HP Business BASIC/XL Reference Manual

HP 3000 MPE/iX Computer Systems

Edition 1

HEWLETT® PACKARD

Manufacturing Part Number: 32715-90001
E1187

U.S.A. November 1987

Restricted Rights Legend

Acknowledgments

UNIX is a registered trademark of The Open Group.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Chapter 1 Introductions

HP Business BASIC/XL is a high level programming language implemented on the 900 Series HP 3000. The BASIC language was developed to teach beginners about computer programming. HP Business BASIC/XL takes advantage of that ease of use, yet also provides a full interface to the powerful MPE XL operating system.

HP Business BASIC/XL contains built-in interfaces to the IMAGE database management system, VPLUS screen handler, and terminal softkeys. It also features a report generator, sophisticated error handling capabilities, and a static analyzer in the interpreter.

HP Business BASIC/XL has an interpreter and a compiler. Programs can be developed with the interpreter, which has the features of an editor, debugger, and calculator. Working programs can be compiled to increase their speed and decrease their required storage space.

The interpreter checks the syntax of each line as it is entered, provides immediate feedback about syntax errors and the effect of program modifications, and allows quick transition between editing and running a program.

HP Business BASIC/XL provides many statements and commands that facilitate debugging. The interpreter's HELP command provides immediate information about the syntax and function of any HP Business BASIC/XL keyword or statement. Used as a calculator, the interpreter returns the value of any expression. TRACE statements print messages when one line transfers control to another line or when variables are assigned a value.The PAUSE statement suspends program execution, during which time variable values and program lines can be displayed and modified. Program execution can then be continued, and the effect of any changes can be examined.

Chapter 2 Program Development Environment

Introduction

HP Business BASIC/XL's program development environment is a line editor and program interpreter that aids in program development and manages program files. HP Business BASIC/XL also has a compiler to compile programs. Compiled programs run faster than interpreted ones, so it is often a good idea to develop a program in the interpreter and compile it once it is running correctly. The compiler is explained in chapter 9.

The material in this chapter is summarized below:

TITLE	**CONTENT**
The Interpreter	How to enter the interpreter.
The Current Program	General information about the structure of a program currently in the interpreter.
Creating and Modifying a Program	How to create and modify an HP Business BASIC/XL program; how to name it, list it, and protect it from listing and modification.
Managing Program Files	How to save a program on a disk file, retrieve it, and execute it; how to make it executable but protect it from listing and modification.
Debugging a Program	How to suspend program execution to examine variable values and change code, and then resume execution from the same point; how to trace line execution and changes in variable values.
HELP Statement	How to display information about HP Business BASIC/XL statements, commands, and errors.
Accessing the Operating System	How to execute operating system commands from HP Business BASIC/XL.
Calculator Mode	How HP Business BASIC/XL evaluates expressions that are not within programs.

NOTE In the examples in this chapter, user input is underlined. User input ends with RETURN unless otherwise specified. In some examples, RETURN is shown to clarify that example.

The Interpreter

Typing the command BBXL in response to the operating system prompt will run the interpreter. You can use options to specify a file from which input will be read or a file to which output will be written or both. You can also use a file which contains commands to be executed by the interpreter, called a command file.

Syntax

```
BBXL [cfile [,ifile [,ofile [,xlfile ]]]]
```

Parameters

cfile Command file which can contain both commands and program lines. The command file *cfile* must be an ASCII file. Its formal file name is BASCOM and its default assignment is $STDINX.

ifile HP Business BASIC/XL program input file which contains data for INPUT statements. The input file *ifile* must be an ASCII file. Its formal file name is BASIN and its default assignment is $STDINX.

ofile HP Business BASIC/XL program output file, that the PRINT statement sends output to. The output file *ofile* must be an ASCII file. Its formal file name is BASOUT and its default assignment is $STDLIST.

xlfile The *xlfile* parameter specifies one or more executable libraries to the interpreter. A single library may be referenced by entering the fully qualified library file name. Two or more libraries may be referenced by enclosing the list of libraries in quotes, separating each name with commas, semicolons, or spaces.

Examples

The first example below uses a command file called Command, and uses the executable libraries Lib.Pub.Sys and Mylib in the log on group and account. The second example specifies an input file (Infil) and an output file (Outfil).

```
BBXL Command,,,"Lib.Pub.Sys,Mylib.!hpgroup.!hpacct"
```

```
     BBXL ,Infil,Outfil
```

HP Business BASIC/XL can also be run as a program using the following syntax:

```
     RUN HPBBXL.PUB.SYS [;PARM=n ]
```

The PARM option is used to specify two things to the interpreter:

1. How much space the interpreter should reserve for representing the currently-executing subunit.

2. Which of the BASCOM, BASIN, or BASOUT files has been respecified using a file equate. *n* specifies which of the parameters have been redefined. The following are the values of *n*:

   0 No redefinition of the files.

   1 BASCOM has been redefined.

   2 BASIN has been redefined.

   3 BASCOM and BASIN have been redefined.

   4 BASOUT has been redefined.

   5 BASCOM and BASOUT have been redefined.

   6 BASIN and BASOUT have been redefined.

   7 BASCOM, BASIN, and BASOUT have been redefined.

To set both of these parameters on the same run of the interpreter, add the two values together and use their sum as the PARM value.

Consider the following two files (HELLO and RUNHELLO) in the following example:

The HELLO file contains the HP Business BASIC/XL program:

```
     10 PRINT "HELLO"
```

The RUNHELLO file contains the commands:

```
     GET HELLO
     RUN
     EXIT
```

You can run the HELLO program by typing in the following command in response to the operating system prompt:

```
     BBXL RUNHELLO
```

The commands in the RUNHELLO file are executed by HP Business BASIC/XL's interpreter. In response to the RUN command, "HELLO" is printed on the terminal's screen. Incorporating the command into a stream job has the same effect.

Redirecting BASCOM, BASIN, and BASOUT is useful when running stream jobs.

Any of the file parameters can be specified by a local file equate statement.

The HELLO program can be run by typing the following commands in response to the operating system prompt or by including the commands in a stream file, as illustrated below:

```
     FILE BASCOM = RUNHELLO
     RUN HPBBXL.PUB.SYS;PARM=1
```

The Current Program in the Interpreter

Within the interpreter, the program being created, modified, executed, or debugged resides in the interpreter's work space. This program is referred to as the current program.

The current program can be permanently saved in a disk file by using the SAVE and RESAVE commands. The GET command is used to read the contents of a permanent disk file into the interpreter's work space.

Line Ranges

Many commands and statements in this chapter operate on ranges of program lines. In syntax specifications, *line_range* is a range of lines and *line_range_list* is a list of line ranges.

The syntax of *line_range_list* is shown below.

Syntax

line_range [, *line_range*]...

Parameters

line_range One of the following:

ALL
su_spec
ln_spec1 [/ *ln_spec2*]

ALL All program lines. In a command or statement where *line_range* or *line_range_list* is optional, ALL is the default unless otherwise specified.

su_spec One of the following program unit specifiers:

SUB *sub_id*	Range is all of *sub_id*, which must exist.

[SUB] FN*func_id* — Range is all of FN*func_id*, which must exist.

MAIN — Range is all of the main program, which must exist.

ln_spec1 — First line in range, specified by one of the following:

line_num — If *line_num* does not exist, *line_range* is null (see Table 2-1 and Table 2-2).

line_num {+|-} *offset* — The line that is *offset* lines from the line numbered *line_num* (see *offset*, below). The line numbered *line_num* must exist in the program. If *line_num* {+|-} *offset* does not exist, the existing line nearest it is used (see Table 2-1 and Table 2-2).

FIRST — First program line.

LAST — Last program line.

* — Last line executed (undefined for a stopped program). The command LIST * displays this line.

line_label — Must be defined in the currently executing program unit.

line_label {+|-} *offset* — The line that is *offset* lines from the line labeled *line_label* (see *offset*, below). The label *line_label* must be defined in the currently executing program unit. If *line_label* {+|-} *offset* does not exist, the line nearest it is used (see Table 2-1 and Table 2-2).

MAIN — The first line of the main program. The main program must exist.

SUB *sub_id* — The first line of the subprogram *sub_id*. The subprogram must exist.

[SUB] FN*func_id* — The first line of the function FN*func_id*. The function must exist.

offset — Number of actual lines past (or before) *line_num*. For example, 10+3 is the line three lines from line 10. This is not necessarily line 13. In the following program, it is line 50:

```
10 PRINT "HI"
20 X=4
30 PRINT X
50 END
```

ln_spec2 Last line in range. The *line_num* through *line_label* {+|-} *offset* are the same as specified for *ln_spec1*. MAIN, SUB and FN change to the corresponding last line in each.

MAIN	Last line in main program. The default is *ln_spec1*.
SUB *sub_id*	Last line in *sub_id*.
[SUB] FN*func_id*	Last line in FN*func_id*.

Examples

The following shows examples of specifying line ranges.

```
 ALL                       !Specifies all lines
SUB Sub1                   !Specifies all lines in Sub1
FNAdd, FNScramble$         !Specifies all of FNAdd and FNScrable$
MAIN, FNAdd/FNScramble$    !Specifies all of FNAdd, FNScramble, and
                           !all of the main program
100,1000+50,Label+50       !Specifies lines 100, and the lines that are 50
                           !lines past 1000 and 50 lines past Label
FIRST+100/LAST-350         !Specifies 100 lines past FIRST through 350 lines
                           !before LAST
*/LAST                     !Specifies the last executed line through the
                           !last program line
*-50/*+10,SUB Sub1/LAST    !Specifies 50 lines before the last executed
                           !line through 10 lines after the last executed
                           !line, and the first line of Sub1 through the
                           !last program line
FIRST/FNAdd                !Specifies the first program line through the
                           !last line of FNAdd
MAIN/2000                  !Specifies the first line of the main program
                           !through line 2000
100/150                    !Specifies lines 100 through 150
```

Table 2-1 shows where the line range begins when *ln_spec1* is not in the program. Table 2-2 shows where the line range ends when *ln_spec2* is not in the program.

