  4261     FDDON   5742     FILEFA  6375
DISMIS  0141     EFLG    0056     FDDONP  5531     FILE1   6714
DISPCH  7523     ELOP    7053     FDIVL   4062     FILE2   6731
DIVBY   0217     EM      3615     FDIVM   4361     FILSTR  3311
DIVID   0125     EMDONE  3600     FDIV1L  4067     FILSTU  4306
DLCDF   2303     EMDONL  3570     FDIV1M  4362     FISUBL  4372
DLCDFL  1164     EMESS   4110     FDVPT   5311     FIXDNE  4525
DLPTR   0016     EMLOOP  3545     FD1     5504     FIXL    4066
DLREAD  2275     EN      4274     FD1P    5721     FIXLP   4515
DLRELK  7554     ENTLOK  4262     FERRLP  4011     FJOCI   0400
DLSTOP  0027     ENTNO   0162     FF      0037     FLDW    0057
DLSTRT  0030     ENTRYN  4032     FFADD   6000     FLEN    4273
DNA1    3621     ENVAL   3516     FFADP   4733     FLING   5142
DNA2    3622     EOFSEL  0143     FFATN   4200     FLINK   4722
DNUMBR  5305     EOFSET  2236     FFCOS   4053     FLN     3637
DO      4045     EOSPA   6677     FFDIV   5722     FLOAT   0562
DOADD   6023     ERRDIS  1457     FFDIV1  5412     FLOATB  3420
DONA    6027     ERRET   4050     FFDP    5446     FLOATL  3645
DONE    3765     ERROR   0116     FFDVP   4734     FLOATM  4356
DONEF   5066     ERRORR  4002     FFD1    5726     FLOOK   4236
DOWRK   6466     ESHFT   0435     FFEXP   4120     FLUSH   6425
DR      1067     ESTRA   4134     FFEXPL  3625     FM      1624
DRARG1  0547     ESTRNG  4112     FFGET   6241     FMPYL   3572
DRARG2  0550     ETAB    4137     FFIN    5200     FMPYLK  4061
DRARG3  0551     ETABA   4135     FFINLK  3476     FMPYLL  5310
DRCALL  0532     ETLOP   4033     FFIN1   5232     FMPYLV  3627
DRERR   3365     EXP     0044     FFIX    4500     FMPYM   4357
DRERRP  0555     EXPA0   4422     FFLOAT  4533     FN      2005
DRIVRL  4046     EXPA1   4417     FFLOG   4263     FNAP    3441
DRIVRN  4200     EXPB1   4414     FFLOGL  3626     FNDMCH  0317
DS      4411     EXPON   3477     FFMPP   4735     FNEGI   1226
DSWIT   0052     EXPONK  3630     FFMPY   5600     FNEGL   0140
DT1     5101     EXPON1  4120     FFMT    4613     FNLP    6334
DV      6355     EXPP    0040     FFNEG   6135     FNOM    4271
DVLP1   5711     E20P10  1307     FFNEGA  5410     FNORL   0136
DVL1    5514     FACCLR  0362     FFNEGK  5501     FNORP   6176
DVOPS   6315     FACR    6031     FFNEGP  5303     FO      1637
DVOPSP  5533     FACREL  0133     FFNEGR  5773     FOTYPE  2355
DVOP1   6330     FACRES  2370     FFNGP   4736     FOUT1   4625
DVOP2   5535     FACSAL  0132     FFNOR   6215     FOUT2   4634
DVOP2P  6333     FACSAV  3361     FFNORR  6236     FOUT3   4674
DVOVR   5777     FADDL   4060     FFORMT  1302     FOUT4   4667
DVTRAP  3574     FADDLK  3473     FFOUT   4600     FPPARG  7246
DV1     5765     FADDLL  5312     FFPUT   6256     FPPPNT  0035
DV2     5755     FADDM   4360     FFPUTP  0135     FPPTM1  1164
DV24    5745     FAD1    6004     FFSIN   4000     FPPTM2  1161
DV24P   5532     FATAL   6771     FFSQ    6347     FPPTM3  1156
DV24PT  5136     FATCHK  4040     FFSUB   6117     FPPTM4  1153
EAE     0000     FB      4012     FFSUB1  5400     FPPTM5  1150
EATONE  6414     FBITGT  6547     FGETL   0134     FPPWRK  7251
EBC     2730     FBITS   0117     FGETM   0134     FPUTL   0135
EBLK    3737     FC      3443     FI      2273     FPUTLL  0720
EDBLK   3746     FCLR    0137     FIDLE   0123     FPUTM   0135
EDON    5260     FCNT    5275     FIDVP   3631     FRACT   4071
```

```
FRANDM 2341       GOSUB  0415       JMPISN 4012       K0210  3205
FROOT  3646       GOTSPT 1036       JMPI2  0744       K0300  4057
FSQRL  4065       GR     0470       JMPI3  1205       K0340  0100
FSQRM  4365       GS     0417       JMPI6  1604       K0377  0101
