Structured Programming ROM

Manual Part Number 09845-93066
Microfiche Part Number 09845-96066

Hewlett-Packard Desktop Computer Division
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

This manual is for use with the System 35A/B or 45B/C Desktop Computers. It is a slightly revised version of the Structured Programming ROM Manual, part number 09835-90066.

The changes which were incorporated into this latest edition are summarized in the System 45 Manual Revision Package (P/N 09845-93099). This package outlines the changes and additions that have been made to System 45 manuals.

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change). The manual part number changes when extensive technical changes are incorporated.

April 1981...First Edition; Updated pages: ii, 2, 14, 17, 22, 26, 28, 57

Table of Contents

Chapter 1: Overview

Chapter 2: Structured Looping Constructs

Chapter 3: Decision Constructs

Chapter 4: Programming Aids

Chapter 5: Error Messages

Chapter 1

Overview

The Purpose of Structured Programming

Concepts embodied in Structured Programming include top-down organization, efficiency in developing a program and ease of understanding the program.

This is achieved using five programming constructs (a construct being a group of logically related BASIC statements). Three constructs are used for loop control and two are for decisions. Two programming aids are available, one for indenting program listings and one to cross-reference identifiers. With these tools, it is very easy to emulate languages such as PASCAL, with most of their benefits while retaining the friendly, interpretive BASIC of your Series 9800 Desktop Computer.

Advantages of Structured Programming

- Logical organization facilitates documentation, modification and maintenance.
- Easy to learn and use.
- Faster to debug than unstructured programs.
- Powerful syntax.
- Smooth program flow (the need for GOTO's is reduced).
- Reduces program development time.
- Compiler errors are non-existent.

The Purpose of This Manual

This manual is written for a programmer that is familiar with HP BASIC as it is implemented on the Series 9800 Desktop Computers. You should be familiar with loops, decisions, and callable sub-program segments, as well as the operation of your Desktop Computer.

This manual is not intended to be a tutorial on structured programming. There are many textbooks available which explain structured programming in detail.

The manual is divided into 7 chapters:

1. An overview of the manual.
2. Structured Looping (WHILE..END WHILE; REPEAT..UNTIL; LOOP..EXIT IF..END LOOP)
3. Structured Decisions (IF..THEN..ELSE..END IF; SELECT..CASE..CASE ELSE..END SELECT)
4. Programming Aids (INDENT; XREF)
5. Errors (causes and solutions)
6. Syntax Information
7. Example Programs

ROM Installation

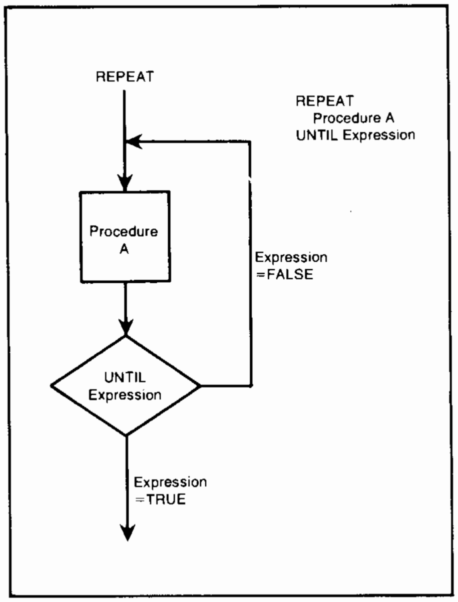
The statements which are described in this manual cannot work unless you have the Structured Programming ROM installed in your Series 9800 Desktop Computer. The installation procedure is quite simple. Information concerning the installation of your Structured Programming ROM for your series 9800 Desktop Computer can be found in the System 35 Owner's Manual or the System 45 Installation, Operation and Test Manual.

Equipment Supplied

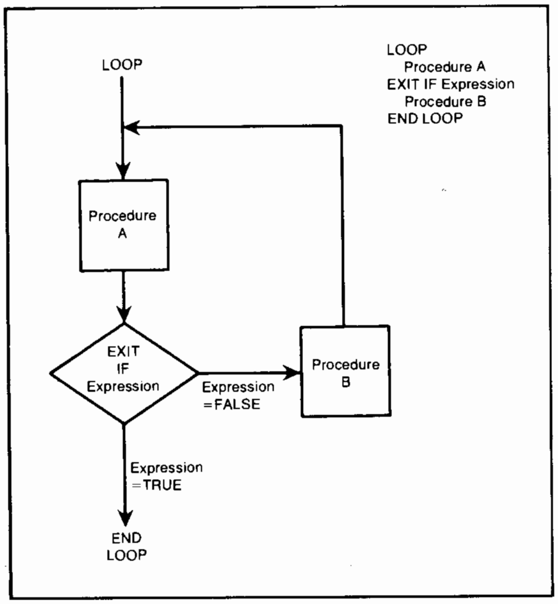
HP 9835 Part Number	HP 9845B/C Part Number	Item
98335A	98415A	Structured Programming ROM
09845-93066	09845-93066	Structured Programming ROM Manual
09835-67991	–	System 35 Six-Slot ROM Drawer

Chapter 2

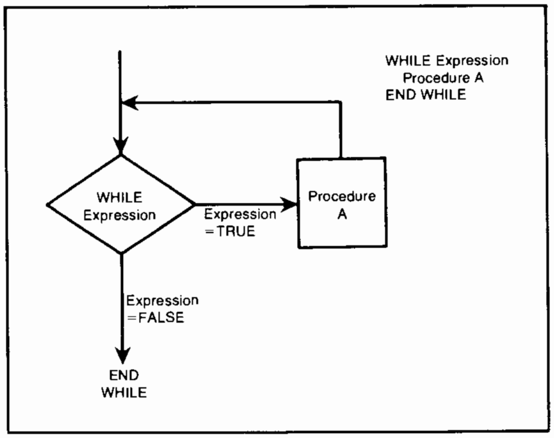
Structured Looping Constructs



```
REPEAT
  Procedure A
UNTIL Expression
```



```
LOOP
  Procedure A
EXIT IF Expression
  Procedure B
END LOOP
```



```
WHILE Expression
  Procedure A
END WHILE
```

What Do They Do?

A looping construct is one which causes an action to be repeated until a condition is met.

A looping construct is active –

- after its WHILE, REPEAT or LOOP statement is executed and until it is deactivated.

A looping construct is deactivated –

- when it is properly exited (its loop test condition is satisfied), or
- when program execution encounters the next higher level of structured loop nesting, or
- when a RETURN is made for a GOSUB which was executed prior to the activation of the looping construct, or
- when a RETURN or SUBEXIT is made from the subprogram containing the looping construct.

Where Would I Use Them?

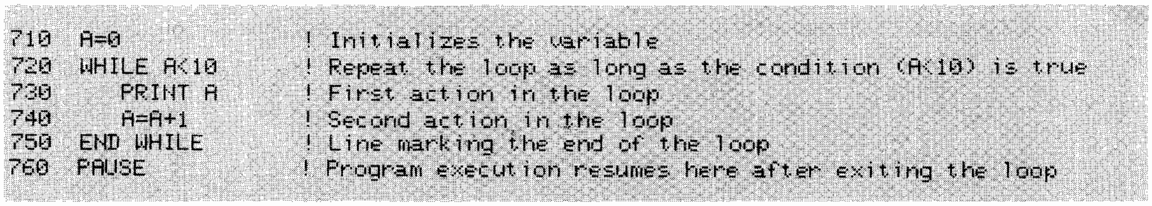
Structured looping constructs are used whenever you have a procedure in your program which has to be repeated. Anything which is iterative in nature (such as polynomial expansion) can be done efficiently using structured looping. Your Structured Programming ROM provides you with the three structured looping constructs shown in the accompanying table.

Construct	**Reason For Use**
WHILE...END WHILE	To test the condition **before** executing the loop.
REPEAT...UNTIL	To test the condition **after** executing the loop.
LOOP..EXIT IF..END LOOP	When there are multiple conditions which can result in the termination of the loop, or when the test occurs in the middle of the loop.

The Statements

WHILE...END WHILE

The WHILE...END WHILE construct allows a loop to be repeated as long as its condition is true (not equal to 0). For example,

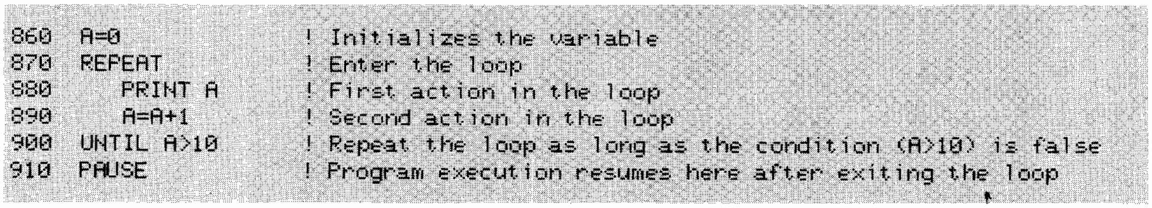


```
710  A=0                    ! Initializes the variable
720  WHILE A<10             ! Repeat the loop as long as the condition (A<10) is true
730     PRINT A             ! First action in the loop
740     A=A+1               ! Second action in the loop
750  END WHILE              ! Line marking the end of the loop
760  PAUSE                  ! Program execution resumes here after exiting the loop
```

As you can see in this example, the loop consists of printing the value of A, and then incrementing the value of A. At the beginning of the loop, the value of A is tested to see if it is less than 10. If A is less than 10, the loop is executed; if A = or > 10, the loop is exited. This results in the numbers 0 thru 9 being printed.

REPEAT...UNTIL

The REPEAT...UNTIL construct allows a loop to be repeated as long as its condition is false (equal to 0). For example,



```
860  A=0                    ! Initializes the variable
870  REPEAT                 ! Enter the loop
880     PRINT A             ! First action in the loop
890     A=A+1               ! Second action in the loop
900  UNTIL A>10             ! Repeat the loop as long as the condition (A>10) is false
910  PAUSE                  ! Program execution resumes here after exiting the loop
```

As you can see in this example, the loop action consists of printing the value of A, and incrementing the value of A. The loop is entered with the REPEAT statement, and the loop action is performed. When the UNTIL statement is executed, the condition (A>10) is tested. In this way, a REPEAT...UNTIL loop is always executed at least once. If the condition is false, the loop is repeated; if the condition is true, the loop is exited. The result of this loop is the printing of the numbers from 0 thru 10.

LOOP...EXIT IF...END LOOP

The LOOP...EXIT IF...END LOOP differs from the previous loops in its testing of the condition. The LOOP and END LOOP mark the beginning and ending of the loop, while the testing of the loop is performed whenever the EXIT IF statement is executed. If the condition tested for is true (non zero), the loop is exited. The LOOP...EXIT IF...END LOOP allows you to test for more than one condition without affecting the readability of your program. For example,

```
1080 A=0                  ! Initializes the variable
1090 B=12                 ! Initializes the variable
1100 LOOP                 ! Beginning of the loop
1110   C=B-A              ! First loop action
1120   PRINT A,B,C        ! Second loop action
1130 EXIT IF C=7          ! First conditional test
1140   B=B-1              ! Third loop action
1150   A=A+1              ! Fourth loop action
1160 EXIT IF A>B          ! Second conditional test
1170 END LOOP             ! End of the loop
1180 PAUSE                ! Program execution resumes here after exiting the loop
```

In this example, the loop actions consist of a simple number-crunch (C=B−A), printing of the values for the variables, a test for a specific value of C, updating the variables, and a test for A>B.

In this case, the results are shown here –

```
0        12        12
1        11        10
2        10         8
3         9         6
4         8         4
5         7         2
6         6         0
```

The loop was exited because the second conditional test (A>B) was true.

A Few Words About FOR...NEXT Loops

FOR...NEXT loops, which are a mainframe capability, may also be considered similar to structured looping constructs. A FOR...NEXT loop has one entry point (the FOR statement), and one exit (the line after the NEXT). The loop is repeated as long as the loop counter in the FOR statement is less than or equal to its final value.

FOR...NEXT loops are different from the structured looping constructs in having a loop counter which, when the FOR...NEXT is exited, has some value greater than the final value. REPEAT...UNTIL and LOOP...EXIT IF...END LOOP constructs exit when their expressions resolve to a non-zero value. The WHILE...END WHILE construct has its expression resolve to 0 to exit the loop. Another difference between FOR...NEXT and structured looping constructs is that a RETURN statement does not deactivate a FOR...NEXT loop, while it does deactivate structured looping constructs.

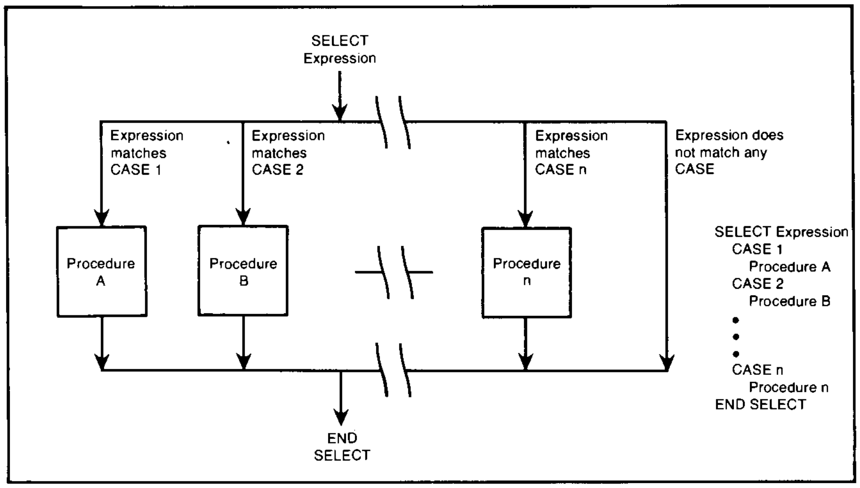
How Are They Similar?

The looping statements are similar in the following manner –

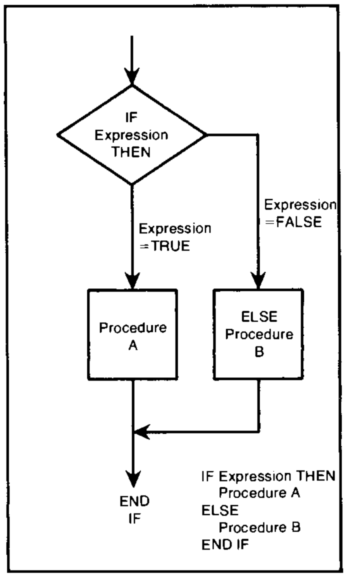
- All loops exist within a defined beginning and ending: WHILE, REPEAT, and LOOP are the beginning of the loops; END WHILE, UNTIL, and END LOOP are the ending of the loops.
- Conditions are always tested:
  At the WHILE statement for WHILE...END WHILE
  At the UNTIL statement for REPEAT...UNTIL
  At the EXIT IF statement for LOOP...END LOOP
- If a loop is entered in the middle (i.e., by a GOTO statement) an ERROR 348 occurs the first time the END WHILE, UNTIL, or END LOOP for that loop is encountered.
- All active loops begin and end with "busy" program lines. This means that once a loop is active, the beginning and ending lines of the loop cannot be edited (this is similar to INPUT).

Chapter 3

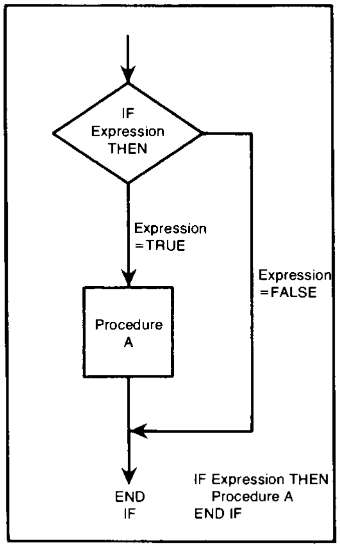
Decision Constructs



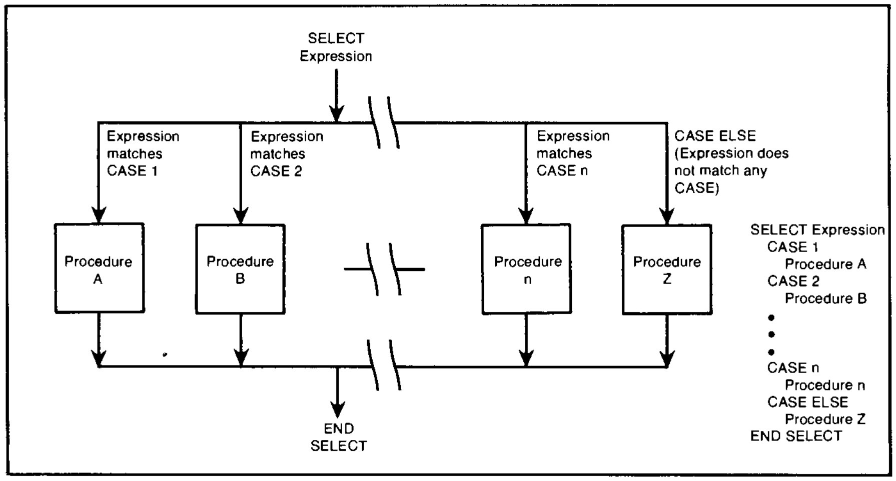
```
SELECT Expression
   CASE 1
      Procedure A
   CASE 2
      Procedure B
   •
   •
   •
   CASE n
      Procedure n
END SELECT
```



```
IF Expression THEN
   Procedure A
ELSE
   Procedure B
END IF
```



```
IF Expression THEN
   Procedure A
END IF
```



```
SELECT Expression
   CASE 1
      Procedure A
   CASE 2
      Procedure B
   •
   •
   •
   CASE n
      Procedure n
   CASE ELSE
      Procedure Z
END SELECT
```

What Do They Do?