Table 2-1. Where Line Range Begins When *ln_spec1* is Not in Program

ln_spec1	***ln_spec2* is Specified**	**Program does not have *line_num***	**Program does not have *line_num* {+ | -} *offset* (but has *line_num*)**
line_num	No	Nothing happens.	Not applicable.
line_num	Yes	Range begins with existing line number that is closest to but greater than *line_num*.	Not applicable.
line_num {+|-}*offset*	Irrelevant	Error occurs.	If *line_num* {+|-}*offset*

			is before first program line, range begins with first program line. If *line_num* {+|-}*offset* is after last program line, range begins with last program line.

Table 2-2. Where Line Range Ends When *ln_spec2* is Not in Program

ln_spec2	**Program does not have *line_num***	**Program does not have *line_num*{+ | -}*offset* (but has *line_num*)**
line_num	Range ends with existing line number that is closest to but less than *line_num*.	Not applicable.
line_num {+| -}*offset*	Error occurs.	If *line_num* {+|-}*offset* is before first program line, range ends with first program line. If *line_num* {+|-}*offset* is after last program line, range ends with last program line.

Examples

Refer to this program when reading the examples that follow it:

```
100 A=3
110 B=4
120 PRINT A,B
130 Add: C=A+B
140 PRINT C
150 END
160 DEF FNTwo
170 PRINT "In FNTwo"
180  RETURN 2
190 FNEND
```

Range Specified	**Range Used (or Effect)**
10	Nothing happens
10/120	100/120
10/125	100/120
110	110
10+1	Error
10+1/130	Error
100+2	120

100+2/140	120/140
100+2/145	120/140
100+2/150-1	120/140
200-3	Error
100/200-3	Error
110-3/140-1	100/130
130+5	180
Add	130 If the main unit is currently executing, otherwise an error results.
Add-1/Add+1	120/140 If the main unit is currently executing, otherwise an error results.
MAIN	100/150
FNTwo	160/190

Halt Key

Pressing CONTROL Y while a program is executing suspends the program. Any I/O (Input or Output) operation that is in process finishes before program execution is suspended. For example, if the program is reading a disk file, that read will complete before the program is suspended. When the HALT is executed, the cursor appears on the terminal screen. To resume program execution, use the CONTINUE command, described later in this chapter.

Pressing the halt key twice in rapid succession suspends the program, but any I/O operation that is in progress is aborted.

INDENT Command

The INDENT command indents the bodies of constructs. This tool makes it easy to see the nesting level of the program's constructs. The INDENT command modifies lines without displaying them.

Syntax

INDENT [*num_expr1* [, *num_expr2*]]

Parameters

num_expr1 The value *num_expr1* +8 is the starting column number of every line that is not in the body of a structured statement. The value of *num_expr1* must be in the range [1,63]. Default is one.

num_expr2 Increment for calculating starting column numbers of nested (indented) lines. If a line is in the body of one structured statement, it is nested (and indented) once and begins in column(*num_expr1* +8)+*num_expr2*. If a line is in the body of *n* structured statements, it is nested (and indented) *n* times, and begins in column (*num_expr1* +8)+(*n* **num_expr2*). The value of *num_expr2* must be in the range [0,63]. The default is three.

The INDENT statement indents the part of the line that follows the line number and label. It does not indent the line number or the label of a line. The line number is always right-justified in columns two through seven. For a labeled line, the indented line will contain the line number, one blank space, the label and a colon (:). The rest of the line begins in the column specified by the INDENT command, with two exceptions:

* If the label or colon occupies the specified column, then the rest of the line begins in the next available column.

* If the specified column is beyond 72, then the rest of the line begins in column 72.

A comment (beginning with "!") is listed in the column originally entered (relative to the line number), if possible. If this is not possible because the statement occupies that column, then the comment begins in the next available column.

If a modified line is too long, the LIST command displays:

* The line, except characters beyond the maximum line length.
* An asterisk (*) in the last column of the line (the asterisk is character 500).

Examples

The following example shows the effect of the INDENT command. First, the starting column of each line is set to seven and each nested line is indented three. The second INDENT command changes the starting column to three, and the indentation to five.

```
>list
 !  exam217
      5 ! BEGIN PROGRAM
     10 DIM A(5),B$(2,4)[2]
     20 INTEGER X,Y,Z
     30 Loop1: FOR I=1 TO 5  !Fill A
     40 A(I)=I
     45 PRINT I
     50 NEXT I
     60 Loop2: FOR I=1 TO 2  !Fill B
     70 FOR J=1 TO 4
     75 REM INNER LOOP
     80 B$(I,J)=CHR$(I)+CHR$(J)
     85 PRINT I,J
     90 NEXT J
    100 NEXT I
    999 END
>indent 7,3
>list
 !  exam217
      5 ! BEGIN PROGRAM
     10       DIM A(5),B$(2,4)[2]
     20       INTEGER X,Y,Z
     30 Loop1: FOR I=1 TO 5  !Fill A
     40          A(I)=I
     45          PRINT I
     50       NEXT I
     60 Loop2: FOR I=1 TO 2  !Fill B
     70         FOR J=1 TO 4
     75            REM INNER LOOP
     80            B$(I,J)=CHR$(I)+CHR$(J)
     85            PRINT I,J
     90         NEXT J
    100       NEXT I
    999       END
>indent 3,5
>list
 !  exam217
      5 ! BEGIN PROGRAM
     10   DIM A(5),B$(2,4)[2]
     20   INTEGER X,Y,Z
     30 Loop1: FOR I=1 TO 5  !Fill A
     40        A(I)=I
     45        PRINT I
     50   NEXT I
     60 Loop2: FOR I=1 TO 2  !Fill B
     70        FOR J=1 TO 4
     75             REM INNER LOOP
     80             B$(I,J)=CHR$(I)+CHR$(J)
     85             PRINT I,J
     90        NEXT J
    100   NEXT I
    999   END
>
```

LIST Command

The LIST command lists all or part of a program to the destination specified by the SEND SYSTEM OUTPUT TO statement (usually the terminal) or to a specified device (usually a spooled printer). The LIST command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program. Compiler formatting options can be used to print page titles or page numbers, control the number of lines printed per page and print a list of the identifiers in the program.

Syntax

```
                                     [ {NONAME   }]
LIST [line_range_list ] [TO dev_spec ] [;{FORMATTED}]
                                     [ {FORMAT   }]
```

Parameters

dev_spec See "Device Specification Syntax," in chapter 6. If this parameter is specified, the LIST command lists the lines on the specified device (*dev_spec*); otherwise, it appends them to the file specified by the most recently executed SEND SYSTEM OUTPUT TO statement.

NONAME The program name is not listed if this parameter is specified. This is relevant only when the program has a name, that is, if it was retrieved by the GET command or named with the NAME command. If you have just typed in the program, and have not used the NAME command, the program will not have a name.

FORMATTED
FORMAT
The listing is formatted using a set of the compiler listing options that appear in the program if this parameter is specified. The set of COPTIONS used to format the interpreter listing are: LINES, LIST, ID TABLES, PAGE, PAGESUB, TITLE, and TITLESUB. These are explained in chapter 9.

The LIST command may add or remove blanks and parentheses to make lines more readable. It also does the following:

* Lists lines in line number order, whether or not they were entered in that order.
* Lists identifiers with first letters upshifted and all other letters downshifted.
* Lists keywords in all uppercase letters.
* Lists empty statements as empty comments (for example, it lists the line "500 Label:" as "500 Label: !").

The column at which long lines are broken depends on the output device and WIDTH. On a terminal screen, the default line length is 80; on a line printer, it is 132.

If a line exceeds the maximum length, the LIST command prints an asterisk (*) in its last column and truncates the line at the maximum length.

To stop the LIST command, press CONTROL Y.

Examples

The following example shows the LIST command. Without parameters, the LIST command below displays the entire program. When LIST has the parameter 10/90, lines 10 through 90 are displayed.

```
>10 real   ALPHA, BeTa,delta
>5 SHORT c, d, e
>100 !
>73 Correction:
>LIST
    5 SHORT C,D,E
   10 REAL Alpha,Beta,Delta
   73 Correction: !
  100 !
>LIST 10/90
   10 REAL Alpha,Beta,Delta
   73 Correction: !
>
```

LIST SUBS Command

The LIST SUBS command prints the name and starting line number of every subunit in the program, and indicates the currently executing program unit with an asterisk (*). The LIST SUBS command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program.

Syntax

```
LIST SUBS
```

Example

The following example shows the use of the LIST SUBS command. When you type LIST SUBS, the first line and subunit name are displayed for each subunit in the program.

```
>LIST
     10 PRINT "this is the main"
     20 CALL A
     100 SUB A
     110    PRINT " in sub a"
     120    PRINT FNX
     130 SUBEND
  100000 DEF FNX
  100100    PRINT " in fnx"
  100200    RETURN 5
  100300 FNEND
>LIST SUBS
First Line   Subunit name
----------   ------------
      10     MAIN *
     100     A
  100000     FNX
```

MODIFY Command

The MODIFY command replaces, deletes, or inserts characters in one or more program lines. The MODIFY command is a command-only statement. That is, it can only be issued at the interpreter prompt, and cannot be placed in a program.