FSTOP  3700       GSP    0447       JMPUSR 1557       K1400  4313
FSTOPI 0557       GSPOT  0003       JMSI   0244       K16    4721
FSTOPN 0556       GSTCK  2314       JMSI4  1417       K20    3266
FSTOPP 0560       GTFLG  4200       JMSI5  1432       K200   0077
FSTOP1 0161       GT1FLG 4263       JMSI7  1540       K2000  4314
FSUBL  4064       HANG10 7565       JMSSI  0303       K215   0054
FSUBLL 3573       HGHCLK 7115       JMSTAD 1531       K232   3461
FSUBM  4363       HGHTME 7113       JNEG   3466       K240   3204
FSUB1L 4063       HOOKL  3776       JSL    2627       K260   4056
FSUB1M 4364       HORD   0045       JUSNEG 3464       K2700  3700
FSWITC 2050       HORDP  0036       KC     3751       K300   6711
FTCOM  2424       IA     1464       KC240  1140       K4207K 4350
FTFLG  6531       IC     0056       KEX    3567       K5700  3702
FTRPRT 4505       IDLE   2270       KFD1   5447       K6     4740
FTYL   3015       IF     4424       KKK240 3573       K6000  3772
FTYPE  6555       IG     0070       KKM10  3704       K6213K 4351
FTYPL  0150       IGS    3534       KKM12  5775       K6222  4572
FTYPSE 3462       ILOOP  0212       KK12   5313       K6223  4570
FUDXXX 1406       ILOOPF 0236       KK13   4333       K73    4564
FUDXX1 1416       ILOOPL 0113       KK2000 4067       K7400  0102
FUDXX2 1405       ILOOP2 0214       KK40   3265       K7477  0104
FUJUMP 1523       IN     4033       KK7    4730       K7506  5346
FUNC1I 1467       INALOG 0277       KK7600 1253       K7554  0436
FUNC2I 1466       INBFP  0007       KL7600 3744       K7577K 4352
FUNC3I 1535       INITFN 6427       KME    5306       K7600  0437
FUNC4I 1550       INPHK  6521       KM12   5135       K7607K 3677
FUNC5I 1465       INPTCL 3477       KM13   5776       K7700  0103
F1I16  0016       INPUT  5347       KM144  5134       K7760  3662
F1I17  0017       INSAV  0065       KM15   3754       K7773K 4353
F1SAV  0020       INSAVP 0034       KM20   4731       LASTB  3325
F1SAV2 0021       INSC   3413       KM22   3777       LDH    0131
F1SAV3 0022       INT    3400       KM4    0004       LDHC   2075
F1TMP1 0023       INTERB 1141       KM40   0105       LDHCDL 4504
F1TMP2 0024       INTL   0114       KM7    4732       LDHDF  2647
F1TMP3 0025       INTPC  0035       KNT    4737       LDHINI 2676
F1TMP4 0026       INTPOS 3420       KNTP   5154       LDHINL 0127
GAINS  0353       INTRET 6764       KSKP   4110       LDHL   2646
GC     7274       IOTAB  2031       KSKP2  4000       LDHPR  4501
GCHR   5322       IOUT   5360       KUPARO 3750       LDHPST 2133
GCSE   1153       ISZAC2 6310       KYBRD  7201       LDHPSW 4502
GCSE1  1157       ISZFGT 6275       K0     4412       LDHR   2705
GC23   7307       IS2BIG 0065       K0001  3464       LDHRST 0157
GD     5025       IV     0060       K0007C 3703       LDHSWT 2704
GE     0045       IVGAIN 0067       K0010  0073       LEN    3414
GENERR 0053       I1LNL  7410       K0017  0074       LEV    5477
GETCH  3125       JEOFI  0450       K0037  0740       LF     6452
GETCHG 3501       JFAIL  0413       K0037C 3677       LINEHI 0066
GETCHL 0142       JFOR   2034       K0057  3701       LINEI  1113
GETE   5250       JMPFIL 1446       K0077  0075       LINEI1 1127
GKNT   5003       JMPI   0245       K0100  0076       LINELO 0067
GON    3761       JMPISA 0734       K0200  0077       LK7607 3745
```

```
LM      6360    MCTRLZ 3103     NOCTC   3724    PATCHP 0155
LMAKE   4060    MDNE    3226    NOCZ    3420    PDNE    3216
LMAKEL  3566    MDONE   5627    NOP1    6111    PDP     5001
LNE     0524    MDSET   5450    NOP2    6053    PHYCHN  0027