Structured decision constructs provide you with alternatives which are available depending upon the results of a conditional test. Decision constructs also have clearly defined entry and exit points. Rather than requiring you to execute a GOTO for your next action, the program exits at the end of the decision construct.

Where Would I Use Them?

Structured decision constructs can be used wherever alternatives are needed because of a test condition. For example, a hypothetical action could be as shown here –

```
1570 IF Coffee_cup=Empty THEN
1580   More_coffee$="YES"
1590 ELSE
1600   More_donuts$="YES"
1610 END IF
```

The result of this simple example is that if the coffee cup is empty you ask for more coffee, otherwise you would ask for more donuts. As you may infer, with a structured decision of this type you are limited to an exclusive OR condition (one action OR the other, but not both). There also is another structured decision statement which offers several actions from which to choose, dependent upon which particular case of the condition is currently being used. Both are shown in the following table.

Construct	**Reason for Use**
IF..THEN..ELSE..END IF	There are either one or two distinct actions which should be taken depending upon the test condition.
SELECT..CASE..CASE ELSE..END SELECT	There are more than two conditions each of which results in a separate action.

The Statements

IF...THEN...ELSE...END IF

The IF...THEN...ELSE...END IF construct provides up to two actions which can be performed based upon the test condition. The test condition is tested at the IF statement, and if the test is true (result not equal to 0), the first action is performed, as shown here –

```
1810 A=1                ! Initializes the variable
1820 IF A THEN          ! Test if A<>0
1830   PRINT "A<>0"     ! First action
1840 END IF             ! End of the structured decision
1850 PAUSE              ! Program execution resumes here after exiting
```

There may be times when you want an alternative action performed inside the structured decision, as shown here –

```
1880 INPUT "A",A        ! Set value for variable A
1890 INPUT "B",B        ! Set value for variable B
1900 IF A>B THEN        ! Enter this construct here
1910    PRINT "A>B"     ! First action performed if condition is true
1920 ELSE               ! Entry for alternate action
1930    PRINT "A<=B"    ! Alternate action performed if condition is false
1940 END IF             ! End of the structured decision
1950 PAUSE              ! Program execution resumes here after exiting
```

Here one of the two actions is performed, regardless of the values of A or B. If more than two separate actions are required, you may prefer using the SELECT..END SELECT construct.

SELECT...CASE...CASE ELSE...END SELECT

The SELECT...CASE...CASE ELSE...END SELECT construct provides alternative actions which are chosen by the SELECT statement. For example –

```
2030 INPUT "PLEASE CHOOSE TYPE 1,2,3 OR 4",A     ! Initialize the variable
2040 SELECT A                      ! Make decisions based upon the value of A
2050 CASE 1                        ! If A=1 then do the next action
2060   PRINT "CASE 1-";A           ! Print only if A=1
2070 CASE 2                        ! If A=2 then do the next action
2080   PRINT "CASE 2-";A           ! Print only if A=2
2090 CASE 3,4                      ! If A=3 or A=4 then do the next action
2100   PRINT "CASE 3-";A           ! Print only if A=3 or A=4
2110 CASE ELSE                     ! In all other cases for A,do the next action
2120   PRINT A;" WAS NOT A VALID CHOICE"    ! Print only if A<>1,2,3,or 4
2130 END SELECT                    ! The structured construct ends here
2140 PAUSE              ! Program execution resumes here after exiting
```

The previous example showed how to use cases to test for an input value. As you can see, you can test for one or more cases in a line as well as testing for all other cases simultaneously. In short, the SELECT statement is similar to executing an IF...THEN; you just have a large number of CASE statements available to you.

How Are They Similar?

The IF...END IF and SELECT...END SELECT constructs are similar in that –

- The resolution of the test condition determines which actions are to be performed.
- The ELSE and CASE ELSE statements define which action is performed if the test condition is not satisfied (i.e., FALSE for IF..THEN and unmatched for SELECT).

Chapter 4
Programming Aids

What Do They Do?

Programming aids make program development easier and your Structured Programming ROM provides you with two such aids, INDENT and XREF.

Where Would I Use Them?

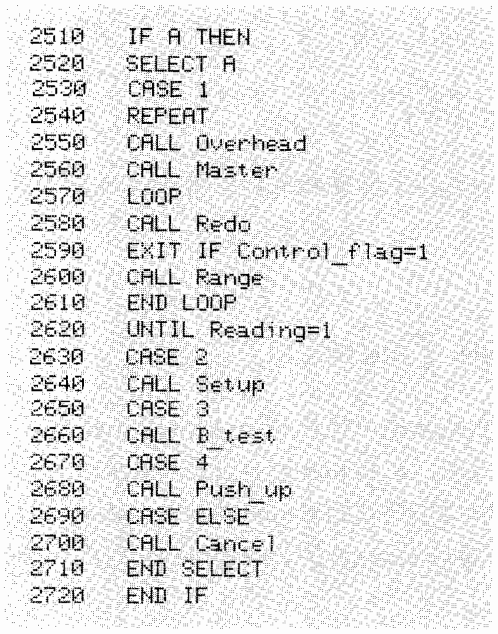
INDENT causes your Series 9800 Desktop Computer to indent all structured loops and decision constructs as your computer would understand them. In this way, if you are ever concerned as to which level contains a particular action, the INDENT command shows where the computer thinks you meant to put it. INDENT also makes reading structured programs much easier.

XREF provides you with a cross-referenced listing of the identifiers which are in your program. This includes variables, arrays, line numbers and constants. By using the XREF statement you can determine, for example, which variables are declared, where they are used, and other things programmers need to know.

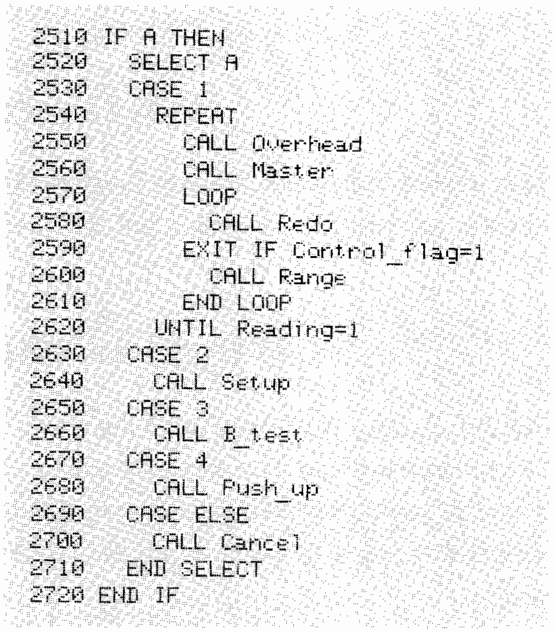
The Aids

INDENT

When writing programs which use nested constructs, the occurrence of the construct, where you expect it, helps achieve a working program. For example, which would you prefer to read?



```
2510   IF A THEN
2520   SELECT A
2530   CASE 1
2540   REPEAT
2550   CALL Overhead
2560   CALL Master
2570   LOOP
2580   CALL Redo
2590   EXIT IF Control_flag=1
2600   CALL Range
2610   END LOOP
2620   UNTIL Reading=1
2630   CASE 2
2640   CALL Setup
2650   CASE 3
2660   CALL B_test
2670   CASE 4
2680   CALL Push_up
2690   CASE ELSE
2700   CALL Cancel
2710   END SELECT
2720   END IF
```



```
2510 IF A THEN
2520   SELECT A
2530   CASE 1
2540     REPEAT
2550       CALL Overhead
2560       CALL Master
2570       LOOP
2580         CALL Redo
2590       EXIT IF Control_flag=1
2600         CALL Range
2610       END LOOP
2620     UNTIL Reading=1
2630   CASE 2
2640     CALL Setup
2650   CASE 3
2660     CALL B_test
2670   CASE 4
2680     CALL Push_up
2690   CASE ELSE
2700     CALL Cancel
2710   END SELECT
2720 END IF
```

It could make quite a difference if you had gotten the various routines nested improperly. The INDENT command also helps you because when you enter your program into the computer, you don't have to worry about indenting your code. The indentation is done after you have entered your code when you type –

INDENT EXEC

You also have the option of specifying in which column you would like the program listing to begin, and how many spaces each section of code is indented.

INDENT 6,2 is the indentation which is performed by typing INDENT and pressing EXECUTE on a System 35. The value of 6 states that the body of the code begins in the sixth column. The value of 2 states that subsequent nested statements are indented two columns from each aligned edge of the nested constructs. You can change the spacing for either value from 0 to 72 columns with the INDENT statement.

INDENT 7,2 is the indentation which is performed by executing INDENT on a System 45.

XREF

XREF provides a cross reference of the following items –

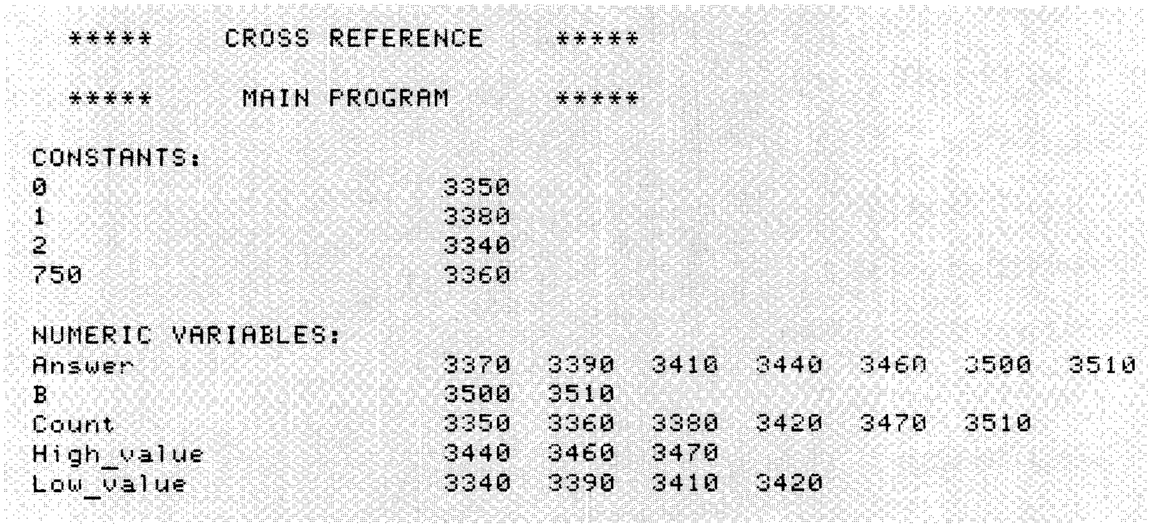
- Constants
- Line Numbers
- Labels
- Numeric Functions
- String Functions
- Subroutine Subprograms
- Assembler Symbols
- String Variables
- String Arrays
- Numeric Variables
- Numeric Arrays

for main programs and subprograms. This cross reference contains the names and the occurrences in the program, based upon the symbol table. The cross referenced listing is printed on the current printing device when the XREF statement is executed.

A program such as –

```
3290 ! This program uses the WHILE and IF THEN constructs .
3300 ! WHILE is used to control the duration of the loop.
3310 ! IF THEN is used to enable specific actions based upon
3320 ! the results of the equation.(RND)
3330 RANDOMIZE
3340 Low_value=2                                 ! Initialize the variables
3350 Count=0
3360 WHILE Count<750                             ! Entry for the loop
3370   Answer=RND*RND+RND*RND+RND                ! Number crunching
3380   Count=Count+1                             ! Update counter
3390   IF Answer<Low_value THEN                  ! Result is a new low
3400     BEEP                                    ! Operator prompt
3410     Low_value=Answer                        ! Update new low value
3420     PRINT "New Low of ";Low_value;" on Count ";Count
3430   ELSE
3440     IF Answer>High_value THEN               ! Result is a new high
3450       BEEP                                  ! Operator prompt
3460       High_value=Answer
3470       PRINT "New High of ";High_value;" on Count ";Count! Update new high
3480     END IF
3490   END IF
3500   B=B+Answer                                ! Update total
3510   DISP Count,Answer,B,B/Count               ! Operator prompt
3520 END WHILE                                   ! Exit for loop
3530 END
```

results in a cross referenced listing of –



```
   *****    CROSS REFERENCE    *****

   *****     MAIN PROGRAM      *****

 CONSTANTS:
 0                          3350
 1                          3380
 2                          3340
 750                        3360

 NUMERIC VARIABLES:
 Answer                     3370  3390  3410  3440  3460  3500  3510
 B                          3500  3510
 Count                      3350  3360  3380  3420  3470  3510
 High_value                 3440  3460  3470
 Low_value                  3340  3390  3410  3420
```

Chapter 5
Error Messages

There are four new error messages which are enabled by your Structured Programming ROM. They are Errors 345, 346, 347, and 348.

Error 345

The message for Error 345 is –

> Data type of expression in "CASE" does not match type of expression in "SELECT".

Cause

Error 345 is caused by using non-matching data types. For example,

```
3600 SELECT A$
3610 CASE 1
3620   PRINT "CASE 1"
3630 CASE 2
3640   PRINT "CASE 2"
3650 END SELECT
```

results in a run time ERROR 345 IN LINE 3610. The argument for the SELECT statement must be the same type as used in the CASE statement.

Cure

On a System 35, verify that:

- when the SELECT argument is a numeric expression, the CASE argument is a real constant.
- when the SELECT argument is a string expression, the CASE argument is a quote field.

On a System 45, verify that:

- when the SELECT argument is a numeric expression, the CASE argument is a numeric expression.
- when the SELECT argument is a string expression, the CASE argument is a string expression.

Error 346

The message for Error 346 is –

> INDENT parameter out of range. (0 to 72 accepted)

Cause

Error 346 is caused by having the value used for either of the two parameters being <0 or >72.

Cure

Change the incorrect value(s) to >=0 and <=72.

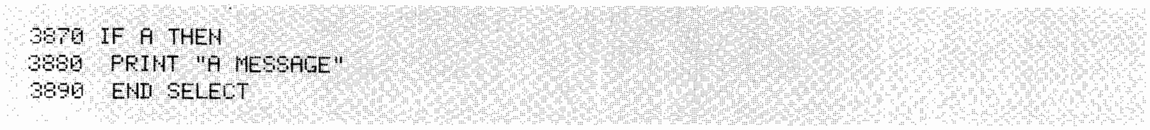
Error 347

The message for Error 347 is –

> Structured construct has improperly matched statements.

Cause

Error 347 is caused by improperly matched constructs. For example,



```
3870 IF A THEN
3880  PRINT "A MESSAGE"
3890  END SELECT
```

results in a run time ERROR 347 IN LINE 3870 (if A=0).

An Error 347 can also be caused by having a CASE statement after a CASE ELSE, or more than one ELSE or CASE ELSE in a construct.

Cure

Use the INDENT statement frequently when entering your program to verify that you are matching –

WHILE and END WHILE
REPEAT and UNTIL
LOOP and END LOOP
IF and END IF (only 1 ELSE is permitted)
SELECT and END SELECT (only 1 CASE ELSE is permitted) and that the CASE ELSE occurs only after all CASE statements for a given SELECT construct.

Error 348

The message for Error 348 is –

> Attempt to execute looping statement when no matching "WHILE", "REPEAT", or "LOOP" is active.

Cause

Error 348 is caused by improperly entering a looping construct. For example, the execution caused the program to encounter an "END WHILE", "UNTIL", "EXIT IF", or "END LOOP" without first executing the corresponding "WHILE", "REPEAT", or "LOOP" statement.

Cure

First use the INDENT command to determine if you forgot to use a "WHILE", "REPEAT", or "LOOP" when writing your code. If you did, the program would show the results.

Secondly, use an XREF to determine if there are any GOTO's or GOSUB's branching into your structured loops. If there are, you may have to redefine your program logic to avoid this.

Special Considerations

Missing ROM Error

If a program containing "RETURN" statements is STORE'd on a Series 9800 Desktop Computer which has a Structured Programming ROM installed, and is then LOAD'ed into a Series 9800 Desktop Computer which **does not have** the Structured Programming ROM, all "RETURN" statements will be flagged with a MISSING ROM error. You would then have to change the MISSING ROM lines to RETURN statements to run the program.

If a program is STORE'd on a Series 9800 Desktop Computer which does not have the Structured Programming ROM, and is then LOAD'ed into a Series 9800 Desktop Computer which has a Structured Programming ROM, no error message results. However, if any structured loops are added, the current RETURN lines **must be re-STORE'd** in order for the Structured Programming ROM to affect them.

It is suggested that programs which will be transported between Series 9800 Desktop Computers in this manner (with and without the Structured Programming ROM being installed) be SAVE'd rather than STORE'd.