Syntax

{MODIFY}
{MOD   } [*line_range_list*]

The MODIFY command displays the lines of *line_range_list* one at a time. If a program line occupies more than one physical line, each physical line is displayed separately. When a line is displayed, the cursor is positioned immediately under the beginning of that portion of the line to be modified. Choose one of the editing commands in Table 2-4 or type thecharacters to be replaced.

After editing the line, press RETURN. The MODIFY command displays the modified line for further modification. When you are finished with the modifications, type RETURN after the modified line is displayed. HP Business BASIC/XL modifies the line and, if it is correct, incorporates the modified line into the program. Then the next line in the *line_range_list* is displayed for modification. If the modified line has a syntax error, the error message associated with that error is displayed and you return to the MODIFY mode for that line.

If you have difficulty modifying the line and wish to start with the version of the line that you had when you began, type "//" or CONTROL Y.

Table 2-4. MODIFY Subcommands

Subcommand	Modification	Usage
D (or d)	Delete one character or a series of characters.	Type D under each character to be deleted.
D (or d)	Delete a series of characters.	Type D under the first and last characters to be deleted.
D (or d)E (or e)	Delete from one character to the end of the line.	Type D under the first character to be deleted and E (or e) immediately after D.
D(or d)&	Delete from one character to the continuation character (&).	Type D under the first character to be deleted and & immediately after D.
I (or i)	Insert characters.	Type I under the character before which characters are to be inserted; after I, type the characters to be inserted.
R (or r)	Replace characters.	Type R under the first character of the string to be replaced. After R, type the *n* characters that will replace the first character and the next *n* -1 characters.
// or CONTROL Y	Cancel modifications on current line.	Type one of the following under the line: // or CONTROL Y.
Any other character	Replace characters.	Type the characters that will replace those in the preceding line.

If a modified line is too long, the MODIFY command displays the following:

- The line, except characters beyond the maximum line length.
- An asterisk (*) in the last column (the asterisk is character 500).

To cancel modifications on a line, type "//" or CONTROL Y. The original line will be displayed for modification.

To stop the MODIFY command, type "//" or CONTROL Y before modifying the currently displayed line, or type "//" or CONTROL Y immediately followed by "//" or CONTROL Y.

Examples

The following examples show the use of the MODIFY command. Lines 30,40, 50, and lines 100 through 180 are modified. "//" is used to cancel the modification of line 150.

```
>LIST
    10 REM 5/21/84
    20 PRINT "BEGIN"
    30 SHORT INTEGER A,B
    40 SHORT INTEGER C,D
    50 INTEGER E,F,G,H,I
    60 READ A,B
    70 READ C,D
    80 READ E,F
    90 DATA 1,2,3,4,5,6,7,8,9
   100 PRINT "A,B,E,F"
   110 PRINT A,B,E,F
   120 PRINT G,H,I
   130 PRINT A,B,H,I
   140 PRINT I,H,G,A,B,C
   150 PRINT E,F,G,H
   160 PRINT F,A,B,C,D,E
   170 PRINT
   180 PRINT "END"
   999 END

>MODIFY 30/50,100/180
    30 SHORT INTEGER A,B
       DDDDDD  RETURN
    30 INTEGER A,B
    RETURN
    40 SHORT INTEGER C,D
       D    D  RETURN
    40 INTEGER C,D
    RETURN
    50 INTEGER E,F,G,H,I
                  DE  RETURN
    50 INTEGER E,F
    RETURN
    100 PRINT "A,B,E,F"
                   IC,D,  RETURN
    100 PRINT "A,B,C,D,E,F"
    RETURN
    110 PRINT A,B,E,F
                  RC,D,E,F  RETURN
    110 PRINT A,B,C,D,E,F
    RETURN
    120 PRINT G,H,I
              B,C,D,E,F,A  RETURN
    120 PRINT B,C,D,E,F,A
    RETURN
    130 PRINT A,B,H,I
              IC,D,E,F,  RETURN
    130 PRINT C,D,E,F,A,B,H,I
                         DE  RETURN
    130 PRINT C,D,E,F,A,B
    RETURN
    140 PRINT I,H,G,A,B,C
              D   DID,E,F  RETURN
    140 PRINT D,E,F,A,B,C
    RETURN
    150 PRINT E,F,G,H
              DE  RETURN
    150 PRINT
              //  RETURN
    150 PRINT E,F,G,H
                  RA,B,C,D  RETURN
    150 PRINT E,F,A,B,C,D
    RETURN
    160 PRINT F,A,B,C,D,E
    //  RETURN

>LIST
     10 REM 5/21/84
     20 PRINT "BEGIN"
     30 INTEGER A,B
     40 INTEGER C,D
     50 INTEGER E,F
     60 READ A,B
     70 READ C,D
     80 READ E,F
     90 DATA 1,2,3,4,5,6,7,8,9
    100 PRINT "A,B,C,D,E,F"
    110 PRINT A,B,C,D,E,F
    120 PRINT B,C,D,E,F,A
    130 PRINT C,D,E,F,A,B
    140 PRINT D,E,F,A,B,C
    150 PRINT E,F,A,B,C,D
    160 PRINT F,A,B,C,D,E
    170 PRINT
```

```
         180 PRINT "END"
         999 END
     >
```

NAME Command

The NAME command names or renames the current program. The NAME command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program.

Syntax

```
     NAME [fname ]
```

Parameters

fname A name for the current program. *fname* is a valid MPE file name that conforms to MPE file name rules. If *fname* is not specified, the program has no name. Therefore, the NAME command can be used to delete a program's name. The SAVE and RESAVE commands use *fname* as the program name.

Examples

```
     NAME "Test1"          !The current program is called Test1
     NAME "File2.grp"      !The current program is called File2.grp
     NAME                  !The current program now has no name
```

REDO Command

The REDO command allows you to replace, delete, or insert characters in the last line that was accessed. The line may have been entered or it may have been accessed by any of the MODIFY, GET, LINK, MERGE, CHANGE, or REDO commands. The REDO command works exactly like the MODIFY command, except that it can modify a command as well as a program line. The REDO command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program.

Note that the GET command accesses each line of the program. Consequently, a REDO following a GET will display the last line that the GET has accessed.

Syntax

```
     REDO
```

Example

The following example shows the use of the REDO command to correct syntax errors.

```
     >20 INTGGER N,P,R  RETURN
     20 INTGGER N,P,R
```

```
           ^
Syntax error at character 12
Statement needs  =
>REDO  RETURN
20 INTGGER N,P,R
      E
20 INTEGER N,P,R
            Iumber  RETURN
20 INTEGER Number,P,R
RETURN
>LIST 20  RETURN
     20 INTEGER Number,P,R
>
```

RENUMBER Command

The RENUMBER command renumbers one range of program lines. The range can be the whole program. The RENUMBER command is a command-only statement. That is, it can only be issued at the interpreter prompt, and cannot be placed in a program.

Syntax

```
{RENUMBER}             {TO}
{RENUM   } [line_range ] {, } [beginning_line_number ] [BY increment ]
{REN     }
```

Parameters

line_range Lines to be renumbered. If you are specifying line numbers, use the line numbers that you had before issuing the RENUMBER command. The default is all program lines.

beginning_line_number New line number for the first line to be renumbered. The default is 10.

increment Increment. Number of lines between each renumbered line. The default is 10.

Secured lines remain secure when they are renumbered.

The RENUMBER command is not executed if it would rearrange lines. The lines that surround the lines that are being renumbered must still surround them after they are renumbered.

If a RENUMBER command would renumber a line with a number greater than 999999, then an error occurs and the command is not executed.

The RENUMBER command renumbers every line in the *line_range*. It also substitutes the new line number for the old one in every reference to an existing line (for example, if line 100 becomes line 350, the statement "GOTO 100" becomes "GOTO 350"). The RENUMBER command does not change line numbers that reference nonexistent lines.

Examples

```
>LIST
   100 GOTO 200
   120 RESTORE 190
   190 GOTO 110           !(Line 110 does not exist)
   200 DATA ABC
>RENUMBER
>LIST
    10 GOTO 40
    20 RESTORE 30
    30 GOTO 110            !(110 does not change)
    40 DATA ABC
>RENUM 20/40 TO 100 BY 5
>LIST
    10 GOTO 110
   100 RESTORE 105
   105 GOTO 110            !(110 does not change, but now exists)
   110 DATA ABC
>REN TO 500
>LIST
   500 GOTO 530
   510 RESTORE 520
   520 GOTO 530            !(110 becomes 530)
   530 DATA ABC
```

SCRATCH Statement

The SCRATCH statement can be used to reset variables to their default values, erase the current program, reset the values returned by functions, or reset the entire interpreter environment.

Syntax

```
        [ALL ]
        [PROG]
SCRATCH [COM ]
        [VARS]
```

Parameters

ALL Set the HP Business BASIC/XL interpreter environment to the same state as that on initial entry following the BBXL command.

COM Deallocates all variables. Also stops program execution if the SCRATCH statement is in a subunit.

PROG Erases the current program in the interpreter's work space. PROG is the default option set for SCRATCH.

VARS Deallocates all variables except those in common areas. Also stops program execution if the SCRATCH statement is in a subunit.