LNEFUC  7167    MDSETK  5774    NORMLP  6225    PINFO   0356
LNKSVE  6747    MDSETP  5445    NOTE    4743    PIOV2   4403
LNSET   1072    MDV     5307    NULLST  4370    PLUS    5362
LN2     4471    MD1     5452    NUM     4113    PNT     1763
LN2OV2  4411    MD1P    5443    NXTARG  7233    POLYNL  4304
LOADDF  3163    MFATAL  4136    N1      4403    POLYSN  4026
LOG     4263    MINUS   5363    N1A     4542    POS     4400
LOGC1   4455    MINUSP  3460    N2      4404    POSITN  4500
LOGC3   4460    MINUS3  0355    N3      4405    POSSET  4424
LOGC5   4463    MIN4    4354    N3A     4402    POVTAB  0357
LOG2E   4406    MK61    2727    N4      4406    POWER   3472
LOOP    5052    MML     4501    N5      4407    PR      5016
LOP01   3722    MMM4    4415    N6      4410    PRDCP   4776
LOP02   3746    MM260   3572    N6A     4543    PRDCPP  5143
LOP1    6075    MNITE   7112    N7666   3611    PREST   3542
LOP2    6037    MNITEP  3757    OACHR   6455    PRNTX   5160
LORD    0046    MNTHCK  3760    OADD    6157    PRNTXP  4723
LORDP   0037    MODEBK  0064    OADDP   5141    PRNTX1  5164
LOWCLK  7116    MODESW  0063    OATADI  1526    PRZRO   5172
LOWTME  7114    MONTH   3755    OCDF    2136    PRZROP  4724
LPY     1026    MPLP    5653    OE      1512    PS      5023
LRD1    7361    MPLP1   5654    OLP     7015    PSFLAG  0031
LRESET  2165    MPLP2   5666    ONE     3474    PSSTRT  0026
LRSCOM  2655    MPY     2244    ONEHAF  4466    PSWAP   1230
LSUB1I  1404    MPYLNK  0121    ONERET  4472    PSWAP2  4321
LSUB2I  1413    MP12L   5701    ONE1    4475    PSWP2P  4544
LS1I    1401    MP12LP  2251    OPENAF  4001    PS1L    0360
LS2I    1400    MP24    5643    OPENAV  4000    PS2L    0361
LTRPRT  4302    M1      4727    OPENNF  4004    PTR     7061
L7466   3676    M1R     3470    OPENNV  4003    PTRBMP  2156
L7600K  4355    M13     4530    OPERI   1200    PTR1    0033
L7605K  3664    M14     0106    OPH     0050    PUSKP   0411
L7605P  4356    M215    0055    OPL     0051    PUTCH   3247
L7607   1527    M240    5361    OPNEG   6146    PUTCHL  0112
L7620   3661    M6      4413    OPNEGP  5502    PWFECH  0200
L7621   3663    NAMEG   4416    OPSR    6034    PWFECL  0120
L7642   4571    NAMEGL  4050    OPX     0047    PWRFLD  7574
L7644   3660    NCG     4450    OTRAPA  4532    PWRJMP  0107
L7721   4573    NCGS    4464    OUT     5144    PWRNAM  3552
L7727   4574    NCHK    4113    OUTDG   5150    PWRUP   0110
L7746   0354    NCHKL   4262    OUTDGP  4720    PZR     5015
MAKED   4052    NEWCDF  6301    OVADD   1511    P1CDF   1240
MASKI   0031    NEWTAG  7330    OVDNE   1516    P1CDF1  1245
MASKL   3427    NEXRCK  4260    OVERLA  3400    P1SWAP  0156
MATCHL  1126    NEXREC  3275    OVML    0343    P2CDF   1243
MCC     3747    NEXREL  0152    OVRLAY  1530    P2CDFL  4566
MCDF1   6667    NFLAG   4120    OO      6354    P2CDF1  1247
MCNTLC  7220    NFLGST  4111    PA      4415    P2CDL1  4567
MCOLON  4400    NGT     4257    PACKCH  2735    P2SWAL  3743
MCRMAL  3456    NHNDLE  4103    PACKL   0145    P200    5211
MCSPE   4401    NHNDLL  4261    PATCHF  5364    QUAD2   4017
```

```
QUAD3   4022     SDIS    0273     STCGTJ 3500     TDI     0476
QUAD4   4024     SDOTBL  0610     STCOM   1676     TDOFF1 1503
QW      0062     SE      4706     STC40C 2124     TDOFF2 0543
RDIGIN 0453     SEG     4271     STDF    1677     TDONE   3553
RDIGOU 0501     SEGCML 4373     STDFL   1163     TDSRCH 7141
RDITBL 0600     SEGCOM 2222     STFILK 0741     TD8P1   6672
RDLIST 7545     SEP     5157     STFIND 1664     TD8P2   6673
RDOTBL 0640     SEP1    0256     STFINK 4503     TE      0066
RE      3010     SETF    2044     STFINL 0302     TEMP1   0040