Errors in Looping Construct Expressions

If an error occurs during the evaluation of the looping construct expression (WHILE, EXIT IF, or UNTIL), the loop is exited and deactivated before the error message is issued. The deactivation of the loop enables you to edit the looping construct program lines (in particular, its beginning and ending) to correct the error.

The looping construct containing the error is the only loop deactivated by this error, so you cannot edit the beginning or ending lines of any other active looping construct. All loops can be deactivated for editing by STOPping the program.

Chapter 6

Syntax

IF ... THEN ... ELSE ... END IF

```
 IF expression THEN
   statement
     •
     •
     •
   statement
[ ELSE
   statement
     •
     •
     •
   statement]
 END IF
```

(NB: no statement)

compare with

IF expression THEN statement

construct

The IF...THEN statement causes up to two choices of action to be implemented as the result of a conditional test.

The **expression** can be any numeric expression. When the expression is evaluated (at the IF...THEN statement), the following occurs –

If the expression is TRUE (Non-zero)

a) the program executes the section of code between the IF and the ELSE statement (if an ELSE statement exists).

b) the program executes the section of code between the IF and the END IF statement (if no ELSE statement exists).

If the expression is FALSE (=zero)

a) the program executes the section of code between the ELSE and the END IF statement (if an ELSE statement exists).

b) the program skips the section of code between the IF and the END IF statement (if no ELSE statement exists). Program execution resumes at the line following the END IF statement.

If the IF..END IF construct is entered without first executing the IF..THEN statement, the execution of the ELSE or END IF statements causes the program execution to continue at the first program line following the END IF statement.

If a statement or line identifier follows the THEN statement (on the same line), the line is executed as the mainframe's IF...THEN statement.

INDENT

`INDENT` [start column [, increment]]

The INDENT command causes uniform indentation of your program code. INDENT is only executable through live keyboard operation.

The **start column** specifies the value of the column where the first statement appears. (It defaults to 6 on a 9835, and to 7 on a 9845.) The **increment** specifies the number of columns that each subsequent construct is indented. (It defaults to 2.) The range for both parameters is 0< =parameter< =72.

Indentation occurs after the following statements –

FOR	IF	LOOP
REPEAT	SELECT	SUB
WHILE	multi-line DEF FN statements	

Indentation is reversed before the following statements, and then re-indented after the statement –

CASE	CASE ELSE	ELSE
FNEND	SUBEND	

Indentation is reversed before the following statements –

END IF	END LOOP	END SELECT
END WHILE	NEXT	UNTIL

The EXIT IF statement is aligned with its corresponding LOOP statement.

Statements beginning with ! (Remarks) and ISOURCE (Assembly Language programs) are not moved. Comments (!...) following a statement are normally not indented unless they would be overwritten as the result of an INDENT. REM statements (remarks) are indented as other statements are indented (see below).

All other statements outside of a structured construct are aligned with the starting column. Other statements occurring inside of a structured construct are indented from the structuring statements.

Improperly matched blocks may result in the indentation attempting to fall below the start column value. Indentation is bounded by the start column on the left and column 72 on the right. Indentation returns to the start column whenever the beginning of the program segment (main program, multi-line function, or subprogram) occurs.

Special Consideration

If a program line becomes too long to list as the result of an INDENT, the line number is followed by an asterisk (e.g. 4250*). When this occurs, use the INDENT statement with smaller arguments to bring the line length back to listable size. This must be done before SAVE'ing the program to avoid losing the line.

LOOP...EXIT IF...END LOOP

```
 LOOP
   statement
     ●
     ●
     ●
   statement
[ EXIT IF expression]
   statement
     ●
     ●
     ●
   statement
[ EXIT IF expression]
   statement
     ●
     ●
     ●
   statement
 END LOOP
```

LOOP...EXIT IF...END LOOP repeats the statements in a structured loop as long as the EXIT IF condition(s) is FALSE.

The **expression** can be any numeric expression. When the expression is evaluated (at the EXIT IF statement), the following occurs –

> If the expression is FALSE (=zero) when the EXIT IF statement is executed, the program continues looping.
>
> If the expression is TRUE (Non-zero) when the EXIT IF statement is executed, program execution continues at the line following the END LOOP statement.

Multiple EXIT IF statements are allowed inside of a LOOP...END LOOP. Having no EXIT IF statement results in an infinite loop.

If a LOOP...END LOOP is entered without first executing the LOOP statement, encountering the END LOOP causes an ERROR 348.

If an EXIT IF statement is executed and the expression is TRUE when the program is not in an active LOOP, an ERROR 348 is generated.

REPEAT ... UNTIL

```
REPEAT
  statement
   •
   •
   •
  statement
UNTIL expression
```

The REPEAT ... UNTIL loop repeats the statements in a structured loop until the expression in the UNTIL statement is true.

The **expression** can be any numeric expression. When the expression is evaluated (at the UNTIL statement), the following occurs –

> If the expression is FALSE (=zero) when the UNTIL statement is executed, the program continues looping.
>
> If the expression is TRUE (Non-zero) when the UNTIL statement is executed, program execution continues at the line following the UNTIL statement.

Because the conditional test is not performed until the execution of the UNTIL statement, a REPEAT ... UNTIL loop is always executed at least one time.

If a REPEAT...UNTIL loop is entered without executing the REPEAT statement, the execution of the UNTIL statement causes an ERROR 348.

SELECT ... CASE ... CASE ELSE ... END SELECT

```
SELECT expression
CASE item [, item... [, item]...]
   statement
      •
      •
      •
   statement
[ CASE item [, item... [, item]...]
   statement
      •
      •
      •
   statement
[ CASE ELSE
   statement
      •
      •
      •
   statement]
END SELECT
```

The SELECT ... CASE ... CASE ELSE ... END SELECT construct provides the execution of a choice of actions depending upon the result of a conditional test.

The **expression** can be any numeric or string expression. The **item** on a System 35 is a literal or a range, or the **item** on a System 45 is an expression or a range, where –

- a literal is either a signed (+, −) real constant (the sign is optional) or a quote field
- all literals in a CASE statement are the same type
- a range on a System 35 is literal TO literal or a relational operator (<, >) with a literal.
- a range on a System 45 is expression TO expression or a relational operator (<, >) with an expression.

When the SELECT statement is executed, the expression in the SELECT statement is tested to see if it matches the condition in a CASE statement. If a match is found (condition is valid), the program execution begins with the line after the first matching CASE statement. If there is no match and a CASE ELSE statement exists, the program execution begins with the line after the CASE ELSE statement. If there is no CASE ELSE statement, program execution resumes with the line following the END SELECT statement.

If the SELECT...END SELECT construct is entered without first executing the SELECT statement, the execution of any CASE, CASE ELSE or END SELECT statement causes program execution to resume at the program line following the END SELECT statement.

The TO enables a closed range to be selected. The TO range is a progressive sequence on the order of –

CASE n TO m

where n is <= m.

Any statements which exist between the SELECT and first CASE statement cannot be executed unless they are specifically branched to by means of a GOTO or GOSUB statement. That is poor programming technique, and is not recommended.

WHILE ... END WHILE

WHILE expression
statement
•
•
•
statement
END WHILE

The WHILE ... END WHILE statement repeats a structured loop when its expression is true.

The **expression** can be any numeric expression. When the expression is evaluated (at the WHILE statement), the following occurs –

> If the expression is TRUE (Non-zero) when the WHILE statement is executed, the program continues looping.
> If the expression is FALSE (=zero) when the WHILE statement is executed, program execution continues at the line following the END WHILE statement.

Because the conditional test is performed at the execution of the WHILE statement, a WHILE ... END WHILE loop may not always be executed.

If a WHILE...END WHILE loop is entered without executing the WHILE statement, the execution of the END WHILE statement causes an ERROR 348.

XREF

```
XREF
XREF options
XREF# select code [, HP-IB device address ] [; options]
```

The XREF statement prints a listing of the identifiers, and where they occur (line numbers) in your program.

The **options** may be either –

- a program environment (MAIN or SUBS)
- an object list (refer to table below)
- an object list within a program environment

Object List Table

Mnemonic	Object
AS	Assembler Symbols
CALL	Subroutine Subprograms
CN	Constants
LB	Labels
LN	Line Numbers
NA	Numeric Arrays
NF	Numeric Functions (DEF FN)
NV	Simple Numeric Variables
SA	String Arrays
SF	String Functions (DEF FN)
SV	Simple String Variables

The normal sequence of the options is –

CN, LN, LB, NF, SF, CALL, AS, SV, SA, NV, NA

with the listed items printed in alphabetical or numerical order on the specified printer.

The specified printer is either the current printer for your Series 9800 Desktop Computer, or the device specified by the select code and HP-IB device address. The **select code** is the switch setting of the interface. The **HP-IB device address** is the decimal value of the device address switches used on the backplane of HP-IB compatible printers.

9845 Select Code Range	Integers 0 thru 12 (16 for the CRT)
9835 Select Code Range	Integers 0 thru 14 (16 for the CRT)
HP-IB Device Addresses	Integers 0 thru 30

If both the select code AND an option list are used with the XREF command, the semicolon (;) **must** appear as a delimiter between the select code/device address and the option list.

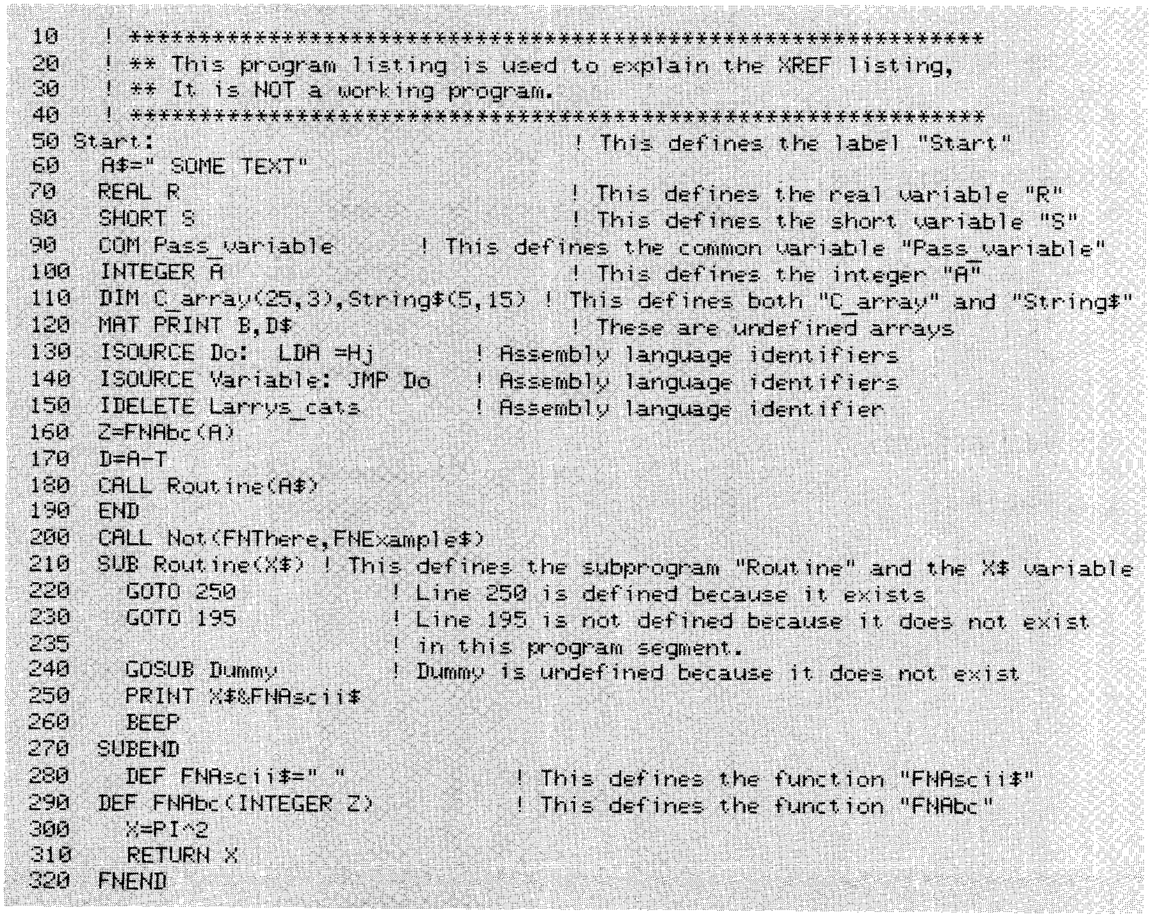
XREF is based upon symbol table entries, therefore –

XREF **will** –

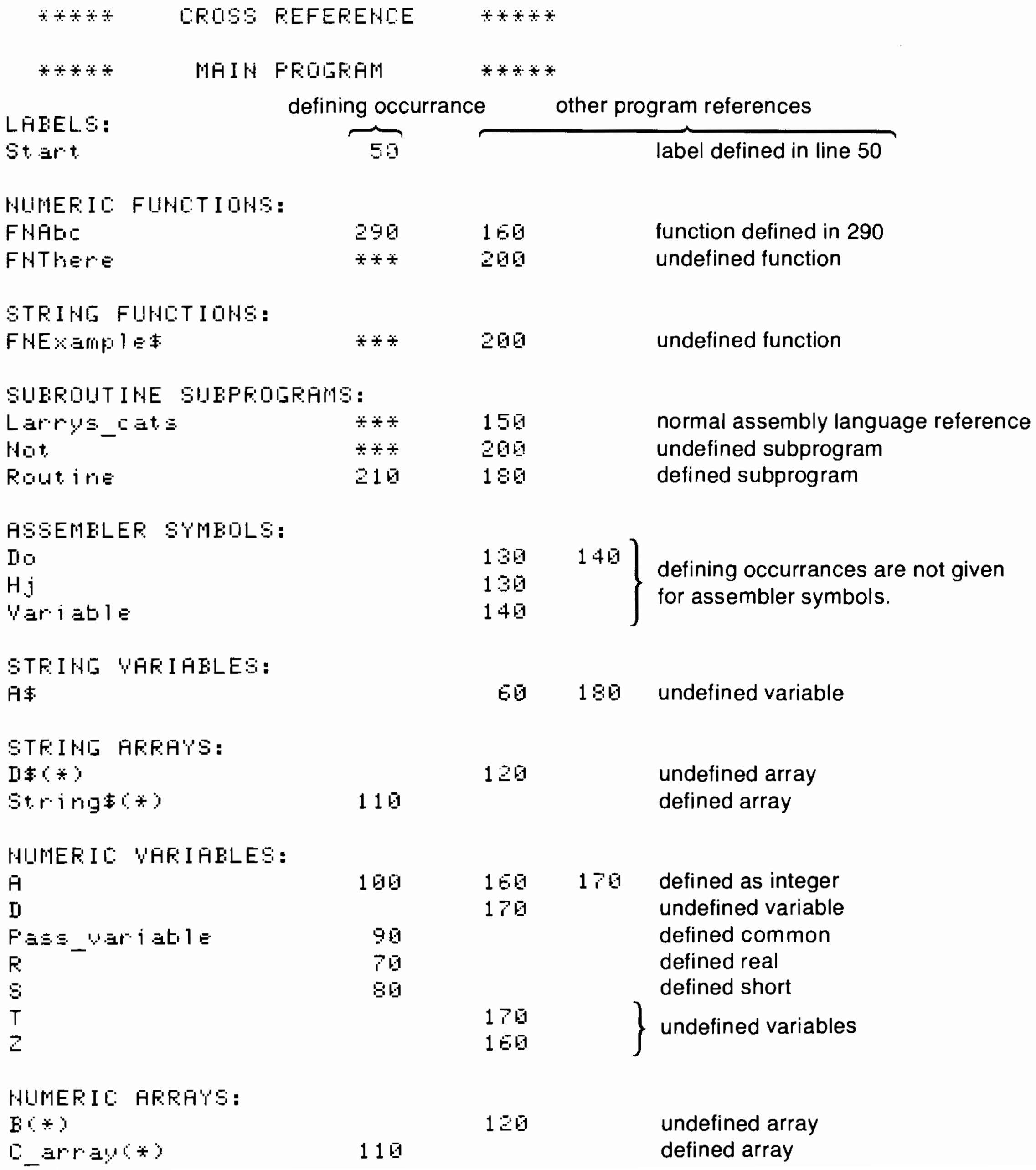
1. combine CALL and ICALL references to the same name, since they both point to the same symbol table entry.
2. list symbol table entries, even if the program does not reference them.
3. list multiple symbol table entries for the same line number. This listing is as consecutive separate items.
4. list symbol table entries for SECURE'd lines, but no line number references are given.