The options used with the SCRATCH statement permit you to select the level of features to be reset. Thus, SCRATCH VARS resets only those features listed below the heading in the following list. SCRATCH COM reinitializes the common area variables as well as resetting the features performed in a SCRATCH VARS. Likewise, SCRATCH PROG resets the SCRATCH PROG, SCRATCH COM and SCRATCH VARS features. Using SCRATCH ALL resets all the features indicated for each of the SCRATCH options and resets the interpreter back to its initial state. The following list summarizes the features reset with each option level:

SCRATCH ALL.

- Interpreter's Configuration File read and configurable options reset.
- FILES ARE IN reset to home group and account under which HP Business BASIC/XL is running.
- SEND OUTPUT TO reset to DISPLAY.
- SEND SYSTEM OUTPUT TO reset to DISPLAY.
- COPY ALL OUTPUT TO reset to DISPLAY.
- Output line width set to printer line width if HP Business BASIC/XL is running in batch mode.
- Output line width set to terminal line width if HP Business BASIC/XL is running interactively.
- FLUSH INPUT buffer.
- Branch-during-input keys cleared.
- Typing definitions of the programmable function keys restored.

SCRATCH COM.

- COMMON area variables reset according to INIT options selected.

SCRATCH PROG.

- Current program erased.
- Current program name erased.
- Current program lockword erased.
- DEFAULT option to assign values to variables on arithmetic overflow or underflow set OFF.
- Random seed number set to default value of PI/180.
- STANDARD numeric output format set.
- Angular units set to RADIANS.

- BREAK key enabled.

- Response function return value set to 0.

- CURKEY function return value set to 0.

- DBASE is reset.

- WORKFILE is reset.

- Form filename reset.

SCRATCH VARS.

- Non-COMMON area variables reset according to INIT option specified.

- Any Tracing is turned off.

- ERRL return value set to 0.

- ERRN return value set to 0.

- All open files that have been opened with the HP Business BASIC/XL ASSIGN statement that are not in a common area are closed.

SECURE Statement

The SECURE statement prevents program lines from being listed or modified.

Syntax

SECURE *line_range_list*

You cannot perform the following on a secured line:

- Modify.
- List (except for the line number, followed by an asterisk).
- Move or copy.

You can perform the following on a secured line:

- Delete.
- Renumber.

Example

The following shows the results of using the SECURE statement:

```
 >LIST
100 INTEGER A,B,C
110 LET A=1
120 LET B=2
```

```
130 LET C=3
140 PRINT A+B+C
999 END
>SECURE 110/130
>LIST
100 INTEGER A,B,C
110 *
120 *
130 *
140 PRINT A+B+C
999 END
>MODIFY 120
Line 120 secured and cannot be modified.
```

In the above example, lines 110 through 130 were secured.

XREF Command

XREF is an interpreter command that generates a cross reference of the entire current program, just the main program unit, or any procedure or function of the current program. A cross reference is a list of the identifiers in the specified part of the current program that includes the following information: name, type, class, and line numbers on which it is used. The cross reference is sorted according to identifier names.

Syntax :

```
                                     [WITH LIST  ]
XREF [sub_name  [,sub_name ]...] [WITH SOURCE] [TO listfile ]
```

Parameters

sub_name Either MAIN or the name of the procedure or function for which the cross reference is to be generated. A cross reference is generated from the entire program if this is not specified.

WITH LIST
WITH SOURCE If this parameter is specified, the cross reference immediately follows the listing of the source code for each individual part (MAIN, procedure, or function). If it is not specified, the identifiers from the main are listed under a banner containing the word MAIN, and identifiers from each procedure or function are listed under a one-line banner with the name of the procedure or function.

listfile The name of the file the cross reference is to be printed to. If not specified, the cross reference is printed to the destination specified by the SEND SYSTEM OUTPUT TO statement (usually the terminal).

The cross reference is generated on a subunit basis. The following information is provided for each identifier:

Name Name of the identifier.

Class Class to which the identifier belongs. Class information is designed to convey dimensionality and usage information. Dimensionality is specified by SIMPLE, identifiers declared implicitly or explicitly as scalars, and array identifiers declared implicitly or explicitly with the DIM statement as arrays. For numeric and string variables, usage is also characterized by the location of the declaration of the variable in the program, PARAMETER, or COMMON. If neither of those is specified, the variable is local. Identifiers that are not variables are characterized by their usage in the program as SUBPROGRAM, FUNCTION, EXTERNAL, or LABEL.

Type — The data type of the identifier: SHORT REAL, REAL, SHORT INTEGER, INTEGER, SHORT DECIMAL, DECIMAL, or STRING.

Declaration — Whether the numeric or string variable has been explicitly declared.

Occurrence — The line numbers of the statements that the identifier occurs in. The line numbers of statements in which a new value is potentially assigned to the identifier are followed by an (*). The line number of the statement that the identifier is declared in is followed by an ampersand (@).

If either WITH LIST or WITH SOURCE is specified, the formatting of the cross reference's output is controlled by any of the compiler options such as LINES, LIST, PAGE, PAGESUB, TITLE, and TITLESUB present in the program. Otherwise, the default compiler options are in effect.

NOTE Because of the large amount of internal information that must be evaluated when creating a cross-reference, a cross-reference for a large program can take a considerable amount of time. As a result, there may be a long delay before the first output is printed or displayed.

The following is a sample output of the cross reference:

```
PAGE 1  HP Business BASIC/XL   HP32715A.00.00   (c) Hewlett-Packard Co.
1989  MON, MAY 18, 1989  4:44 PM

* * * * * * * * * * * * * * * * * MAIN * * * * * * * * * * * * * * * * *

A                            SIMPLE              SHORT INTEGER
     250@    470*   1210*   1590*   1660    3420*   3430    3450   3550*
    3560    3590

A$                           SIMPLE              STRING
     230@   1250*   1260*   1270    1290    1520*   1530    1540   1610*
    1630    1650    1790*   1800    1820    2070*   2080    2100   2210*
    2220    2230*   2250    2290    2300    2570*   2710*   2760   3030*
    3060*   3070    3670    3680

B                            SIMPLE              SHORT INTEGER
     250@   3430

B                            ARRAY               SHORT INTEGER
     280@

B$                           SIMPLE              STRING
     230@   1760*   1780    2040*   2060

B1                           SIMPLE              SHORT INTEGER
```

```
     260@

B2                          SIMPLE              SHORT INTEGER

     260@

I1                          SIMPLE COMMON       SHORT INTEGER
     220@    1920*   1930    1940    2800*   2810    2820    2850*  2880
    2890     2910*   2920    2930    2940    2950    2970    3190*  3200
    3210     3280*   3290    3670*   3680    3750

Z                           SIMPLE COMMON       SHORT INTEGER
     220@    1270*   1280    1630*   1640    1800*   1810    2080*  2090
    2220*    2240    2760*   2770    3160*   3680    3700    3720
```

Examples

```
>XREF                            !Default parameters
>XREF TO Printer                 !Listing to Printer
>XREF WITH LIST                  !Cross reference will follow the source
                                 !for each program part
>XREF MAIN,Sub1,FNX TO Display   !Listing for selected program units,
                                 !to the terminal
>XREF SUB Sub2 WITH LIST         !Cross reference for Sub2
```

Program File Management

A program file is a file that contains an HP Business BASIC/XL program. The program is stored in a file in one of the following formats:

BASIC SAVE — A binary program file that contains a correct HP Business BASIC/XL program. It can be stored and retrieved more efficiently than an ASCII or BASIC DATA file. It does not have to be converted to ASCII or BASIC DATA format when it is stored, or have the syntax checked when it is retrieved. A program is to be compiled must be stored in this form.

> **NOTE** "Clean" BASIC SAVE files from time to time by saving the program in that file to an ASCII file from the interpreter, using GET to bring the ASCII file into the interpreter and then using RESAVE to store the file to the BASIC SAVE file. This is necessary because the interpreter does not do complete "garbage collection" when program lines are deleted or modified.

ASCII — An ASCII program file has fixed-length 80-byte records. Each program line is a series of one or more records. If a line exceeds the record length, the record ends with a continuation character (&) and the line continues in the next record. An ASCII file looks like the output of the LIST;NONAME command.

BASIC DATA   A BASIC DATA file has fixed length 128-word records.

In an ASCII or BASIC DATA file, a line that exceeds 500 characters is truncated and an asterisk is substituted for the 500th character. An error occurs when the line is accessed.

Table 2-5 explains the program file management commands and statements and shows which of them are compilable.

Table 2-5. File Management Commands and Statements

Statement or Command	**Compilable**	**File Type**	**Effect**
GET	No	BASIC SAVE	Replaces current program with program in specified disk file. Can execute program, starting at any line in the main program.
GET	No	ASCII BASIC DATA	Retrieves program from specified disk file, one line at a time. Syntaxes each line. Converts syntactically illegal lines to comments. Can replace all or part of current program. Can execute program, starting at any line in the main program. Renumbers new lines.
GET SUB	No	BASIC SAVE	Adds specified subunit(s) to current program.
LINK	No	ASCII BASIC DATA	Same as GET for ASCII and BASIC DATA files, except: cannot replace busy lines, execution must resume in program unit that executed LINK statement, does not affect variables.
MERGE	No	ASCII BASIC DATA	Same as LINK, except: replaces current line only if retrieved line has or receives same line number, execution resumes at line following MERGE statement.
RESAVE	No	Any	Stores current program in

			existing or new disk file.
RUN	No	Any	Executes current or specified program, beginning at first or specified line.
RUNONLY	Yes	BASIC SAVE	Protects interpreted HP Business BASIC/XL program from listing and modification, but allows execution.
SAVE	No	Any	Stores current program in new disk file.

The term *fname* appears in the syntax specifications of several file management commands and statements. For a description of *fname*, see "File Identification," in chapter 6.