READFL 3000     SETTTY 6532     STH     0130     TEMP10 0061
READFW 3145     SETUP   1343     STHDF   2603     TEMP11 0062
READI   3105     SE1     4707     STHDKK 0374     TEMP17 3023
READIT 0735     SFN     2000     STHINI 2636     TEMP18 3016
RELUSR 4261     SFNLP   2023     STHINL 0126     TEMP2   0006
REPOWR 7555     SG      0054     STHL    2600     TEMP21 2636
RESDLS 2552     SGN     3632     STHR    2645     TEMP24 0525
RESTI   1650     SHFTNO 0032     STHRST 0160     TEMP3   0042
RESTOR 2555     SHLFT   5635     STHSWT 2644     TEMP4   0043
RETMDL 3674     SIGNF   5300     STOPEM 1107     TEMP5   0047
RETMOD 3521     SIN     4000     STOPI   4051     TEMP6   0050
RETRNI 0456     SINA1   4367     STOR    6750     TEMP7   0051
RETRNO 3571     SINA3   4372     STPCNT 1246     TEN     5317
RETRN1 3610     SINA5   4375     STPTMR 1161     TICKS   0364
RET0    3565     SINA7   4400     STR     3422     TIME    0677
RIGHTL 2665     SL      0521     STRCNT 0071     TIMER   1000
RIGHTS 2633     SLOAD   3146     STRLEN 0032     TIMEX   1105
RIPTR   2455     SLOOP   3713     STRMAX 0070     TIMINT 7117
RND     4544     SLOVER 2541     STRNGA 6374     TINCNT 1105
RO      6437     SLRCOM 2614     STRPTR 0072     TINCN2 7121
RONLY   3307     SMODE   0266     STSLP   3440     TM      0043
RO6     0251     SNEQ    2117     STSTRT 0023     TMPY    6353
RSEED   2345     SNEQ1   2116     SU      0623     TMRCNT 7060
RSEEDL 4563     SPFUNC 1600     SUB0    6125     TMREND 7055
RTN2    5371     SPINNR 0017     SUB0P   5411     TMRSCN 7011
RTZRO   5620     SQRP5   4452     SUCJMP 0426     TMRTM1 7056
RWONC   3305     SR      3127     SUPFUD 6400     TMRTM2 7057
SAC     0316     SRCLP   4445     SW      3256     TM3     5155
SACCHK 2122     SREAD   2416     SWCLP   2477     TM3PT   5304
SACEM   2216     SRESET 0370     SWITCC 0042     TO      0064
SACL    3421     SRFIN   2447     SWIT1   0054     TOOBIG 0256
SACPTR 0111     SRLIST 2400     SWIT2   0055     TOOMNY 0063
SAC40C 2106     SRLOOP 2411     SWRITE 2460     TOUT    3272
SAD     0733     SSAD    4525     S1      0033     TOVPI   4160
SAFIND 1745     SSLOOP 0510     S1PRAY 6527     TO2BIG 0042
SARRAY 0722     SSLP    3156     S2      0034     TP      5314
SASTRT 0024     SSMODE 2307     S2PRAY 6530     TPRINT 3524
SC      2232     SSTEX   0523     TAB     1753     TP1     5315
SCALDF 0314     SSTORE 0473     TABLE   0226     TRACE   3770
SCALDL 1165     ST      2444     TADTAB 1532     TRHOOK 1134
SCASE   0241     STAENT 0465     TAGIT   0750     TRPRET 1065
SCDF    1735     STARTB 4272     TAGITP 0352     TRREST 3775
SCOMP   2051     START3 1141     TBLDYP 3767     TRYCLS 3526
SCONTU 4462     START4 6600     TBLSCN 0233     TSMET   7052
SCON1   2200     STATE   0551     TCLFLG 6760     TSTTTY 1042
SCSTRT 0021     STB     3636     TDFIXL 4347     TTCHCT 1032
```

```
TTEST2 4001       UDSC   6353       XDRITE 4522
TTFULP 1003       UDSF   6361       XFLOAT 4505
TTGET  3141       UDSS   6351       XIT    0450
TTGETP 1031       UNPACK 3047       XLCOM  4514
TTMORE 1055       UNSFIX 1615       XPUT   0122
TTPUTP 1030       UNSLP  1642       XPUTCH 1000
TTYBUF 6600       UNSOUT 1646       XR0    0010
TTYCHK 1034       UP     0134       XR1    0011
TTYCHX 3565       UPDATE 3705       XR2    0012
TTYEND 6677       UPDAY1 3715       XR3    0013
TTYF   6712       UPDAY2 3720       XR4    0014
TTYINB 6600       UPLVL  6542       XR5    0015
TTYSIZ 1033       USE    2562       XX110  2340
TTYTCF 1053       USECON 0005       XX212  2313
TTYUSR 4111       USELOG 3613       XX4    2312
T1H    7063       USELOL 3567       X7607  1235
T1HR   7073       USR    0077       YEAR   3756
T1L    7062       USRCAL 4072       ZAPION 1513