XREF **will not** –

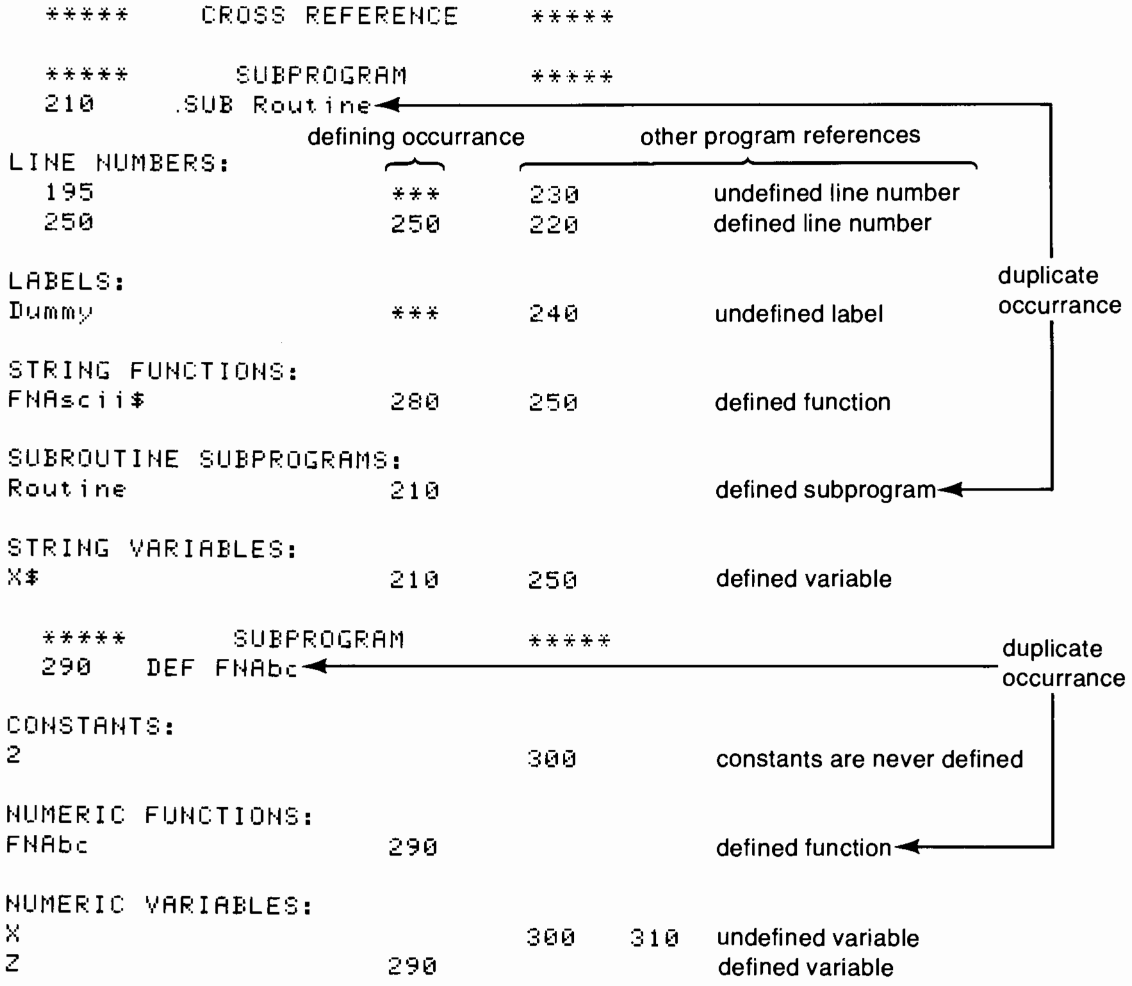
1. cross reference constants in the MAIN program for DIM, INTEGER, SHORT, REAL and COM statements
2. cross reference string literals (quote field).
3. cross reference file numbers in SUB and DEF FN statements.
4. list line numbers for SECURE'd program areas



```
10   ! *****************************************************************
20   ! ** This program listing is used to explain the XREF listing,
30   ! ** It is NOT a working program.
40   ! *****************************************************************
50 Start:                              ! This defines the label "Start"
60   A$=" SOME TEXT"
70   REAL R                            ! This defines the real variable "R"
80   SHORT S                           ! This defines the short variable "S"
90   COM Pass_variable      ! This defines the common variable "Pass_variable"
100  INTEGER A                         ! This defines the integer "A"
110  DIM C_array(25,3),String$(5,15) ! This defines both "C_array" and "String$"
120  MAT PRINT B,D$                    ! These are undefined arrays
130  ISOURCE Do:  LDA =Hj       ! Assembly language identifiers
140  ISOURCE Variable: JMP Do   ! Assembly language identifiers
150  IDELETE Larrys_cats        ! Assembly language identifier
160  Z=FNAbc(A)
170  D=A-T
180  CALL Routine(A$)
190  END
200  CALL Not(FNThere,FNExample$)
210  SUB Routine(X$) ! This defines the subprogram "Routine" and the X$ variable
220    GOTO 250           ! Line 250 is defined because it exists
230    GOTO 195           ! Line 195 is not defined because it does not exist
235                       ! in this program segment.
240    GOSUB Dummy        ! Dummy is undefined because it does not exist
250    PRINT X$&FNAscii$
260    BEEP
270  SUBEND
280    DEF FNAscii$=" "             ! This defines the function "FNAscii$"
290  DEF FNAbc(INTEGER Z)           ! This defines the function "FNAbc"
300    X=PI^2
310    RETURN X
320  FNEND
```



```
    *****     CROSS REFERENCE     *****

    *****      MAIN PROGRAM       *****
                              defining occurrance     other program references
LABELS:
Start                             50                           label defined in line 50

NUMERIC FUNCTIONS:
FNAbc                            290       160                 function defined in 290
FNThere                          ***       200                 undefined function

STRING FUNCTIONS:
FNExample$                       ***       200                 undefined function

SUBROUTINE SUBPROGRAMS:
Larrys_cats                      ***       150                 normal assembly language reference
Not                              ***       200                 undefined subprogram
Routine                          210       180                 defined subprogram

ASSEMBLER SYMBOLS:
Do                                         130     140  }
Hj                                         130          }  defining occurrances are not given
Variable                                   140          }  for assembler symbols.

STRING VARIABLES:
A$                                          60     180     undefined variable

STRING ARRAYS:
D$(*)                                      120             undefined array
String$(*)                       110                       defined array

NUMERIC VARIABLES:
A                                100       160     170     defined as integer
D                                          170             undefined variable
Pass_variable                     90                       defined common
R                                 70                       defined real
S                                 80                       defined short
T                                          170          }
Z                                          160          }  undefined variables

NUMERIC ARRAYS:
B(*)                                       120             undefined array
C_array(*)                       110                       defined array
```



```
*****     CROSS REFERENCE     *****

*****       SUBPROGRAM        *****
210     .SUB Routine

LINE NUMBERS:
  195                       ***     230
  250                       250     220

LABELS:
Dummy                       ***     240

STRING FUNCTIONS:
FNAscii$                    280     250

SUBROUTINE SUBPROGRAMS:
Routine                     210

STRING VARIABLES:
X$                          210     250

*****       SUBPROGRAM        *****
290    DEF FNAbc

CONSTANTS:
2                                   300

NUMERIC FUNCTIONS:
FNAbc                       290

NUMERIC VARIABLES:
X                                   300     310
Z                           290
```

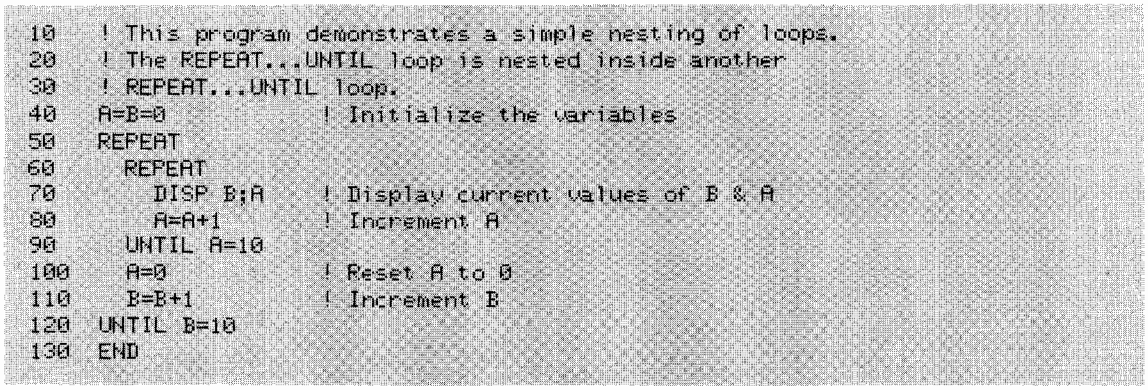
Special Consideration

If you are using an assembly language DAT statement which contains characters with video highlights (inverse video, blinking, or underline) and you execute an XREF, there is a possibility that other identifiers which occur in the symbol table may be cross-referenced to the DAT statement. If this happens, disregard any cross-references to the DAT statement for identifiers not listed in the DAT statement.

Chapter 7
Examples

The examples used in this chapter show typical applications of your Structured Programming ROM. In providing "real-world" examples, certain ROMs are required for all examples to run properly. When option ROMs are required, the program listing is annotated to reflect this.

When reviewing these examples, keep in mind that there are n ways to solve a problem, and these examples do not represent the only way that the problem can be solved.



```
10   ! This program demonstrates a simple nesting of loops.
20   ! The REPEAT...UNTIL loop is nested inside another
30   ! REPEAT...UNTIL loop.
40   A=B=0           ! Initialize the variables
50   REPEAT
60     REPEAT
70       DISP B;A    ! Display current values of B & A
80       A=A+1       ! Increment A
90     UNTIL A=10
100    A=0           ! Reset A to 0
110    B=B+1         ! Increment B
120  UNTIL B=10
130  END
```

```
10 ! ** This program converts integers from number systems
20 ! ** with a base less than 10 to base 10. The WHILE
30 ! ** construct is used for polynomial expansion. (SBEX1)
40 Overhead:  !
50   OPTION BASE 1                        ! Miscellaneous overhead
60   INTEGER Number,Base,Array(9)
70   Array_pointer=1                      ! Initialize the pointer
80 Input:  !
90   INPUT "NUMBER",Number
100  Old_number=Number                    ! Initialize the variable
110  INPUT "BASE",Base
120 Fill_array: !
130  WHILE Number                         ! Entry for the loop
140    Array(Array_pointer)=Number MOD 10 ! Strip off the LSD
150    Number=Number DIV 10               ! Drop last digit and truncate
160    Array_pointer=Array_pointer+1      ! Update array pointer
170  END WHILE                            !
180 Conversion: !
190  Array_pointer=Array_pointer-1        ! Initialize the pointer
200  New_number=0                         ! Initialize the variable
210  WHILE Array_pointer                  ! Nested conversion
220    New_number=New_number*Base+Array(Array_pointer)
230    Array_pointer=Array_pointer-1      ! Update array pointer
240  END WHILE                            !
250 Output: !
260  PRINT Old_number;"Base";Base;"=";New_number;"Base 10"
270  END
```

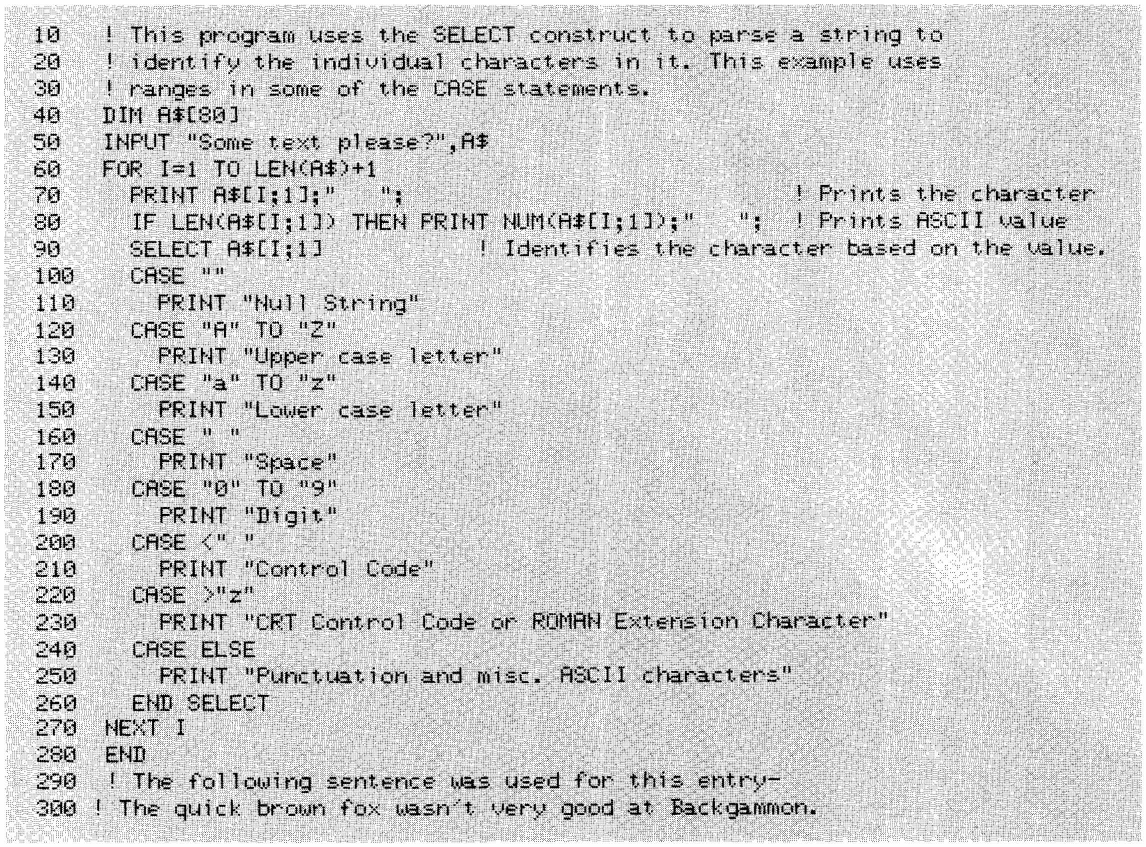
```
15 Base 8 = 13 Base 10
789 Base 10 = 789 Base 10
10110 Base 2 = 22 Base 10
7777 Base 8 = 4095 Base 10
```

```
10   ! ** This example shows the SELECT construct being used
20   ! ** with closed ranges. (SELRNG)
30   INPUT "Give me a number between -32 768 and 32 767",Variable
40   SELECT Variable                     ! Use the variable for testing
50   CASE -32768,32767                   ! Test for limits
60     PRINT Variable;"is at the limit"
70   CASE 0                              ! Test for 0
80     PRINT "It is zero"
90   CASE -32768 TO 0                    ! Test for negative value in range
100    PRINT Variable;"is negative"
110  CASE 0 TO 32767                     ! Test for positive value in range
120    PRINT Variable;"is positive"
130  CASE ELSE                           ! Variable is out of range
140    PRINT Variable;"is out of range"
150  END SELECT
160  END
```

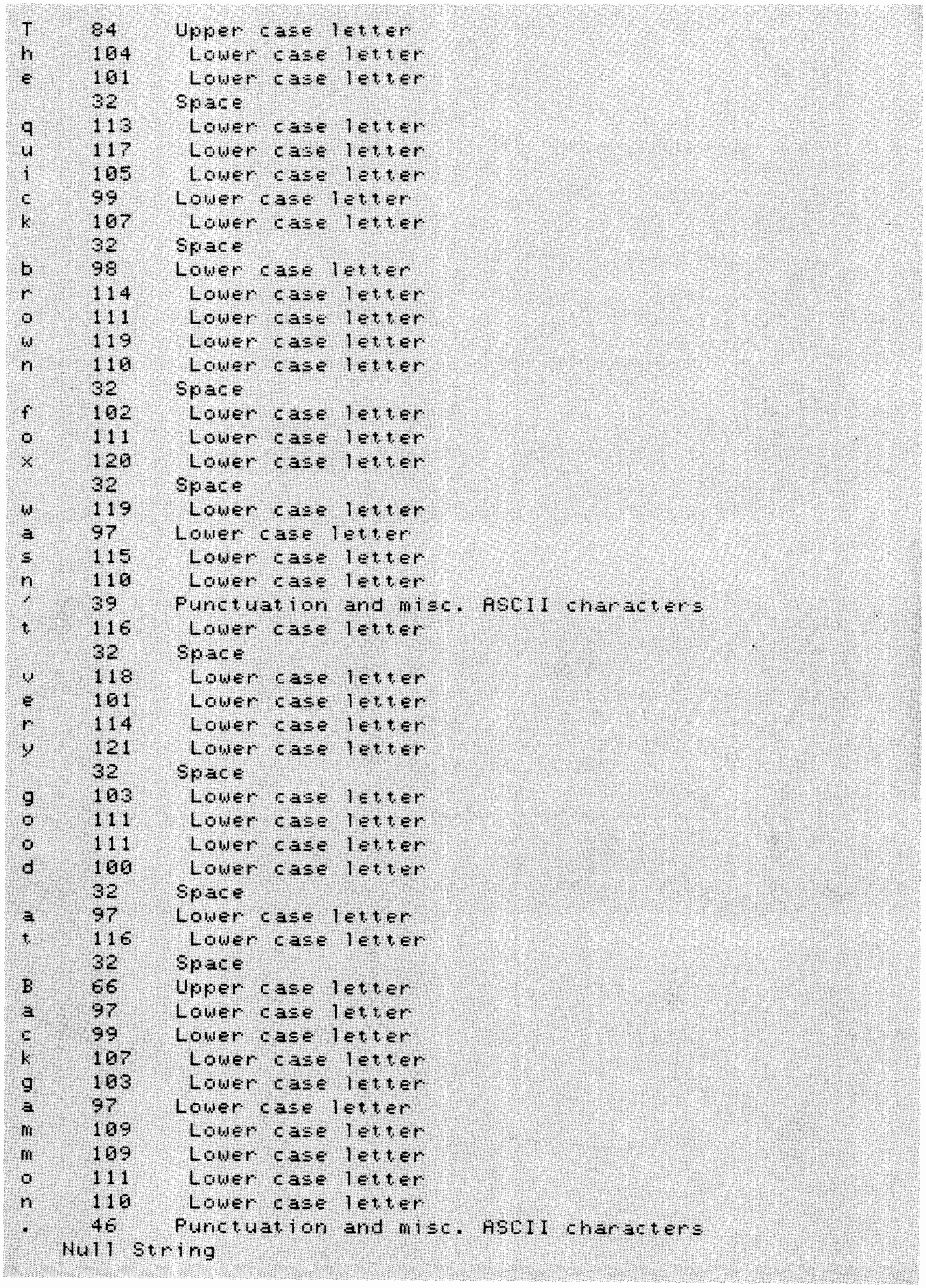
```
 2442 is positive
-73 is negative
It is zero
 32767 is at the limit
 32768 is out of range
-32768 is at the limit
```

```
10    ! ** This example shows the SELECT construct being used
20    ! ** with open ranges. The result is similar to the
30    ! ** previous example. (SELRN2)
40    INPUT "Give me a number between -32 768 and 32 767",Variable
50    SELECT Variable                               ! Use the variable for testing
60    CASE <-32768                                  ! Test for out of range
70       PRINT Variable;"is out of range"
80    CASE -32768,32767                             ! Test for limits
90      PRINT Variable;"is at the limit"
100   CASE 0                                        ! Test for 0
110     PRINT "It is zero"
120   CASE >-32768                                  ! Test for value in range
130     SELECT Variable                             ! Use the variable for testing
140     CASE >32767                                 ! Test for out of range
150       PRINT Variable;"is out of range"
160     CASE <0                                     ! Test for negative value
170       PRINT Variable;"is negative"
180     CASE >0                                     ! Test for positive value
190       PRINT Variable;"is positive"
200     END SELECT
210   END SELECT
220   END
```