GET Statement

The GET statement retrieves a program from a disk file and can execute it. The file type greatly affects the result of the GET statement, as stated in Table 2-6.

Syntax

GET *fname* [, *line_num*][; *execution_line*]

Parameters

fname The file name of the file containing the program that is to be retrieved. The file is in the BASIC SAVE, ASCII, or BASIC DATA format.

line_num Line number to be assigned to the first retrieved line.

execution_line Line identifier at which to begin execution subsequent to completing execution of the GET statement.

See Table 2-6 for more information on all of the above.

Table 2-6. How File Type Affects GET Statement

Affected Parameter	***fname* Is BASIC SAVE File**	***fname* Is ASCII Or BASIC DATA File**
fname	Program specified by *fname* replaces current program and its variables. Common blocks remain only if they are declared by the new program.	Program specified by *fname* is retrieved line by line. Each line is checked for correct syntax. Syntactically illegal lines are converted to comments. The GET stops if it retrieves a command (it does not execute the command).

line_num	Ignored.	Current program lines from *line_num* to the end of the program are deleted. Default for *line_num* is one. Lines before *line_num* are not affected. First retrieved line is renumbered *line_num*; if its old line number was *n*, then the number (*line_num* - *n*) is added to every line number in the retrieved lines (their own line numbers and the line numbers that they reference).
execution_line	If *execution_line* is specified, the new program begins executing at that line. The line must be in the main program. If *execution_line* is not specified, and a program executed the GET statement, then the new program begins executing at its first line. If *execution_line* is not specified, and the GET command was executed, control returns to the terminal.	If *execution_line* is specified, the resulting program begins executing at that line. The line must be in the main program.

Examples

The contents of Filea and Fileb are:

Filea (BASIC SAVE) **Fileb (ASCII)**

```
10 PRINT "Program A"                      10 PRINT "Program B"
 20 CALL A_sub                             20 CALL B_sub
 30 PRINT "END of Program A"               30 PRINT "End of Program B"
 40 STOP                                   40 STOP
100 SUB A_sub                             100 SUB B_sub
110   PRINT "In subprogram A_sub"         110   PRINT "In subprogram B_sub"
120 SUBEND                                120 SUBEND
```

Example 1

The following examples show the effect of the GET statement of an ASCII file while there is a program in the interpreter. Lines beginning with *line_num* are deleted from the first program.

```
 >GET "Filea"
>LIST
 ! Filea
    10 PRINT "Program A"
    20 CALL A_sub
    30 PRINT "End of Program A"
    40 STOP
   100 SUB A_sub
   110   PRINT "In subprogram A_sub"
   120 SUBEND
>GET "Fileb",40
>LIST
```

```
 ! FileA
    10 PRINT "Program A"
    20 CALL A_sub
    30 PRINT "End of Program A"
    40 PRINT "Program B"
    50 CALL B_sub
    60 PRINT "End of Program B"
    70 STOP
   130 SUB B_sub
   140   PRINT "In subprogram B_sub"
   150 SUBEND
>
```

Example 2

The following example shows the effect of a programmatic GET of an ASCII file. Because the GET specifies line 120, Program A is left intact, and Program B becomes part of A_sub.

```
 >GET "Filea"
>LIST
 ! Filea
    10 PRINT "Program A"
    20 CALL A_sub
    30 PRINT "End of Program A"
    40 STOP
   100 SUB A_sub
   110  PRINT "In subprogram A_sub"
   120  SUBEND
>15   GET "Fileb",120;20  !First line of Fileb is 120, execution &
 skips to 20
>16   PRINT "This line should be skipped."
>LIST
 ! Filea
    10 PRINT "Program A"
    15 GET "Fileb", 120;20  !First line of Fileb is 120, execution skips to 20
    16 PRINT "This line should be skipped."
    20 CALL A_sub
    30 PRINT "End of Program A"
    40 STOP
   100 SUB A_sub
   110   PRINT "In subprogram A_sub"
   120 SUBEND
>RUN
Program A
In subprogram A_sub
Program B
In subprogram B_sub
End of Program B
>LIST
 ! Filea
 10 PRINT "Program A"
 15 GET "Fileb", 120;20 !First line of Fileb is 120, execution skips to 20
 16 PRINT "This line should be skipped."
 20 CALL A_sub
 30 PRINT "End of Program A"
 40 STOP
100 SUB A_sub
110   PRINT "In subprogram  A_sub"
120 PRINT "Program B"
130 CALL B_sub
140 PRINT "End of Program B"
150 STOP
210 SUB B_sub
220   PRINT "In subprogram B_sub"
230 SUBEND
```

>

GET SUB Statement

The GET SUB statement retrieves specified subunits from a BASIC SAVE file and adds them to the current program. Current program lines are not affected. If the current program executed the GET SUB statement, execution continues at the line following the GET SUB statement.

The GET SUB statement retrieves subunits from BASIC SAVE files only. Use the MERGE statement to retrieve subunits from ASCII and BASIC DATA files.

Syntax

GET SUB *fname* [, *first_line* [,*increment*]] [; *first_sub* [, *last_sub*]]

Parameters

fname BASIC SAVE file containing subunits to be retrieved.

first_line A numeric literal that is the line number assigned to the first retrieved line. If not specified, the default value is the last line number + 1.

increment A numeric literal that is the increment used for renumbering the retrieved lines. If not specified, the default value for the increment is 10.

first_sub A numeric literal that is the number of the first subunit to retrieve from the *fname* file. The first subprogram or multi-line function in the *fname* file is designated number 1. If no value is specified, the value of *first_sub* is one. If the value of *first_sub* is greater than the highest numbered subunit in the *fname* file, then an error occurs.

last_sub A numeric literal that is the number of the last subunit to retrieve from the *fname* file. If *last_sub* is greater than the number of the last subunit in the *fname* file, then all subunits from *first_sub* through the last subunit in the *fname* file are retrieved. If no value is specified, the value is the highest numbered subunit in the *fname* file.

Examples

Consider a BASIC SAVE file named SUBFILE.

```
10 ! A file of subunits
20 SUB Subunit_1
30 SUBEND
40 SUB Subunit_2
50 SUBEND
60 SUB Subunit_3
70 SUBEND
80 SUB Subunit_4
90 SUBEND
```

The following program statement retrieves Subunit_1 through Subunit_4. Numbering of the first line of Subunit_1 begins at the highest line number in the current program + 1. The line numbers of subsequent lines are incremented by 10.

```
10 GET SUB "SUBFILE"

10 GET SUB "SUBFILE" ;1        !Retrieves Subunit_1 through Subunit_4
10 GET SUB "SUBFILE" ;2,4      !Retrieves Subunit_2 through Subunit_4
10 GET SUB "SUBFILE" ;3,3      !Retrieves Subunit_3
10 GET SUB "SUBFILE",100,10 ;3 !Retrieves Subunit_3 and Subunit_4
                               and begins numbering at line 100 with
                               lines subsequently incremented by 10.
```

LINK Statement

The LINK statement is identical to the GET statement for ASCII and BASIC DATA files, except in the following ways:

* Busy lines in the current program cannot be replaced (see "Busy Lines and Busy Subunits" for the definition of "busy").

* Execution must resume within the program unit that executed the LINK statement.

* The LINK statement does not affect local or common variables in the current program. Redeclarations are ignored.

Syntax

LINK *fname* [, *line_num*] [; *execution_line*]

Parameters

fname The filename of the file containing the program in the ASCII or BASIC DATA file format that is to be retrieved.

line_num Line number to be assigned to the first retrieved line. If this parameter is not specified, retrieved lines are not renumbered.

execution_line Line identifier that program resumes execution at after file is retrieved. Must belong to the program unit that executed the LINK statement. Default is the line following the LINK statement (or the line that replaced the LINK statement).

Execution does not resume after a LINK command unless *execution_line* is specified.

Examples

```
LINK "File1"            !File1 is retrieved
LINK "File2",200        !File2 is retrieved, the first line is 200
LINK "File3",300;330    !File3 is retrieved, the first line is 330,
                        ! the current program resumes at line 300
LINK "File4";150        !File4 is retrieved, the current program
                        !resumes at line 150
```

MERGE Statement

The MERGE statement is identical to the LINK statement, except in the following ways:

- A current line is replaced only if its line number belongs to a retrieved line. The retrieved line may have the same line number if it has been renumbered.
- If a subunit header (a SUB or DEF statement) is inserted anywhere except immediately before another subunit header, it becomes a comment and a warning is issued.
- If a subunit header is replaced by anything but another subunit header, an error occurs. Lines retrieved before the error occurred remain in the program.

Syntax

MERGE *fname* [, *line_num*][; *execution_line*]

Parameters

fname The filename of the file containing the program stored in the ASCII or BASIC DATA file to be retrieved.

line_num Line number to be assigned to the first retrieved line. Retrieved lines are not renumbered if this parameter is not specified.

execution_line Line identifier where execution resumes after the file is retrieved. The default is the line following the MERGE statement (or the line following the line that replaced the MERGE statement).

Execution does not resume after a MERGE command unless *execution_line* is specified.

Examples

```
MERGE "File1"          !File1 is retrieved
MERGE "File2",100      !File2 is retrieved, the first line is line 100
MERGE "File3",250;250  !File3 is retrieved, the first line is line 250
                       !and the current program resumes at line 250
MERGE "File4";600      !File4 is retrieved, and the current program
                       !resumes at line 600
```

RESAVE Statement

The RESAVE statement stores the current program in an existing or a new disk file. It can store a file in the ASCII, BASIC DATA, or BASIC SAVE formats.