T1LNH  7103       USRCP  3470       ZCNT   3773
T1LNL  7102       USRERR 6763       ZEXP   6341
T1LR   7072       USRREL 4103       ZMINY  4334
T2H    7065       USRRP  3471       ZR     1342
T2HR   7075       U123C  3056       ZRCONT 1352
T2L    7064       VAL    3461       ZROFF  1334
T2LNH  7105       VALCNT 3422       ZRORET 4470
T2LNL  7104       VALGET 3502       ZZZZ   7357
T2LR   7074       VALLK  0154
T3H    7067       VALUE  0057
T3HR   7077       VR     3003
T3L    7066       WDONE  3245
T3LNH  7107       WE     3032
T3LNL  7106       WHO    7535
T3LR   7076       WIDTH  0536
T4H    7071       WORD0  0163
T4HR   7101       WORD1  0172
T4L    7070       WORD10 0165
T4LNH  7111       WORD11 0166
T4LNL  7110       WORD12 0167
T4LR   7100       WORD13 0170
UDCBAD 0050       WORD14 0171
UDCHNL 0107       WORD2  0173
UDCI   7254       WORD3  0174
UDCM3  6664       WORD4  0175
UDCTAG 7257       WORD5  0176
UDCTM1 7362       WORD6  0177
UDCTM2 7363       WORD7  0164
UDCWRK 7306       WRBLK  3342
UDDI   6365       WRBLKK 3463
UDEI   6364       WRDA   3112
UDLA   6363       WRITEI 3200
UDLD   6367       WRITFL 3023
UDLS   6357       WRITFW 0125
UDRA   6356       W0PTR  3466
UDRD   6366       W0PTRA 3465
UDRS   6355       XDGET  4506
```

INDEX

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections, are published by Software Information Service in the following newsletters.

DIGITAL Software News for the PDP-8 and PDP-12
DIGITAL Software News for the PDP-11
DIGITAL Software News for 18-bit Computers

These newsletters contain information applicable to software available from DIGITAL'S Software Distribution Center. Articles in DIGITAL Software News update the cumulative Software Performance Summary which is included in each basic kit of system software for new computers. To assure that the monthly DIGITAL Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest DIGITAL office.

Questions or problems concerning DIGITAL'S software should be reported to the Software Specialist. If no Software Specialist is available, please send a Software Performance Report form with details of the problems to:

Digital Equipment Corporation
Software Information Service
Software Engineering and Services
Maynard, Massachusetts 01754

These forms, which are provided in the software kit, should be fully completed and accompanied by terminal output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual, and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest DIGITAL field office or representative. USA customers may order directly from the Software Distribution Center in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information, please write to:

Digital Equipment Corporation
DECUS
Software Engineering and Services
Maynard, Massachusetts 01754

OS/8 Industrial BASIC
DEC-S8-OSIBA-A-D

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback--your critical evaluation of this document.

Did you find errors in this document? If so, please specify by page.

____

____

____

____

____

How can this document be improved?

____

____

____

____

____

How does this document compare with other technical documents you have read?

____

____

____

____

____

Job Title______________________ Date:____________

Name:____________________ Organization:____________

Street:____________________ Department:____________

City:______________ State:________ Zip or Country________

-- Fold Here --

-- Do Not Tear - Fold Here and Staple --

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Digital Equipment Corporation
Software Information Service
Software Engineering and Services
Maynard, Massachusetts 01754