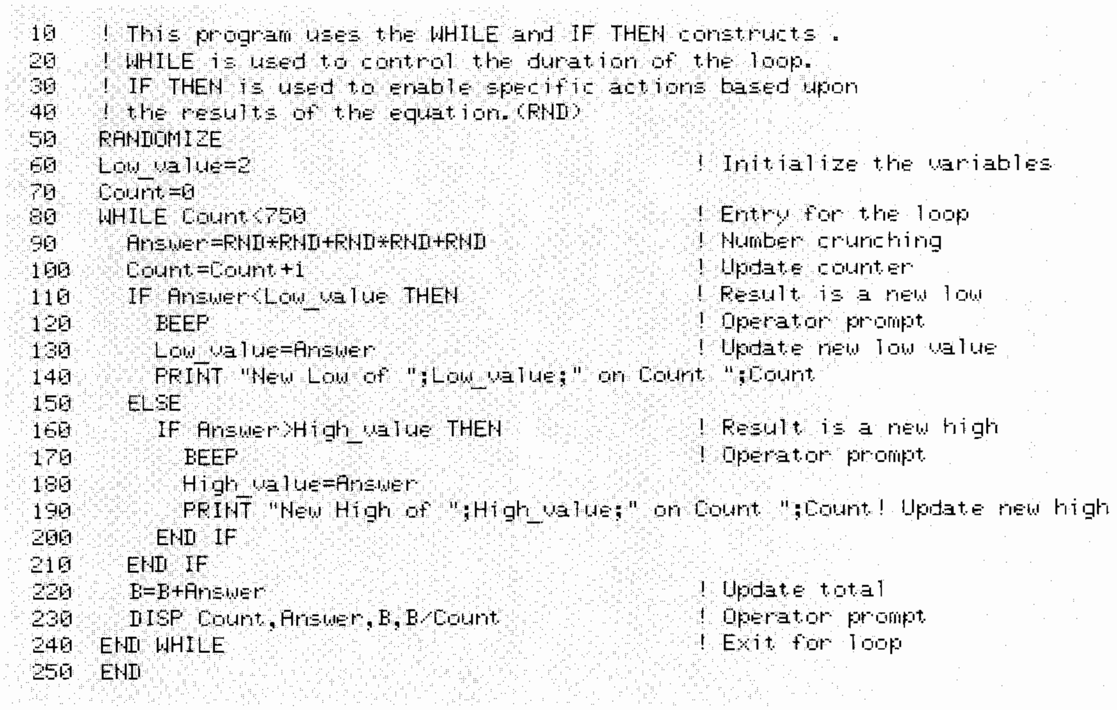
```
 2442 is positive
-73 is negative
It is zero
 32767 is at the limit
 32768 is out of range
-32768 is at the limit
```



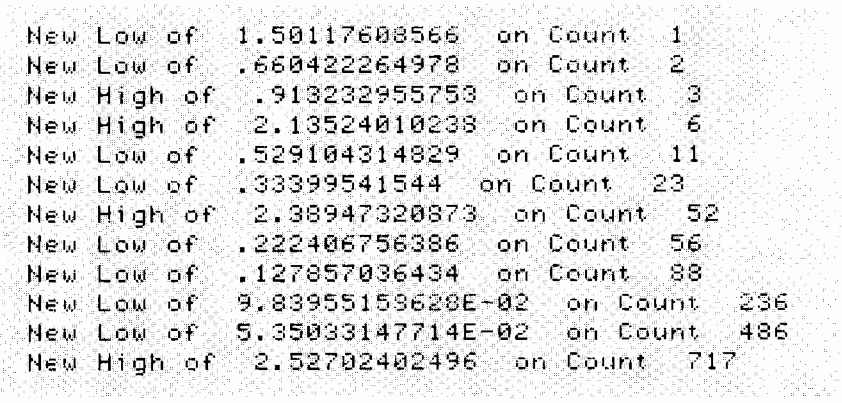
```
10   ! This program uses the SELECT construct to parse a string to
20   ! identify the individual characters in it. This example uses
30   ! ranges in some of the CASE statements.
40   DIM A$[80]
50   INPUT "Some text please?",A$
60   FOR I=1 TO LEN(A$)+1
70     PRINT A$[I;1];"   ";                            ! Prints the character
80     IF LEN(A$[I;1]) THEN PRINT NUM(A$[I;1]);"   ";  ! Prints ASCII value
90     SELECT A$[I;1]           ! Identifies the character based on the value.
100    CASE ""
110      PRINT "Null String"
120    CASE "A" TO "Z"
130      PRINT "Upper case letter"
140    CASE "a" TO "z"
150      PRINT "Lower case letter"
160    CASE " "
170      PRINT "Space"
180    CASE "0" TO "9"
190      PRINT "Digit"
200    CASE <" "
210      PRINT "Control Code"
220    CASE >"z"
230      PRINT "CRT Control Code or ROMAN Extension Character"
240    CASE ELSE
250      PRINT "Punctuation and misc. ASCII characters"
260    END SELECT
270  NEXT I
280  END
290  ! The following sentence was used for this entry-
300 ! The quick brown fox wasn´t very good at Backgammon.
```



```
T    84    Upper case letter
h    104    Lower case letter
e    101    Lower case letter
     32    Space
q    113    Lower case letter
u    117    Lower case letter
i    105    Lower case letter
c    99    Lower case letter
k    107    Lower case letter
     32    Space
b    98    Lower case letter
r    114    Lower case letter
o    111    Lower case letter
w    119    Lower case letter
n    110    Lower case letter
     32    Space
f    102    Lower case letter
o    111    Lower case letter
x    120    Lower case letter
     32    Space
w    119    Lower case letter
a    97    Lower case letter
s    115    Lower case letter
n    110    Lower case letter
'    39    Punctuation and misc. ASCII characters
t    116    Lower case letter
     32    Space
v    118    Lower case letter
e    101    Lower case letter
r    114    Lower case letter
y    121    Lower case letter
     32    Space
g    103    Lower case letter
o    111    Lower case letter
o    111    Lower case letter
d    100    Lower case letter
     32    Space
a    97    Lower case letter
t    116    Lower case letter
     32    Space
B    66    Upper case letter
a    97    Lower case letter
c    99    Lower case letter
k    107    Lower case letter
g    103    Lower case letter
a    97    Lower case letter
m    109    Lower case letter
m    109    Lower case letter
o    111    Lower case letter
n    110    Lower case letter
.    46    Punctuation and misc. ASCII characters
   Null String
```



```
10   ! This program uses the WHILE and IF THEN constructs .
20   ! WHILE is used to control the duration of the loop.
30   ! IF THEN is used to enable specific actions based upon
40   ! the results of the equation.(RND)
50   RANDOMIZE
60   Low_value=2                                 ! Initialize the variables
70   Count=0
80   WHILE Count<750                             ! Entry for the loop
90     Answer=RND*RND+RND*RND+RND                ! Number crunching
100    Count=Count+1                             ! Update counter
110    IF Answer<Low_value THEN                  ! Result is a new low
120      BEEP                                    ! Operator prompt
130      Low_value=Answer                        ! Update new low value
140      PRINT "New Low of ";Low_value;" on Count ";Count
150    ELSE
160      IF Answer>High_value THEN               ! Result is a new high
170        BEEP                                  ! Operator prompt
180        High_value=Answer
190        PRINT "New High of ";High_value;" on Count ";Count! Update new high
200      END IF
210    END IF
220    B=B+Answer                                ! Update total
230    DISP Count,Answer,B,B/Count               ! Operator prompt
240  END WHILE                                   ! Exit for loop
250  END
```

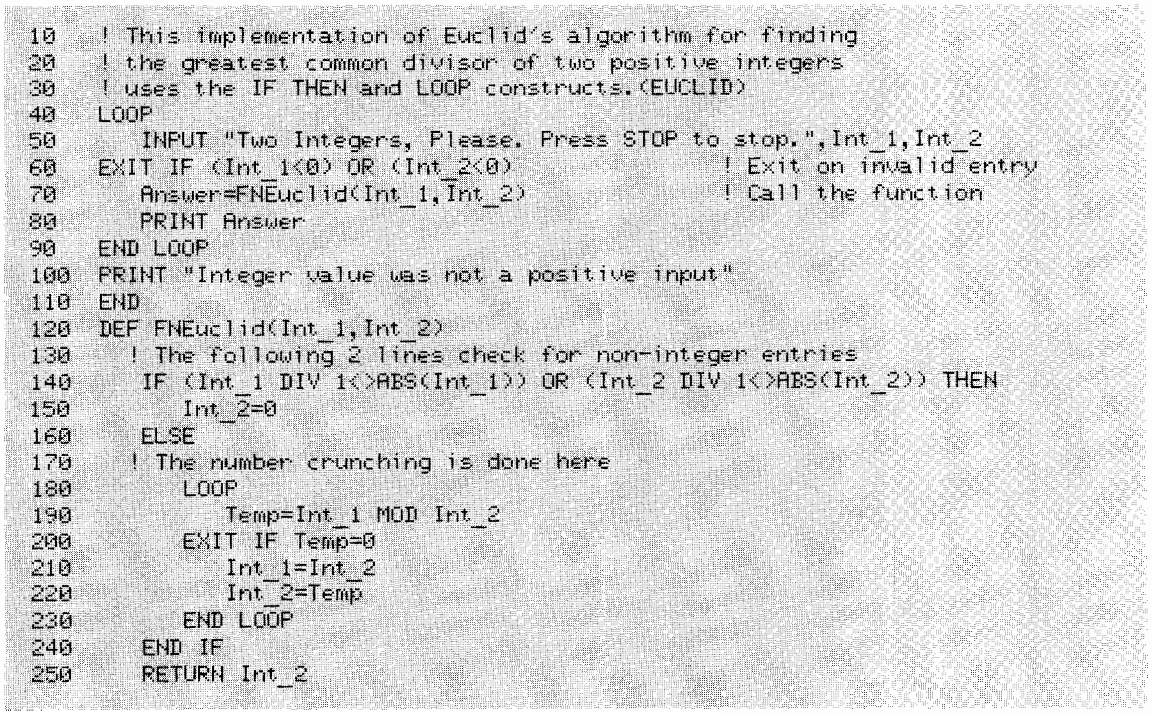


```
New Low of  1.50117608566  on Count  1
New Low of  .660422264978  on Count  2
New High of  .913232955753  on Count  3
New High of  2.13524010238  on Count  6
New Low of  .529104314829  on Count  11
New Low of  .33399541544  on Count  23
New High of  2.38947320873  on Count  52
New Low of  .222406756386  on Count  56
New Low of  .127857036434  on Count  88
New Low of  9.83955159628E-02  on Count  236
New Low of  5.35033147714E-02  on Count  486
New High of  2.52702402496  on Count  717
```

```
10   ! ** This program uses the LOOP and REPEAT constructs
20   ! ** to generate a "magic square" for odd numbered matrices.
30   ! ** In a magic square, the sum of any row equals the sum
40   ! ** of any of its columns. This algorithm works for any valid
50   ! ** positive integer, but it is limited to 15 so the entire
60   ! ** square is visible on the CRT at one time. (MAGIC)
70   INPUT "An odd integer, please.(Range of 1 to 15)",S
80   IF S MOD 2 AND (S<16) THEN CALL Magic(S-1)
90   END
100  SUB Magic(S)
110    DIM Magic(S,S)              ! Dynamic allocation of array
120    X=S/2                       ! Initialize variable
130    Y=S                         ! Initialize variable
140    MAT Magic=(0)               ! Initialize array
150    Counter=1                   ! Initialize variable
160    LOOP                        ! This section nests a REPEAT loop inside of
170      REPEAT                    ! a LOOP construct. As shown, an EXIT IF for
180        Magic(X,Y)=Counter      ! the LOOP occurs within the REPEAT construct.
190        Counter=Counter+1       ! The LOOP and REPEAT are used to generate the
200    EXIT IF Counter>(S+1)^2     ! values used in the magic square.
210        X=(X-1) MOD (S+1)
220        Y=(Y+1) MOD (S+1)
230      UNTIL Magic(X,Y)
240      X=(X+1) MOD (S+1)
250      Y=(Y-2) MOD (S+1)
260    END LOOP
270    FOR Y=S TO 0 STEP -1       ! The nested FOR...NEXT loops are used to print
280      FOR X=0 TO S             ! the elements for the magic square.
290        PRINT RPT$(" ",2-INT(LGT(Magic(X,Y))));Magic(X,Y);
300      NEXT X
310      PRINT
320    NEXT Y
330  SUBEND
```

Magic Square for 9

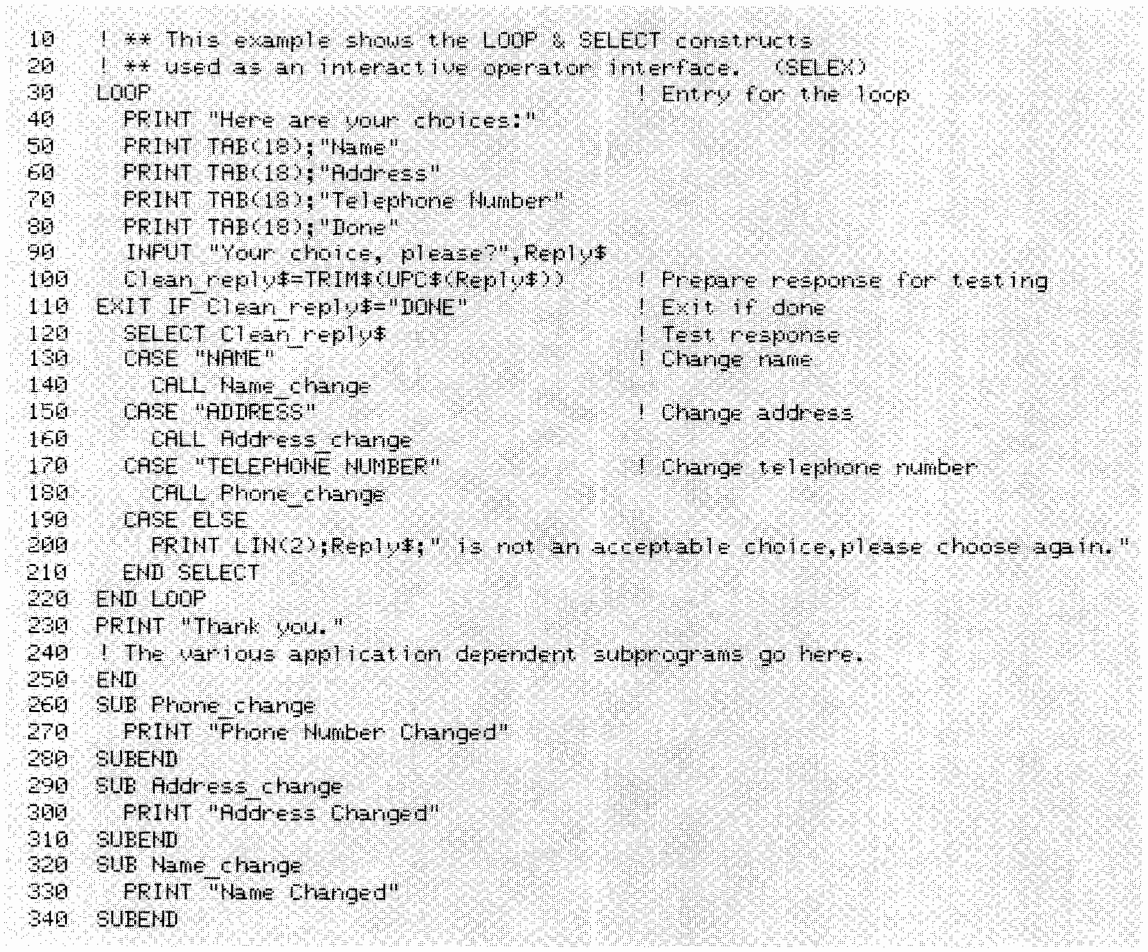
```
45   34   23   12    1   80   69   58   47
46   44   33   22   11    9   79   68   57
56   54   43   32   21   10    8   78   67
66   55   53   42   31   20   18    7   77
76   65   63   52   41   30   19   17    6
 5   75   64   62   51   40   29   27   16
15    4   74   72   61   50   39   28   26
25   14    3   73   71   60   49   38   36
35   24   13    2   81   70   59   48   37
```