Syntax

```
       [LIST ]
RESAVE [BDATA] [fname  [, line_range_list ][; NOMSG]]
```

If neither LIST nor BDATA is specified, the program is stored in the format of the existing file if the file already exists. If *fname* refers to a new file, and neither LIST nor BDATA is specified, the default is type BASIC SAVE.

Parameters

LIST
Stores program in ASCII format. If the file exists, it must be an ASCII file.

BDATA
Stores program in BASIC DATA format. If the file exists, it must be a BASIC DATA file.

fname
This specifies a new or existing file. RESAVE overwrites an existing file. If *fname* does not exist, RESAVE creates it and issues a warning that the file did not exist.

fname defaults to the file name of the current program as determined by the most recent GET or NAME command if this parameter is not specified.

line_range_ list
If the program is to be stored in ASCII or BASIC DATA format, *line_range_list* is as explained in "Specifying Line Ranges."

If the program is to be stored in program format, a program unit is the smallest unit that can be specified, and the main program is always stored, whether it is specified or not. If a *ln_spec1* is a line number or label, it must belong to the first line of a program unit; if *ln_spec2* is a line number or label, it must belong to the last line of a program unit. If *ln_spec1* and *ln_spec2* do not belong to the same program unit, all program units between them are stored.

NOMSG
Suppresses messages issued by RESAVE. The WARNINGS ON and WARNINGS OFF statements do not affect these messages.

Examples

```
RESAVE "File6"                     !File 6 is the same type as the existing file
RESAVE LIST "File7"                !File 7 is in ASCII format
RESAVE BDATA "File8"               !File 8 is a BASIC DATA file
RESAVE "File9",MAIN,5000/5999      !File 9 is a main program, lines 5000/5999
RESAVE "File10",Sub1/Sub6;NOMSG    !File 10 is Sub1/Sub6, and no
                                   !RESAVE messages are issued
```

RUN Command

The RUN Command executes the current program or retrieves a program from a disk file and executes it. Execution begins at the specified line or the first program line. The RUN command is a command-only statement. That is, it can only be issued at the interpreter prompt and can not be placed in a program.

Syntax

```
              [{,}         ]
RUN [fname ] [{;} line_id ] [; INFO=str_expr ]
```

Parameters

fname The name of the disk file containing the program to be retrieved and executed. This program replaces the program that is in the interpreter when the RUN command is issued. The current program in the interpreter's workspace is executed if *fname* is not specified. If *fname* specifies a nonexistent file, an error occurs, and the current program is not overwritten.

line_id The line number or line label in the main program at which to begin execution. The default is the first program line. If *line_id* is a line number that is not in the program, execution begins at the line with the next higher line number.

str_expr Assigns the value, a string, to be returned by a call to the INFO$ function. (See chapter 5).

The RUN command retrieves a disk file the way that the GET statement does, except that the RUN command removes all common declarations; the GET statement does not.

Before executing the program, the RUN command allocates space for variables, initializes them, and resets the following parameters in the interpreter:

Program execution	Stopped
Numeric output format	Standard
DEFAULT	OFF
Random number seed	PI/180
ERRL	0
ERRN	0
BREAK	Enabled
Automatic line numbering	Off
Options	Options specified in configuration file

Note that the RUN, HOP, and STEP commands stop any program that is executing when the command is entered. Then program execution begins as specified.

Example

```
RUN                           !Runs the current program
RUN "File1"                   !Runs program File1
RUN "File2";200               !Runs program File2, execution begins at line 200
RUN "File3",200               !Runs program File3, execution begins at line 200
RUN;1200                      !Runs the current program, execution begins at
                              !line 1200
RUN;INFO="TEST CODE"          !Runs the current program, the INFO$ function
                              !returns TEST CODE
RUN "File1"; INFO="Name"      !Runs program File1, the INFO$ function
                              !returns Name
RUN "File2";200;INFO="1985"   !Runs File2, execution begins at line 200,
                              !the INFO$ function returns 1985
RUN "File3",200;INFO="West"   !Runs File3, execution begins at line 200,
                              !the INFO$ function returns WEST
RUN;1200;INFO="yellow"        !Runs the current program, execution begins
                              !at line 1200, the INFO$ function
                              !returns yellow
```

Consider the execution of the following program sequence with and without the INFO option:

```
 1000 !
1010 IF INFO$="DEBUG" THEN
1020    PRINT "Just before assignment to Sum"
1030    PRINT "   Subtotal_1 = ";Subtotal_1
1040    PRINT "   Subtotal_2 = ";Subtotal_2
1050 ENDIF
1060 !
1070 Sum=Subtotal_1+Subtotal_2
1080 PRINT "The total is: ";Sum
>RUN
The total is:  1169.04
>RUN;INFO="DEBUG"
Just before assignment to Sum
  Subtotal_1 =  475.53
  Subtotal_2 =  693.51
The total is:  1169.04
```

RUNONLY Statement

The RUNONLY statement protects an interpreted HP Business BASIC/XL program from listing and modification, but allows its execution. Run-only status cannot be reversed. Once a program is protected with the RUNONLY statement, it can only be run with the GET statement.

Syntax

```
RUNONLY fname
```

Parameters

fname The file name of the file that will be protected. This file must be a BASIC SAVE file.

When a run-only program is retrieved, it begins executing at its first line unless another run-only program retrieved it and specified a starting line.

Example

```
RUNONLY "File1"          !Protects File1
RUNONLY "File2.lab"      !Protects File2 in the group lab
```

SAVE Statement

The SAVE statement stores the current program in the interpreter's work space in a new disk file. The SAVE statement can store a file in any format.

Syntax

```
     [LIST ]
SAVE [BDATA] [fname  [, line_range_list ][; NOMSG]]
```

If neither LIST nor BASIC DATA is specified, the program is stored in the BASIC SAVE format.

Parameters

LIST — Stores program in ASCII format.

BDATA — Stores program in BASIC DATA format.

fname — This must specify a new file.

fname defaults to the file name of the current program as determined by the most recent GET or NAME command if this parameter is not specified.

line_range_ list — If the program is to be stored in ASCII or BASIC DATA format, *line_range_list* has the syntax explained in "Specifying Line Ranges" earlier in this chapter.

If the program is to be stored in program format, a program unit is the smallest unit that can be specified, and the main program is always stored, whether it is specified or not. If *ln_spec1* is a line number or label, it must belong to the first line of a program unit; if *ln_spec2* is a line number or label, it must belong to the last line of a program unit. If *ln_spec1* and *ln_spec2* do not belong to the same program unit, all program units between them are stored.

NOMSG — Suppresses messages issued by SAVE. The WARNINGS ON and WARNINGS OFF statements do not affect these messages.

Examples

```
     SAVE "File1"                    !Saves File1 as type BASIC SAVE
     SAVE LIST "File2"               !Saves File2 as type ASCII
     SAVE BDATA "File3"              !Saves File3 as type BASIC DATA
     SAVE "File4",MAIN,5000/5999     !Saves File4 as a main program, with
                                     !lines 5000/5999
     SAVE "File5",Sub1/Sub6;NOMSG    !Saves File 5 as Sub1/Sub6, any SAVE
                                     !messages are suppressed.
```

Program Debugging

Two HP Business BASIC/XL features make debugging the current program easier: trace statements and suspension during execution. Trace statements print messages when one line transfers control to another that is not sequentially the next line in the program, or when a variable is assigned a value.

The program is suspended when one of the following occurs:

* The program executes a PAUSE statement.

* You press CONTROL Y (when no ON HALT is active).

* You press CONTROL Y twice in rapid succession.

* An error occurs (and error-handling is not active).

When program execution is suspended, control returns to the terminal keyboard. From the keyboard, you can do any of the following:

* Variable values can be displayed (type the variable name and press RETURN).

* Commands can be executed.

* Variable values can be changed (with the LET command).

* Program lines can be modified (with the MODIFY command).

* Program lines can be inserted.

* Program lines can be deleted (with the DELETE command or as explained in "Creating and Modifying a Program").

* Control can be transferred to another part of the program (with a GOTO, GOSUB, or CALL command).

* Program lines can be added (as explained in "Creating and Modifying a Program").

A busy line or subunit cannot be modified or deleted when the program is suspended. See "Busy Lines and Busy Subunits" for more information.

Table 2-7 lists the debugging statements and commands and their effects. All of the debugging statements and commands affect the current program. None of the debugging commands are compilable.

Table 2-7. Program Debugging Commands

Command	**Command or Statement?**	**Effect**
CALLS	Command	Prints names of busy program units on system printer.
CONTINUE	Command	Restarts suspended program.
FILES	Command	Prints names and numbers of open files on system printer.
HOP	Command	Executes program and suspends it at next line that is in same program unit and not in a loop.
PAUSE	Statement	Suspends program execution.*

STEP	Command	Executes next line of suspended program and suspends program at line following it.
Trace Statements	Either	See Table 2-8.
Untrace Statements	Either	See Table 2-9.

Table 2-7 Note

* The PAUSE statement is defined in chapter 4.

Busy Lines and Busy Subunits

A line is *busy* if one of the following is true:

- The line made a call that has not returned.
- The line was interrupted with the halt key before it finished executing. (Not all lines can be interrupted in this way. A PRINT statement is an example of a line that cannot be busy.)

A busy line cannot be modified or deleted.

A subunit is *busy* if it has been called, but has not returned.

A busy subunit can be modified, but not deleted. When modifying a subunit, observe the following restrictions:

- A busy SUB statement can only be changed to another SUB statement.
- A busy DEF statement can only be changed to another DEF statement. Numeric type variables can only be changed to another numeric type. The type cannot be changed from numeric to string or vice versa.
- Changes to the subunit header take effect the next time the subunit is called.