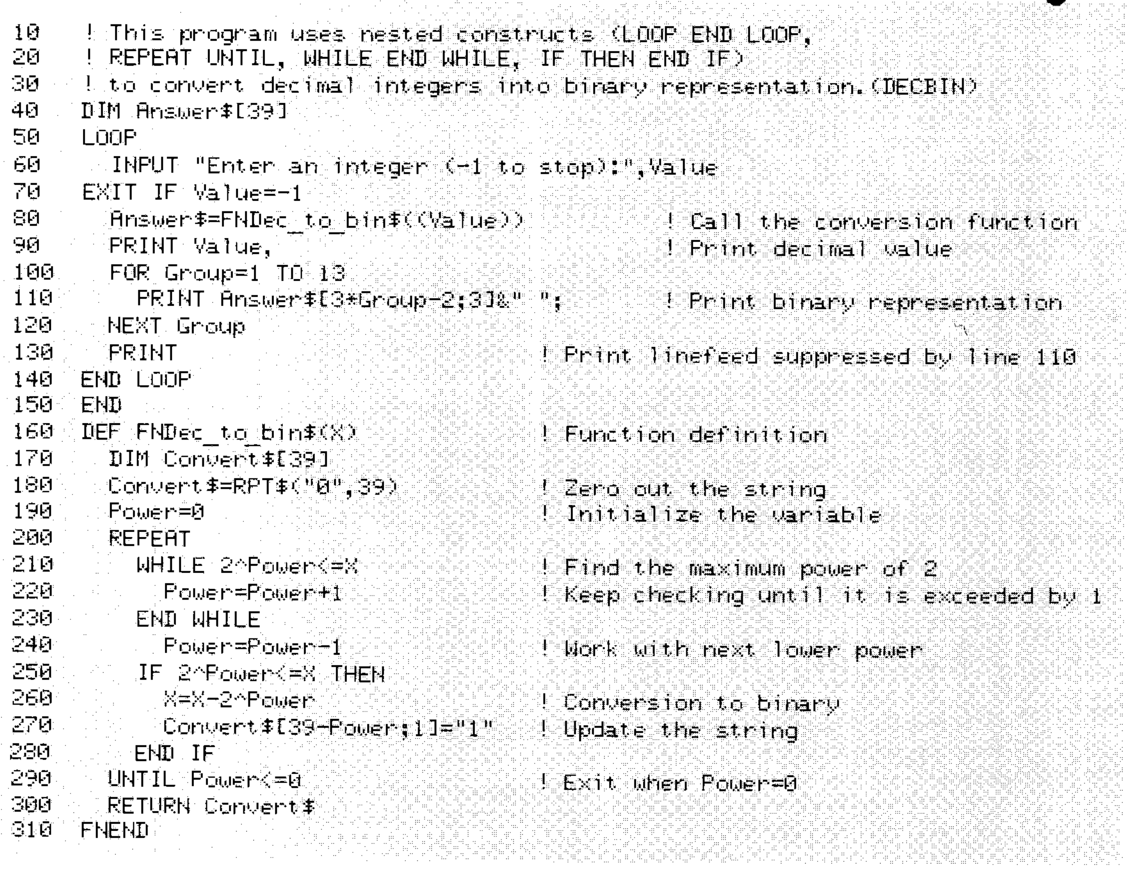
```
10   ! This implementation of Euclid's algorithm for finding
20   ! the greatest common divisor of two positive integers
30   ! uses the IF THEN and LOOP constructs.(EUCLID)
40   LOOP
50      INPUT "Two Integers, Please. Press STOP to stop.",Int_1,Int_2
60   EXIT IF (Int_1<0) OR (Int_2<0)               ! Exit on invalid entry
70      Answer=FNEuclid(Int_1,Int_2)              ! Call the function
80      PRINT Answer
90   END LOOP
100  PRINT "Integer value was not a positive input"
110  END
120  DEF FNEuclid(Int_1,Int_2)
130    ! The following 2 lines check for non-integer entries
140     IF (Int_1 DIV 1<>ABS(Int_1)) OR (Int_2 DIV 1<>ABS(Int_2)) THEN
150        Int_2=0
160     ELSE
170    ! The number crunching is done here
180        LOOP
190           Temp=Int_1 MOD Int_2
200        EXIT IF Temp=0
210           Int_1=Int_2
220           Int_2=Temp
230        END LOOP
240     END IF
250     RETURN Int_2
```

```
10   ! *******************************************
20   !      THIS EXAMPLE REQUIRES AN I/O ROM.
30   ! *******************************************
40   ! This example uses WHILE and SELECT for serial polling.
50   STATUS 0;Interfacestatus                 ! Read status of interface 0
60   PRINT PAGE;"Interface Status Value=";OCTAL(Interfacestatus);"Octal"
70   Divisor=2                                ! Initialize the variable
80   WHILE Interfacestatus>0                  ! Entry for the loop
90     Testcondition=Interfacestatus MOD Divisor! Check for bit being set
100    SELECT Testcondition                   ! Bit result
110    CASE 1                                 ! Most important service routine
120    PRINT "Servicing Device for bit #0"
130    CASE 2
140    PRINT "Servicing Device for bit #1"
150    CASE 4
160    PRINT "Servicing Device for bit #2"
170    CASE 8
180    PRINT "Servicing Device for bit #3"
190    CASE 16
200    PRINT "Servicing Device for bit #4"
210    CASE 32
220    PRINT "Servicing Device for bit #5"
230    CASE 64
240    PRINT "Servicing Device for bit #6"
250    CASE 128                               ! Least important service routine
260    PRINT "Servicing Device for bit #7"
270    END SELECT
280    Divisor=Divisor*2                      ! Update divisor for next bit
290    Interfacestatus=Interfacestatus-Testcondition    ! Update result
300  END WHILE
310  END
```

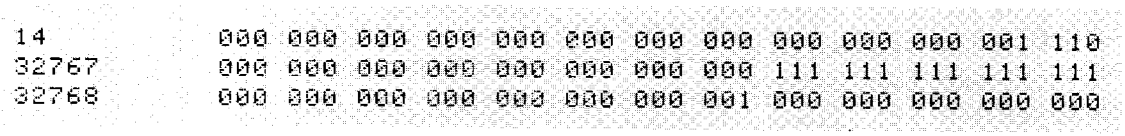
```
Interface Status Value= 13 Octal
Servicing Device for bit #0
Servicing Device for bit #1
Servicing Device for bit #3
```



```
10   ! ** This example shows the LOOP & SELECT constructs
20   ! ** used as an interactive operator interface.  (SELEX)
30   LOOP                                    ! Entry for the loop
40     PRINT "Here are your choices:"
50     PRINT TAB(18);"Name"
60     PRINT TAB(18);"Address"
70     PRINT TAB(18);"Telephone Number"
80     PRINT TAB(18);"Done"
90     INPUT "Your choice, please?",Reply$
100    Clean_reply$=TRIM$(UPC$(Reply$))      ! Prepare response for testing
110  EXIT IF Clean_reply$="DONE"             ! Exit if done
120    SELECT Clean_reply$                   ! Test response
130    CASE "NAME"                           ! Change name
140      CALL Name_change
150    CASE "ADDRESS"                        ! Change address
160      CALL Address_change
170    CASE "TELEPHONE NUMBER"               ! Change telephone number
180      CALL Phone_change
190    CASE ELSE
200      PRINT LIN(2);Reply$;" is not an acceptable choice,please choose again."
210    END SELECT
220  END LOOP
230  PRINT "Thank you."
240  ! The various application dependent subprograms go here.
250  END
260  SUB Phone_change
270    PRINT "Phone Number Changed"
280  SUBEND
290  SUB Address_change
300    PRINT "Address Changed"
310  SUBEND
320  SUB Name_change
330    PRINT "Name Changed"
340  SUBEND
```

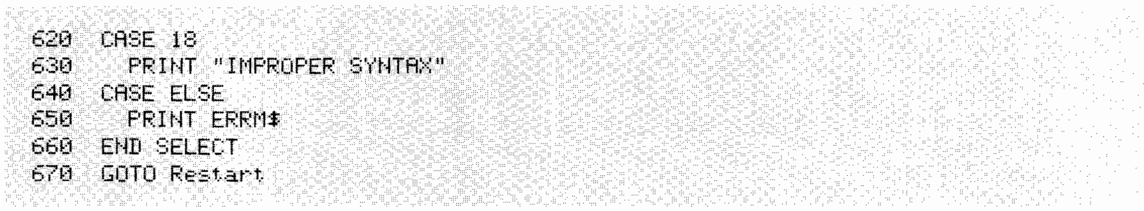


```
10   ! This program uses nested constructs (LOOP END LOOP,
20   ! REPEAT UNTIL, WHILE END WHILE, IF THEN END IF)
30   ! to convert decimal integers into binary representation.(DECBIN)
40   DIM Answer$[39]
50   LOOP
60     INPUT "Enter an integer (-1 to stop):",Value
70   EXIT IF Value=-1
80     Answer$=FNDec_to_bin$((Value))          ! Call the conversion function
90     PRINT Value,                            ! Print decimal value
100    FOR Group=1 TO 13
110      PRINT Answer$[3*Group-2;3]&" ";       ! Print binary representation
120    NEXT Group
130    PRINT                           ! Print linefeed suppressed by line 110
140  END LOOP
150  END
160  DEF FNDec_to_bin$(X)              ! Function definition
170    DIM Convert$[39]
180    Convert$=RPT$("0",39)           ! Zero out the string
190    Power=0                         ! Initialize the variable
200    REPEAT
210      WHILE 2^Power<=X              ! Find the maximum power of 2
220        Power=Power+1               ! Keep checking until it is exceeded by 1
230      END WHILE
240        Power=Power-1               ! Work with next lower power
250      IF 2^Power<=X THEN
260        X=X-2^Power                 ! Conversion to binary
270        Convert$[39-Power;1]="1"    ! Update the string
280      END IF
290    UNTIL Power<=0                  ! Exit when Power=0
300    RETURN Convert$
310  FNEND
```

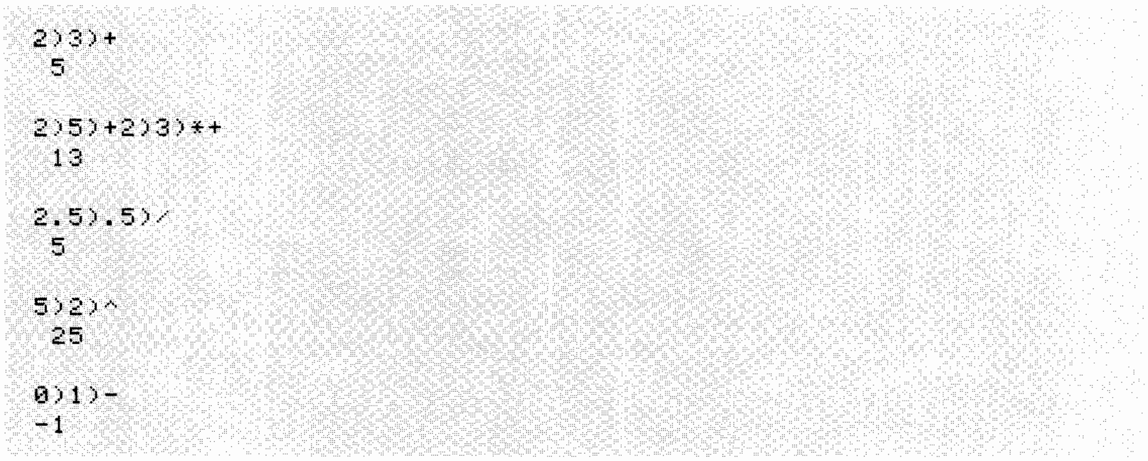


```
14              000 000 000 000 000 000 000 000 000 000 000 001 110
32767           000 000 000 000 000 000 000 000 111 111 111 111 111
32768           000 000 000 000 000 000 000 001 000 000 000 000 000
```

```
10   ! This program is a Reverse Polish Notation calculator.
20   ! Enter a string of numbers and operators. Numbers must
30   ! be followed by ). Operators are binary and postfix.
40   ! Negative numbers are realized by subtracting from 0).
50   ! Allowed operators are +, -, *, /, ^.
60   ! Samples: 2)3)+, 2)5)+2)3)*+, 2.5).5)/, 5)2)^, 0)1)-
70   DIM Stack(20),String$[160]
80   ON ERROR GOTO Err
90 Restart:  !
100  LOOP
110    Pointer=-1
120    INPUT String$
130    WHILE LEN(String$)
140 ! This section uses the SELECT...CASE construct for determining
150 ! which operations are performed.
160      SELECT String$[1;1]
170      CASE "+"
180        Stack(Pointer-1)=Stack(Pointer)+Stack(Pointer-1)
190        Pointer=Pointer-1
200      CASE "-"
210        Stack(Pointer-1)=Stack(Pointer-1)-Stack(Pointer)
220        Pointer=Pointer-1
230      CASE "*"
240        Stack(Pointer-1)=Stack(Pointer)*Stack(Pointer-1)
250        Pointer=Pointer-1
260      CASE "/"
270        Stack(Pointer-1)=Stack(Pointer-1)/Stack(Pointer)
280        Pointer=Pointer-1
290      CASE "^"
300        Stack(Pointer-1)=Stack(Pointer-1)^Stack(Pointer)
310        Pointer=Pointer-1
320      CASE "0" TO "9","."
330        Pointer=Pointer+1
340        Stack(Pointer)=VAL(String$)
350        REPEAT
360          String$=String$[2]
370        UNTIL String$[1;1]=")"
380      END SELECT
390      String$=String$[2]
400    END WHILE
410    IF NOT Pointer THEN
420      PRINT Stack(0)
430    ELSE
440      PRINT "SYNTAX ERROR"
450    END IF
460  END LOOP
470 Err:! This section uses the SELECT...CASE construct for determining
480     ! which error messages are printed.
490  SELECT ERRN
500  CASE 27
510    PRINT "NEGATIVE BASE TO NON-INTEGER POWER"
520  CASE 26
530    PRINT "ZERO TO NEGATIVE POWER"
540  CASE 23
550    PRINT "INTERMEDIATE RESULT OVERFLOW"
560  CASE 22
570    PRINT "REAL PRECISION OVERFLOW"
580  CASE 31
590    PRINT "DIVISION BY ZERO"
600  CASE 17
610    PRINT "STACK UNDERFLOW OR OVERFLOW"
```



```
620  CASE 18
630    PRINT "IMPROPER SYNTAX"
640  CASE ELSE
650    PRINT ERRM$
660  END SELECT
670  GOTO Restart
```