To make other header changes to a busy DEF or SUB statement, you must first stop the program with the STOP command (chapter 4 explains the STOP command).

CALLS Command

The CALLS command prints the names of busy program units on the system printer or another device. (The SEND SYSTEM OUTPUT TO statement specifies the device; see chapter 6.) The CALLS command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program.

Syntax

```
CALLS
```

Example

If the program is not running:

```
>CALLS
 MAIN   Not executing.
```

If the program has paused at line 10:

```
>CALLS
 MAIN @ 10
```

Suppose that the following are true:

Line 10 of the main program calls subunit FNBeep$.
Line 40 of FNBeep$ calls subunit FNBeep.
Line 50 of FNBeep calls subunit B.
The program pauses at line 100 in subunit B.

Then:

```
>CALLS
 SUB B @ 100
 FNBeep @ 50
 FNBeep$ @ 40
 MAIN @ 10
```

CONTINUE Command

The CONTINUE command restarts a suspended program. The CONTINUE command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program.

Syntax

```
{CONTINUE}[line_id ]
{CONT    }[*      ]
```

Parameters

line_id Line that program execution will restart at. The line must belong to the program unit that was executing when the program was suspended. An error occurs if *line_id* is not in the program.

* Restarts the program at the last line executed.

If neither *line_id* nor * is specified, the CONTINUE command restarts program execution at the next line to be executed.

An error occurs if the CONTINUE command is executed and there is no current program in the work space.

Examples

The following shows examples of the CONTINUE command:

```
CONTINUE
CONT
CONTINUE 100     !Continues the program at line 100
CONT Label5      !Continues the program at the line number in Label5
CONT *           !Continues the program at the last line executed
```

FILES Command

The FILES command prints the file numbers of the files that have been declared in the currently executing subunit. If a file is open, the FILES command prints the file name after the number. The FILES command prints a message if a file is not open. The FILES command also prints COMMON after each common file and PARAMETER after each file that was passed to the subunit as a parameter. The FILES command prints its information on the system printer. The FILES command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program.

Syntax

```
FILES
```

Examples

The following shows an example of the FILES command:

```
>FILES
#1: File is not currently open.
#2: MYFILE.MYGROUP.MYACCT, REC:1, WRD:1
#3: MYFILE1.MYGROUP.MYACCT, REC:1, WRD:1, PARAMETER
#4: MYFILE2.MYGROUP.MYACCT, REC:1, WRD:1, COMMON
```

If no files are declared in the currently executing subunit the FILES command will return a message.

```
>FILES
 No files are declared in the current subprogram.
```

HOP Command

The HOP command can single-step from one line of a program unit to the next line of the same program unit, without suspending execution during loops or subunits. Specifically, the HOP command does the following:

- Does a TRACE PAUSE on the line that follows the next line to be executed (even if it is in another program unit).

* Does a CONTINUE.

The HOP command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program.

Syntax

HOP

The HOP command is useful for the following:

* Hopping through a GOSUB or CALL.
* Hopping through a loop (when executed on the last line of the loop).

The breakpoint that the HOP statement sets is reset by the next HOP statement (only one HOP breakpoint per program).

Example

The following shows an example of the HOP command. The program pauses at line 110. The HOP command has been issued during that pause.

```
>LIST
 !  exam246
    100 LET A=3
    105 PRINT "HI"
    110 PAUSE
    120 PRINT A
    130 PRINT A+A
    140 PRINT A*A
    150 PRINT "BYE"
    999 END
>RUN
HI
>HOP
 3
      130 PRINT A+A
>HOP
 6
      140 PRINT A*A
>hop
 9
      150 PRINT "BYE"
>HOP
BYE
      999 END
```

STEP Command

The STEP command--or several STEP commands--can single-step through a suspended program. Specifically, the STEP command does the following:

1. Executes the line that the program is suspended at.

2. Displays the next line to be executed.

3. Suspends the program at the displayed line.

The STEP command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program.

Pressing CONTROL E also issues the STEP command.

Syntax

STEP

Examples

The following shows an example of the STEP command. The program pauses at line 110, and the STEP command is issued during that pause.

```
>LIST
   100 LET A=3
   105 PRINT "HI"
   110 PAUSE
   120 PRINT A
   130 PRINT A+A
   140 PRINT A*A
   150 PRINT "BYE"
   999 END
>RUN
HI
>STEP
3
   130 PRINT A+A
>STEP
6
   140 PRINT A*A
>STEP
9
   150 PRINT "BYE"
>CONT
BYE
>
```

Trace and Untrace Statements

Trace statements trace lines, variables, or both. Untrace statements cancel trace statements.

A trace statement, while tracing lines, prints the following message whenever one line transfers control to another:

TRACE IN LINE *line_num1*; BRANCH TO *line_num2*

A trace statement, while tracing variables, prints the following messages whenever variables change value:

For a scalar variable:

TRACE IN LINE *line_num*; *var_name* = *new_value*

For an entire array variable:

TRACE IN LINE *line_num*; ARRAY *var_name* IS CHANGED

For an array element:

TRACE IN LINE *line_num*; *array_name* (*Subscript_of_element*) = *new_value*
TRACE IN LINE *line_num*; ELEMENT *n* IN ARRAY *var_name* = *new_value*

Trace statements print their output on the system printer. (The system printer is specified by the SEND SYSTEM OUTPUT TO statement. The default is the standard list device, that is, the terminal if HP Business BASIC/XL is running interactively.)

Every trace and untrace statement can also be a command.

Table 2-8 shows which trace statements trace lines, which trace variables and how the trace statements differ. For details about a particular trace statement, see the section about that statement.

Table 2-8. Trace Statements

Statement	**Traces lines**	**Traces variables**
TRACE ALL	Throughout program.	Throughout program.
TRACE EXEC[UTION]	From execution of first specified line through execution of last specified line.	No.
TRACE EXEC[UTION] VAR[S]	No.	From execution of first specified line through execution of last specified line.
TRACE LINES	Within specified range.	No.
TRACE PAUSE	Within specified range and pauses before each line is executed.	No.
TRACE VAR[S]	No.	As specified.
TRACE VAR[S] IN	No.	In specified program units or lines.
TRACE WAIT	No.	No.

Table 2-9 matches each trace statement with the untrace statements that partially or totally cancel it.

Table 2-9. Trace/Untrace Statement Correspondence

Trace Statement	Untrace Statements That Cancel It
TRACE ALL	UNTRACE LINES UNTRACE VAR[S] UNTRACE VAR[S] IN UNTRACE ALL TRACE OFF
TRACE EXEC[UTION]	UNTRACE EXEC[UTION] TRACE OFF
TRACE EXEC[UTION] VAR[S]	UNTRACE EXEC[UTION] VAR[S] TRACE OFF
TRACE LINES	UNTRACE LINES UNTRACE ALL TRACE OFF
TRACE PAUSE	UNTRACE PAUSE TRACE OFF
TRACE VAR[S]	UNTRACE VAR[S] UNTRACE ALL TRACE OFF
TRACE VAR[S] IN	UNTRACE VAR[S] IN UNTRACE VAR[S] UNTRACE ALL TRACE OFF
TRACE WAIT	TRACE OFF TRACE WAIT *n* where *n* <0

TRACE ALL and UNTRACE ALL

The TRACE ALL statement is the equivalent of the TRACE LINES and TRACE VARS statements. It traces lines and variables throughout the program.

The UNTRACE ALL statement cancels TRACE ALL.

Syntax

TRACE ALL

UNTRACE ALL

TRACE EXEC and UNTRACE EXEC

The TRACE EXEC statement traces lines, beginning when the first specified line executes and ending when the last specified line executes. If lines are not specified, TRACE EXEC applies to the entire program. If the first specified line does not exist or is not executed, TRACE EXEC does not trace lines. If the last specified line does not exist or is not executed, the TRACE EXEC statement does not stop tracing lines (unless an UNTRACE EXEC or TRACE OFF statement executes).

The UNTRACE EXEC statement cancels the TRACE EXEC statement (for every line).

Syntax

```
      {EXECUTION}
TRACE {EXEC     } [line_id1  [TO line_id2 ]]

        {EXECUTION}
UNTRACE {EXEC     }
```

Parameters

line_id1 Line tracing begins when this line executes. If this line does not execute, line tracing never begins. Default is the first program line.

line_id2 Line tracing ends when this line executes. If this line is not specified or does not execute, line tracing does not end until an UNTRACE EXEC or a TRACE OFF statement executes.

TRACE EXEC VARS and UNTRACE EXEC VARS

The TRACE EXEC VARS statement traces variables, beginning when the first specified line executes and ending when the last specified line executes. If lines are not specified, TRACE EXEC VARS applies to the entire program. If the first specified line does not exist or is not executed, TRACE EXEC VARS does not trace variables. If the last specified line does not exist or is not executed, the TRACE EXEC VARS statement does not stop tracing variables (unless an UNTRACE EXEC or TRACE OFF statement executes).

The UNTRACE EXEC VARS statement cancels TRACE EXEC VARS (for every line).

Syntax

```
      {EXECUTION} {VARS}
TRACE {EXEC     } {VAR } [line_id1  [TO line_id2 ]]

        {EXECUTION}{VARS}
```

UNTRACE {EXEC }{VAR }

Parameters

line_id1	Variable tracing begins when this line executes. If this line does not execute, variable tracing never begins. The default is the first program line.
line_id2	Variable tracing ends when this line executes. If this line is not specified or does not execute, variable tracing does not end until an UNTRACE EXEC VARS or an TRACE OFF statement executes.