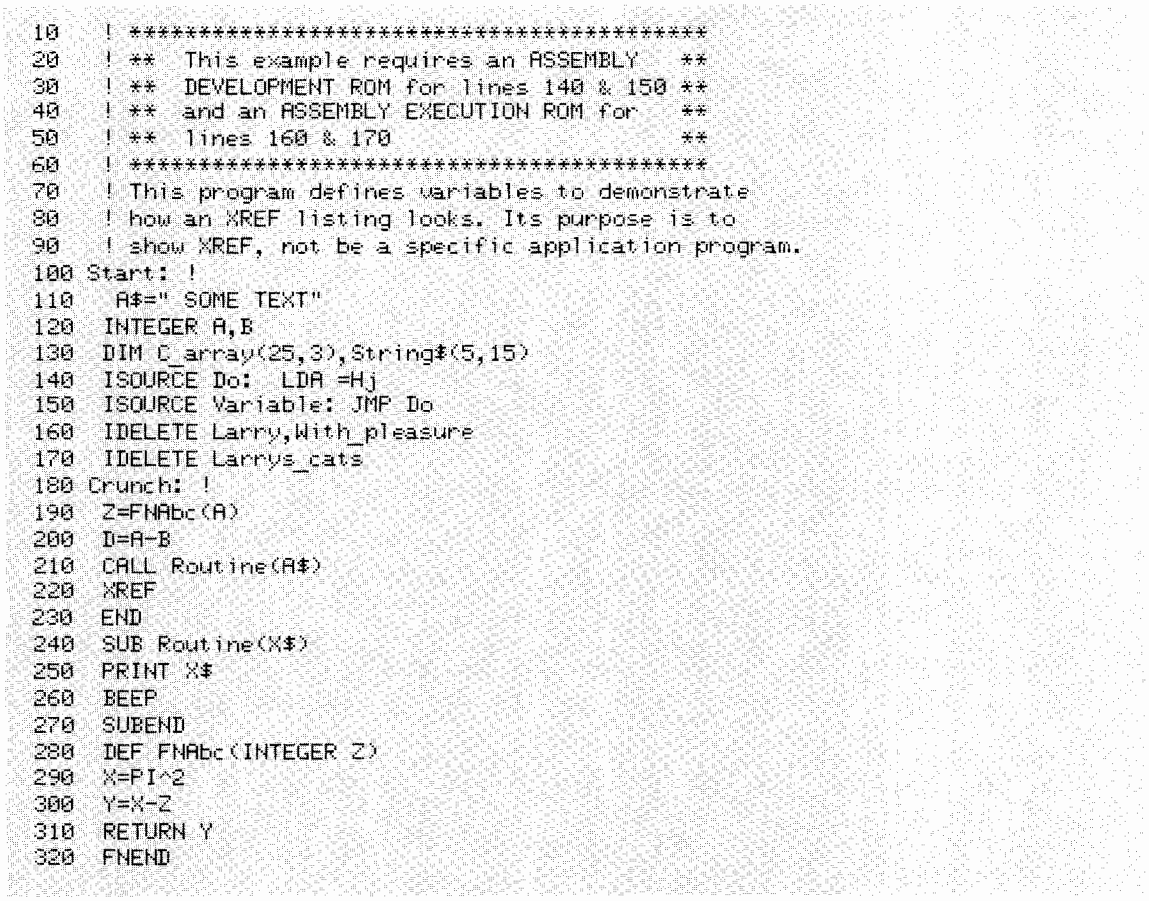
```
2)3)+
 5

2)5)+2)3)*+
 13

2.5).5)/
 5

5)2)^
 25

0)1)-
-1
```

```
10   ! ** This program provides a PRIME SIEVE routine
20   ! ** which can be compared with a PASCAL version
30   ! ** of the same program. (SIEVE) It generates all
31   ! ** the prime integers up to the number you give it.
40   INTEGER J,K,M,N
50   DIM X$[1000]
60   REPEAT
70     INPUT "Enter an integer",N   ! The REPEAT loop is used for error
80   UNTIL (N>0) AND (N<=1000)      ! checking.
90   X$=RPT$("1",N)                 ! Initialize X$ to all "1"'s.
100  K=2                            ! Initialize K.
110  REPEAT                         ! This REPEAT loop contains the
120    IF X$[K;1]="1" THEN          ! Prime Number Sieve.
130      PRINT K                    ! Print a prime number.
140      M=N DIV K         ! Calculate number of integers divisible by K
150      FOR J=1 TO M
160        X$[J*K;1]="0"   ! Wipes out multiples of K.
170      NEXT J
180    END IF
190    K=K+1               ! Increment to next value of K.
200  UNTIL K*K>N           ! Limit because all remaining values are primes.
210  REPEAT                ! This REPEAT loop checks for "1" in the string
220    IF X$[K;1]="1" THEN PRINT K    ! which signifies a prime number &
230    K=K+1               ! prints the value.
240  UNTIL K>N
250  END
```

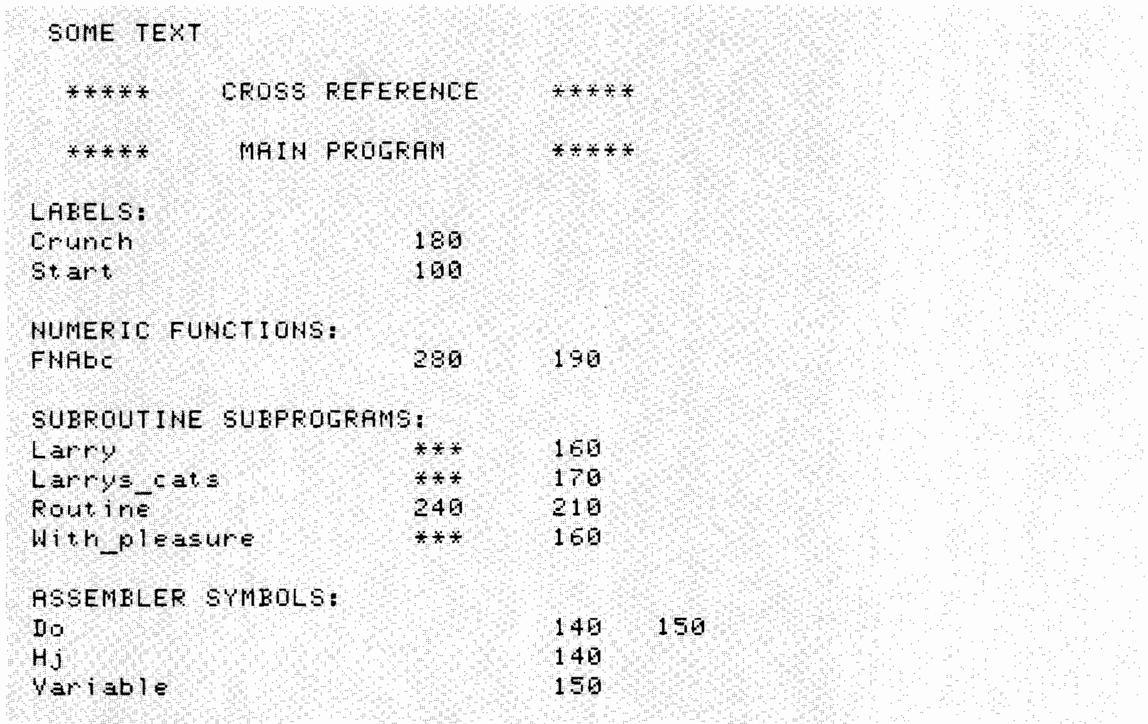
```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

PASCAL Version of the Prime Sieve Program

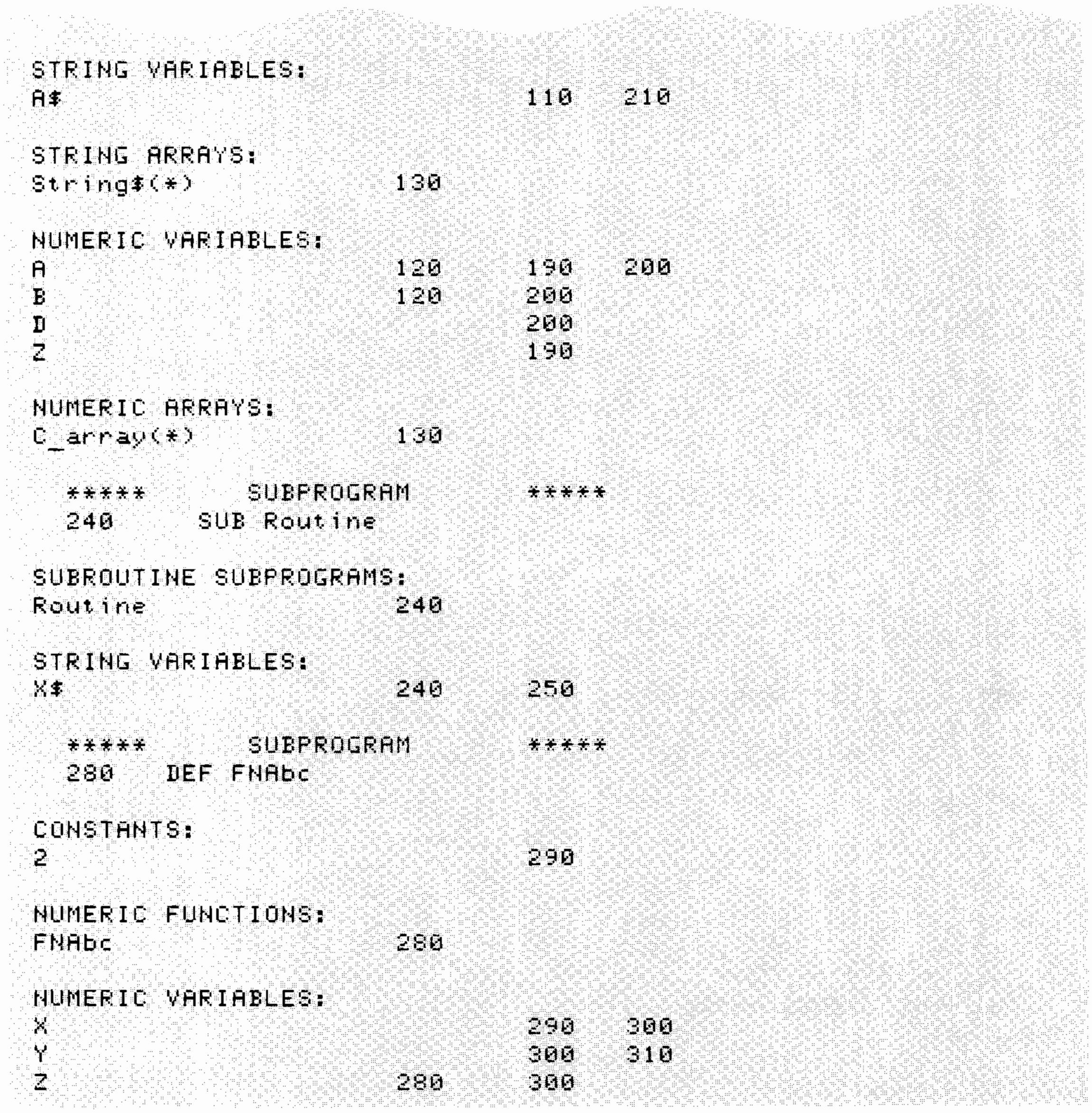
```pascal
(* prime sieve *)
program sieve(input,output);

var j,k,m,n:integer;
    x:array[1..10000] of boolean;

begin
   repeat
   begin
      write('Enter an integer: ');
      readln(n);
   end
   until (n>0) and (n<=10000);
      for j:=1 to n do x[j]:=true;   (* set all flags *)
      k:=2;
         repeat
         if x[k] then        (* k is a prime *)
            begin
               writeln(k);
               m:=(n div k);
               for j:=1 to m do x[j*k]:= false (*clear every kth flag*)
               end;
            k:=k+1;
          until k*k>n;
          repeat
             if x[k] then writeln(k);     (* k is a prime *)
             k:=k+1;
          until k>n;
       end.
```



```
10    ! *****************************************
20    ! **  This example requires an ASSEMBLY   **
30    ! **  DEVELOPMENT ROM for lines 140 & 150 **
40    ! **  and an ASSEMBLY EXECUTION ROM for   **
50    ! **  lines 160 & 170                     **
60    ! *****************************************
70    ! This program defines variables to demonstrate
80    ! how an XREF listing looks. Its purpose is to
90    ! show XREF, not be a specific application program.
100 Start: !
110   A$=" SOME TEXT"
120  INTEGER A,B
130  DIM C_array(25,3),String$(5,15)
140  ISOURCE Do:  LDA =Hj
150  ISOURCE Variable: JMP Do
160  IDELETE Larry,With_pleasure
170  IDELETE Larrys_cats
180 Crunch: !
190  Z=FNAbc(A)
200  D=A-B
210  CALL Routine(A$)
220  XREF
230  END
240  SUB Routine(X$)
250  PRINT X$
260  BEEP
270  SUBEND
280  DEF FNAbc(INTEGER Z)
290  X=PI^2
300  Y=X-Z
310  RETURN Y
320  FNEND
```



```
 SOME TEXT

  *****    CROSS REFERENCE    *****

  *****     MAIN PROGRAM      *****

LABELS:
Crunch                180
Start                 100

NUMERIC FUNCTIONS:
FNAbc                 280     190

SUBROUTINE SUBPROGRAMS:
Larry                 ***     160
Larrys_cats           ***     170
Routine               240     210
With_pleasure         ***     160

ASSEMBLER SYMBOLS:
Do                            140    150
Hj                            140
Variable                      150
```

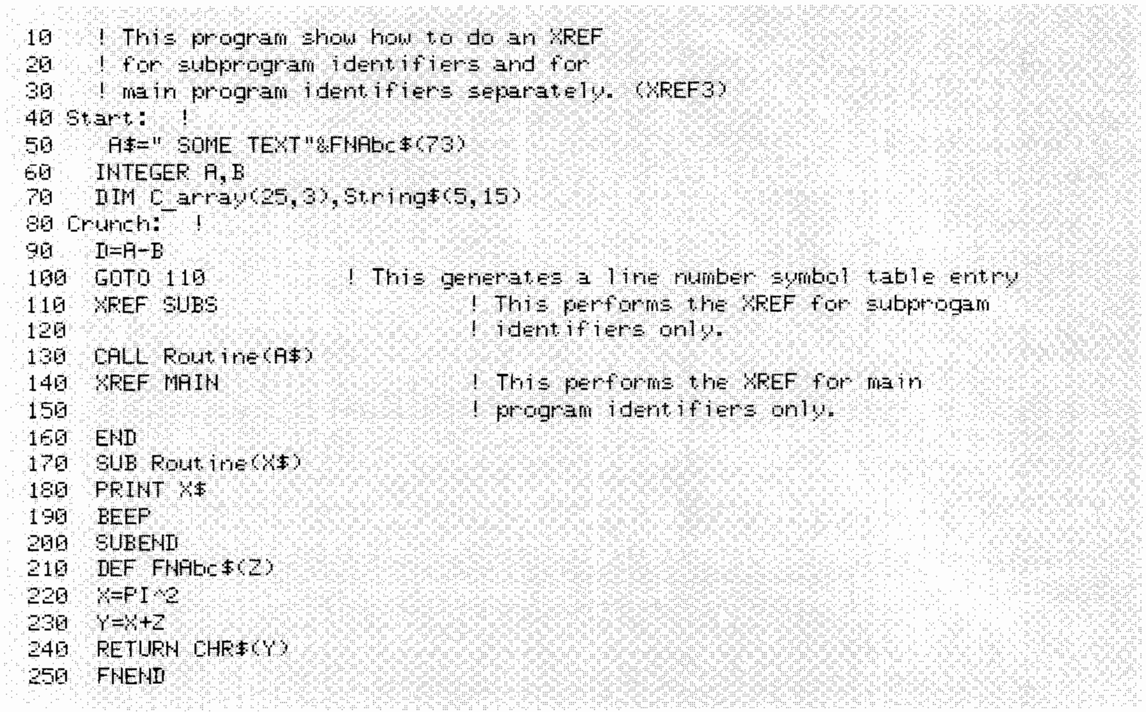


```
STRING VARIABLES:
A$                            110   210

STRING ARRAYS:
String$(*)            130

NUMERIC VARIABLES:
A                     130     190   200
B                     120     200
D                             200
Z                             190

NUMERIC ARRAYS:
C_array(*)            130

  *****      SUBPROGRAM       *****
  240     SUB Routine

SUBROUTINE SUBPROGRAMS:
Routine               240

STRING VARIABLES:
X$                    240     250

  *****      SUBPROGRAM       *****
  280   DEF FNAbc

CONSTANTS:
2                             290

NUMERIC FUNCTIONS:
FNAbc                 280

NUMERIC VARIABLES:
X                             290   300
Y                             300   310
Z                     280     300
```

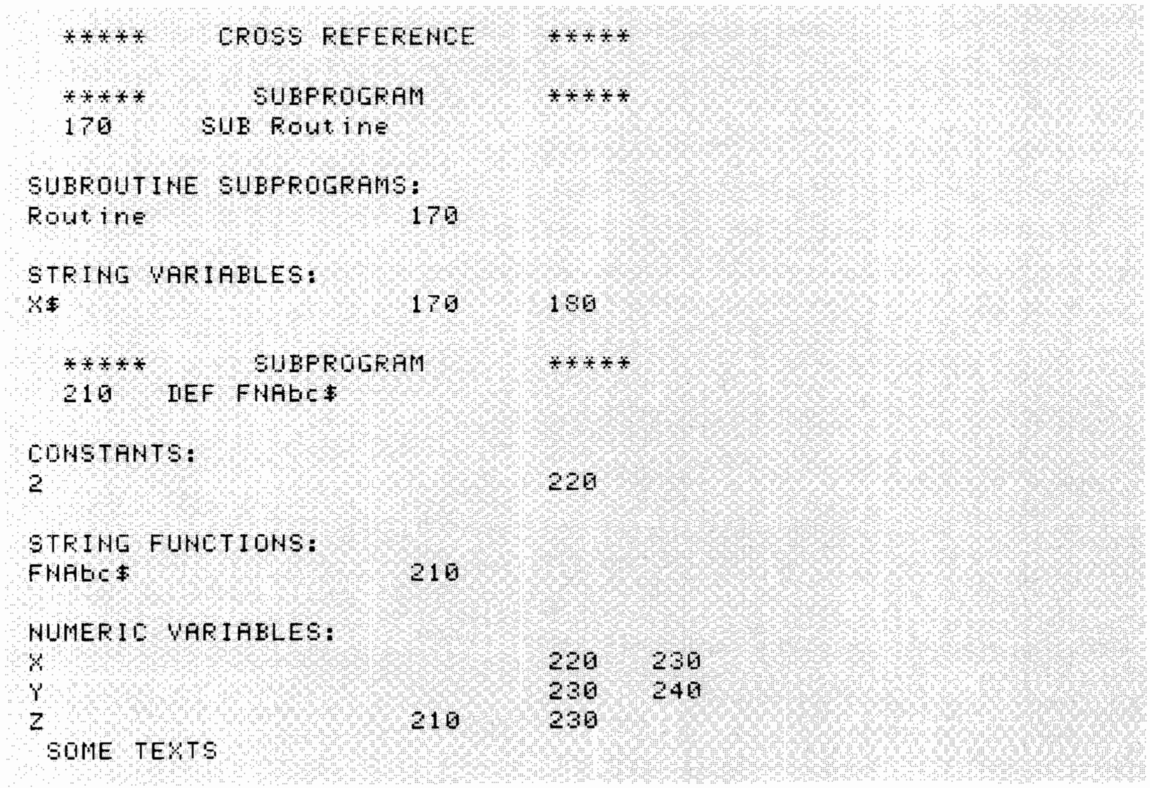
```
10   ! ***************************************************
20   ! **    This example requires a printer at 7,7     **
30   ! ***************************************************
40   ! This program shows how to do an XREF to an external
50   ! printer and use the option list. (XREF2)
60 Start:  !
70    A$=" SOME TEXT"
80   INTEGER A,B
90   DIM C_array(25,3),String$(5,15)
100 Crunch: !
110  D=A-B
120  CALL Routine(A$)
130  XREF #7,7                   ! This performs the XREF to a printer at 7,7
140  XREF #7,7;SA,NA             ! This performs the XREF to a printer at 7,7
150                              ! Note that a  ;  is needed in line 120
160  END
170  SUB Routine(X$)
180  PRINT X$
190  BEEP
200  SUBEND
```

```
 SOME TEXT

  *****    CROSS REFERENCE    *****

  *****     MAIN PROGRAM      *****

CONSTANTS:
7                             130   140

LABELS:
Crunch                100
Start                  60

SUBROUTINE SUBPROGRAMS:
Routine               170     120

STRING VARIABLES:
A$                             70   120

STRING ARRAYS:
String$(*)             90

NUMERIC VARIABLES:
A                      80     110
B                      80     110
D                             110

NUMERIC ARRAYS:
C_array(*)             90

  *****      SUBPROGRAM       *****
  170     SUB Routine
```

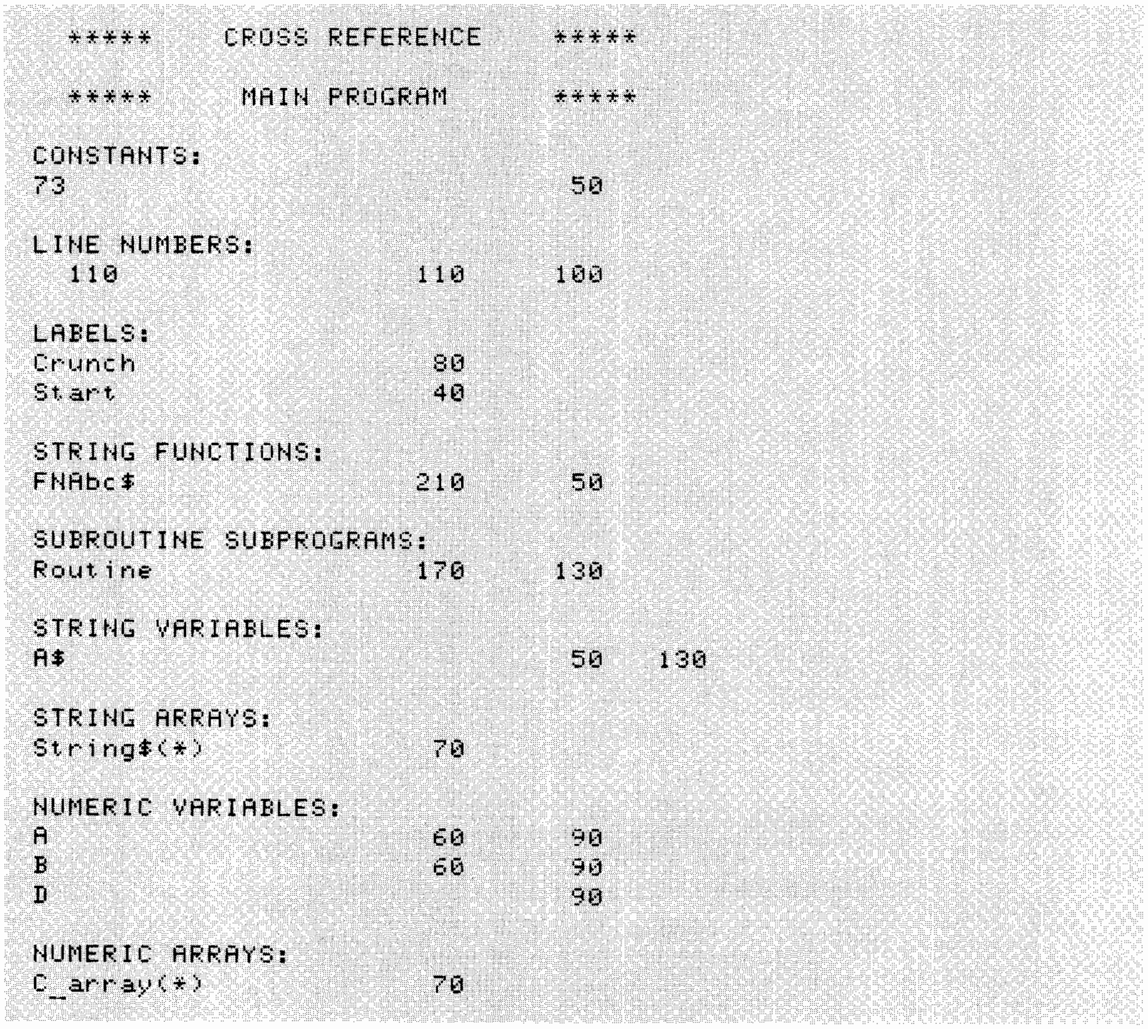
```
SUBROUTINE SUBPROGRAMS:
Routine                   170

STRING VARIABLES:
X$                        170       180




  *****    CROSS REFERENCE     *****

  *****     MAIN PROGRAM       *****

STRING ARRAYS:
String$(*)                 90

NUMERIC ARRAYS:
C_array(*)                 90

  *****      SUBPROGRAM        *****
  170     SUB Routine
```



```
10   ! This program show how to do an XREF
20   ! for subprogram identifiers and for
30   ! main program identifiers separately. (XREF3)
40 Start:  !
50    A$=" SOME TEXT"&FNAbc$(73)
60   INTEGER A,B
70   DIM C_array(25,3),String$(5,15)
80 Crunch:  !
90   D=A-B
100  GOTO 110          ! This generates a line number symbol table entry
110  XREF SUBS                  ! This performs the XREF for subprogam
120                             ! identifiers only.
130  CALL Routine(A$)
140  XREF MAIN                  ! This performs the XREF for main
150                             ! program identifiers only.
160  END
170  SUB Routine(X$)
180  PRINT X$
190  BEEP
200  SUBEND
210  DEF FNAbc$(Z)
220  X=PI^2
230  Y=X+Z
240  RETURN CHR$(Y)
250  FNEND
```



```
  *****    CROSS REFERENCE    *****

  *****      SUBPROGRAM       *****
  170     SUB Routine

SUBROUTINE SUBPROGRAMS:
Routine                 170

STRING VARIABLES:
X$                      170     180

  *****      SUBPROGRAM       *****
  210    DEF FNAbc$

CONSTANTS:
2                               220

STRING FUNCTIONS:
FNAbc$                  210

NUMERIC VARIABLES:
X                               220    230
Y                               230    240
Z                       210     230
 SOME TEXTS
```

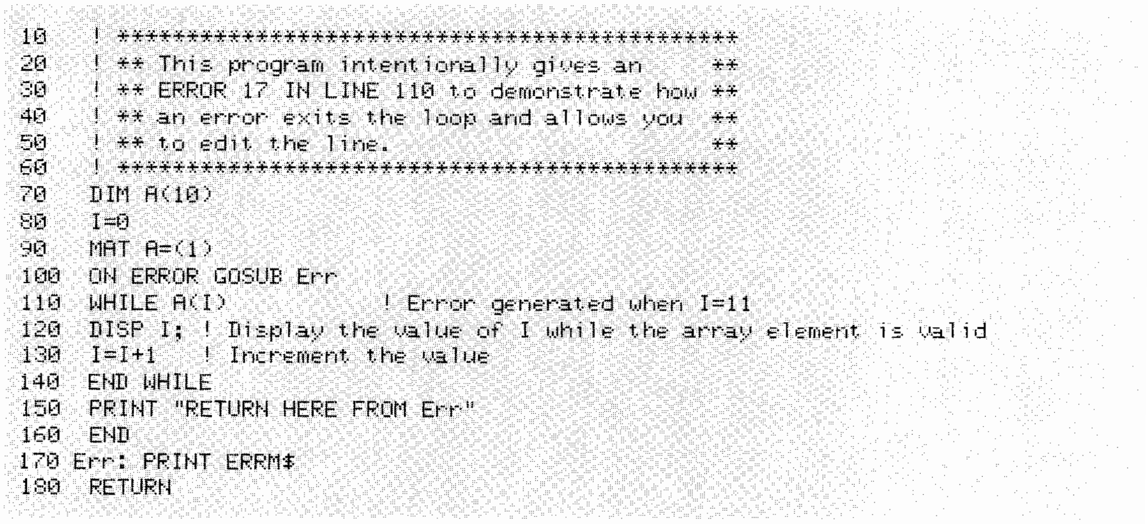


```
  *****    CROSS REFERENCE    *****

  *****     MAIN PROGRAM      *****

CONSTANTS:
73                             50

LINE NUMBERS:
  110                 110     100

LABELS:
Crunch                 80
Start                  40

STRING FUNCTIONS:
FNAbc$                210      50

SUBROUTINE SUBPROGRAMS:
Routine               170     130

STRING VARIABLES:
A$                             50   130

STRING ARRAYS:
String$(*)             70

NUMERIC VARIABLES:
A                      60      90
B                      60      90
D                              90

NUMERIC ARRAYS:
C_array(*)             70
```

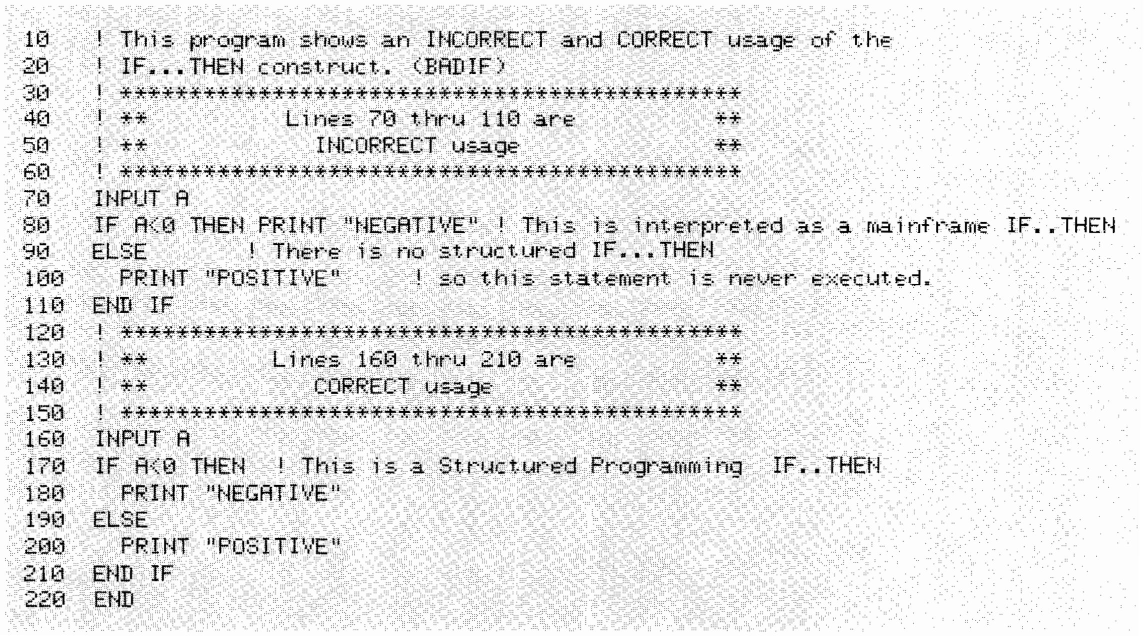
```
10    ! This is an XREF listing of a secured
20    ! version of XREF3 so you can compare
30    ! a secured and unsecured XREF. (XREF5)
40  *
50  *
60  *
70  *
80  *
90  *
100  *
110  *
120  *
130  *
140  *
150  *
160  *
170  *
180  *
190  *
200  *
210  *
220  *
230  *
240  *
250  *
```

```
  *****    CROSS REFERENCE    *****

  *****      SUBPROGRAM       *****
  170   SECURED

SUBROUTINE SUBPROGRAMS:
Routine                  ***

STRING VARIABLES:
X$                       ***

  *****      SUBPROGRAM       *****
  210   SECURED

CONSTANTS:
2

STRING FUNCTIONS:
FNAbc$                   ***

NUMERIC VARIABLES:
X
Y
Z                        ***
```

```
 SOME TEXTS


  *****    CROSS REFERENCE    *****

  *****     MAIN PROGRAM      *****

CONSTANTS:
73

LINE NUMBERS:
  110                 ***

LABELS:
Crunch                ***
Start                 ***

STRING FUNCTIONS:
FNAbc$                ***

SUBROUTINE SUBPROGRAMS:
Routine               ***

STRING VARIABLES:
A$

STRING ARRAYS:
String$(*)            ***

NUMERIC VARIABLES:
A                     ***
B                     ***
D

NUMERIC ARRAYS:
C_array(*)            ***
```



```
10   ! *********************************************
20   ! ** This program intentionally gives an     **
30   ! ** ERROR 17 IN LINE 110 to demonstrate how **
40   ! ** an error exits the loop and allows you  **
50   ! ** to edit the line.                       **
60   ! *********************************************
70   DIM A(10)
80   I=0
90   MAT A=(1)
100  ON ERROR GOSUB Err
110  WHILE A(I)           ! Error generated when I=11
120  DISP I; ! Display the value of I while the array element is valid
130  I=I+1   ! Increment the value
140  END WHILE
150  PRINT "RETURN HERE FROM Err"
160  END
170 Err: PRINT ERRM$
180  RETURN
```



```
10   ! This program shows an INCORRECT and CORRECT usage of the
20   ! IF...THEN construct. (BADIF)
30   ! *********************************************
40   ! **         Lines 70 thru 110 are           **
50   ! **           INCORRECT usage               **
60   ! *********************************************
70   INPUT A
80   IF A<0 THEN PRINT "NEGATIVE" ! This is interpreted as a mainframe IF..THEN
90   ELSE       ! There is no structured IF...THEN
100    PRINT "POSITIVE"      ! so this statement is never executed.
110  END IF
120  ! *********************************************
130  ! **        Lines 160 thru 210 are           **
140  ! **           CORRECT usage                 **
150  ! *********************************************
160  INPUT A
170  IF A<0 THEN  ! This is a Structured Programming  IF..THEN
180    PRINT "NEGATIVE"
190  ELSE
200    PRINT "POSITIVE"
210  END IF
220  END
```

Appendix A

Language Translation and Comparison

	Structured Programming using HP Extended BASIC	**Fortran (X3J3/90.5 unofficial)**	**PASCAL (P4 compiler)**
Variable Types	Integer Short Real String Arrays of all above	Integer Double Precision Real Character Logical Complex Arrays of all above	Integer Real Char Boolean User Defined Arrays of all above
Variable Names	15 characters	5 characters	8 characters significant

Looping Constructs	REPEAT
	UNTIL..
	WHILE..
	END WHILE
	LOOP
	EXIT IF..
	END LOOP
	FOR..TO..
	NEXT..
		DO..(start,stop,inc)	REPEAT
			UNTIL..
			WHILE..DO
			END
			FOR..TO..

Decision Constructs	IF..THEN
	ELSE
	END IF
	SELECT
	CASE
	CASE ELSE
	END SELECT
		IF..
		ELSE
		END IF
			IF..THEN
			ELSE
			END
			CASE..OF
			END

Multiple Environments	CALL SUB SUBEND Functions	Subroutines Functions	Procedures Functions
Recursion	Yes	No	Yes
Operators			
Matrix	Yes	No	No
String	Yes	No	No
Dimensioning			
Dynamic	Yes (Subprograms)	No	No
Re-Dim	Yes	No	No
Formats			
Input	Yes	Yes	No
Output	Yes	Yes	Limited

Your Comments, Please...

Your comments assist us in improving the usefulness of our publications; they are an important part of the inputs used in preparing updates to the publications.

In order to write this manual, we made certain assumptions about your computer background. By completing and returning the comments card on the following page you can assist us in adjusting our assumptions and improving our manuals.

Feel free to mark more than one reply to a question and to make any additional comments.

Please do not use this form for questions about technical applications of your system or requests for additional publications. Instead, direct those inquiries or requests to your nearest HP Sales and Service Office.

If the comments card is missing, please address your comments to:

HEWLETT-PACKARD COMPANY
Desktop Computer Division
3404 East Harmony Road
Fort Collins, Colorado 80525 U.S.A.

Attn. Customer Documentation
Dept. 4231

All comments and suggestions become the property of Hewlett-Packard.