TRACE LINES and UNTRACE LINES

The TRACE LINES statement traces specified lines.

The UNTRACE LINES statement cancels TRACE LINES for specified lines (not necessarily for every line that TRACE LINES traces).

Syntax

TRACE LINES [*line_range_list1*]

UNTRACE LINES [*line_range_list2*]

Parameters

line_range_ list1	Lines to be traced. The default is all program lines.
line_range_ list2	Lines that TRACE LINES is to be canceled for (can be a subset of *line_range_list1*). The default is all program lines.

TRACE PAUSE and UNTRACE PAUSE

The TRACE PAUSE statement traces specified lines exactly as TRACE LINES does. It also suspends the program like the PAUSE statement does before the lines are executed. The CONTINUE command causes the suspended program to resume execution.

The UNTRACE PAUSE statement cancels TRACE PAUSE for specified lines (not necessarily for every line that TRACE PAUSE traces). If another trace statement traces those lines, that statement continues to trace them, but TRACE PAUSE does not delay the program after the trace messages that are associated with those lines.

Syntax

TRACE PAUSE [*line_range_list1*]

UNTRACE PAUSE [*line_range_list2*]

Parameters

line_range_list1 Lines to be traced with pause. The default is all program lines.

line_range_list2 Lines for which TRACE PAUSE is to be canceled (can be a subset of *line_range_list1*). The default is all program lines.

TRACE VARS and UNTRACE VARS

The TRACE VARS statement traces specified variables.

The UNTRACE VARS statement cancels TRACE VARS for specified variables (not necessarily for every variable that TRACE VARS traces).

Syntax

```
      {VARS}
TRACE {VAR } [var_name1  [, var_name2 ]...]

        {VARS}
UNTRACE {VAR } [var_name3  [, var_name4 ]...]
```

Parameters

var_name1 *var_name1* specifies variable to be traced. The default is all variables in the program.

var_name2 Each *var_name2* specifies an additional variable to be traced.

var_name3 *var_name3* specifies a variable that TRACE VARS is to be canceled for. The default is all variables in the program.

var_name4 Each *var_name4* specifies an additional variable that TRACE VARS is to be canceled for.

TRACE VARS IN and UNTRACE VARS IN

The TRACE VARS IN statement traces all variables in one or more specified subunits.

The UNTRACE VARS IN statement cancels TRACE VARS IN for specified subunits (not necessarily for every subunit that TRACE VARS IN specified).

Syntax

```
      {VARS}
TRACE {VAR } IN sub_id1  [, sub_id3 ]...

        {VARS}
UNTRACE {VAR } IN sub_id2  [, sub_id4 ]...
```

Parameters

sub_id1 — *sub_id1* specifies a subunit that variables will be traced in. *sub_id1* is specified by [SUB] *sub_id* or MAIN.

sub_id3 — Each *sub_id3* specifies an additional subunit that variables will be traced in. Each *sub_id3* is specified by [SUB] *sub_id* or MAIN.

sub_id2 — *sub_id2* specifies a subunit that variables will no longer be traced in. *sub_id* is specified by [SUB] *sub_id* or MAIN.

sub_id4 — Each *sub_id4* specifies an addition subunit that variables will no longer be traced in. Each *sub_id4* is specified by [SUB] *sub_id* or MAIN.

The UNTRACE ALL, UNTRACE VARS, and TRACE OFF statements also cancel the TRACE VARS IN statement.

Example

The following shows examples of the TRACE VARS IN command. The first example is in a program, the last two are issued as commands.

```
10 TRACE VARS IN MAIN, SUB A
TRACE VARS IN Sub_a, Sub_b, FNX
UNTRACE VARS IN X,Y,Z
```

TRACE WAIT

The TRACE WAIT statement delays the program for a specified time after each trace message (for line tracing and variable tracing).

Syntax

TRACE WAIT *num_expr*

Parameters

num_expr — Number of seconds to delay the program after each trace message. The value of *num_expr* must be in the range [-32768, 32767]. If *num_expr* < 0, TRACE WAIT does not delay the program after trace messages.

TRACE OFF

The TRACE OFF statement cancels every TRACE statement.

Syntax

TRACE OFF

OPTION TRACE and OPTION NOTRACE

The OPTION TRACE statement enables the trace statements in the program unit that contains it. The OPTION NOTRACE statement disables the trace statements in the program unit that contains it. If a program unit contains neither an OPTION TRACE nor an OPTION NOTRACE statement, the global option applies (its default is GLOBAL OPTION TRACE).

Syntax

```
OPTION TRACE

       {NO TRACE}
OPTION {NOTRACE }
```

The OPTION TRACE and OPTION NOTRACE statements can appear anywhere in a program unit. The highest-numbered TRACE statement affects the entire program unit the entire time that the program unit is executing.

Example

```
     100 TRACE LINES
     110 READ A,B
     120 IF A=B THEN GOTO 140
     130 PRINT "A<>B"
     140 PRINT "A=B"
             .
             .
             .
     200 OPTION TRACE
             .
             .
             .
     300 OPTION NOTRACE
     999 END
```

In the above program, the OPTION NOTRACE statement at line 300 disables the trace statement at line 100, despite the OPTION TRACE statement at line 200. HP Business BASIC/XL does not trace the branch from line 120 to line 140 when the program runs.

The Program Analyst

The interpreter maintains extensive data structures that describe the current program. That information can be valuable for developing and maintaining programs. The Program Analyst is an environment that makes this information available and provides tools for analyzing the information. The Program Analyst can be used for design optimization, memory usage analysis, program statistics information, and optimization of subunit sizes.

ANALYST

The ANALYST command enters the Program Analyst environment. The following conditions must be met to successfully run the Program Analyst:

- The terminal fully supports the terminal-specific features of HP Business BASIC/XL.
- The interpreter is running in a session, rather than a batch job.
- There is at least one program line in the current program.
- The program is not running or paused.
- The program has no VERIFY errors.
- The destination for OUTPUT and SYSTEM OUTPUT is the display.

Syntax

ANALYST [*screen_argument*]

Parameters

screen_argument An argument that specifies which screen to enter. If no screen is specified, the Main Menu/Browse screen is displayed.

Table 2-10 lists each argument, and the screen that it displays.

Table 2-10. Analyst Command Arguments

Argument	Screen Selected
S	Static Analysis.
O	Optimize.
D	Data Types.
C	Suggest COPTIONs.
G	Replace GOSUBs.
P	Optimize PACKFMTs.
E	Extract Subunit.

This section describes each screen. Each action that occurs in a particular screen is explained. The following control actions work from any screen:

- To exit the Program Analyst press f8.
- Pressing HALT while the Program Analyst is waiting for input transfers control to the next height level screen.
- Pressing HALT while the Program Analyst is writing information to the screen returns control to the Main Menu/Browse screen.
- Pressing HALT while in the Main Menu/Browse screen exits the Program Analyst.

All files created by the Program Analyst have a set of comments giving the file name, the date and time of creation, the name of the screen being used, and the original name of the program as their first few lines.

The user interface capabilities used in the Program Analyst are available in HP Business BASIC/XL applications through the following statements and functions:

ON KEY
ON HALT
CURSOR
RESPONSE
ACCEPT
TINPUT

NOTE The features of the Program Analyst can change from one release to the next. Whenever you receive a new version of HP Business BASIC/XL, check the NEWS category in the HELP facility for information on changes and enhancements.

In some screens, the Program Analyst creates file whose names have the form BBPA*nn*. After you have merged one of these files, purge it. If you have many of these files in your group, the Program Analyst spends extra time trying to find an unused name.

The Main Menu/Browse Screen. The purpose of this screen is to provide general information about the current program.

The Main Menu/Browse Screen displays the following information about the current program:

* The name of the program (the BSAVE file name).
* The time and date when the program was last saved.
* The overall subunit space and fixed data space requirements of the program.
* The subunit space and fixed data space requirement of each subunit.
* The source code listing, shown in blocks of eight lines.

The cursor is on the first character of the current subunit name. Table 2-11 shows the actions that you can perform while in the Main Menu/Browse screen.

Table 2-11. Main Menu/Browse Screen Actions

Action	**Effect**
Softkeys	Pressing one of the labeled softkeys moves between adjacent subunits or to a different screen.
Line number	Entering a line number moves to that line. If the line exists, it is the first line displayed, and the subunit containing it is the

	current subunit.
Subunit Name	Entering a subunit name moves to that subunit. If the name entered matches a subunit in the program, that subunit is the current subunit and its first eight lines are displayed.
RETURN	Pressing RETURN displays the next eight lines of the current subunit. This method can only be used in the current subunit, and cannot be used to move to the next subunit.

The Static Analysis Screen. The purpose of this screen is to provide detailed statistics about the program and its system requirements.

The Static Analysis screen displays detailed information about each subunit of a program. It contains three types of information. The following lists describe the information that this screen provides.

Interpreter Resource Utilization: This section contains information about resource utilization within the interpreter. Resources include tables, data segments, and interpreter space. The fields and their contents are:

Field	**Contents**
Global Tables	The amount of space (in words) taken to store the interpreter's directory information for locating and managing all of the programs subunits. Included are tables holding the names of the subunits.
Local Tables	For the subunit being displayed, this value is the amount of space for all tables that reside in the subunit space area of the interpreter's data stack.
New Run-time Tables	When a program contains references to external routines or intrinsics, the interpreter requires additional space for parameter information at run-time. This value is an estimate of that space requirement.
Recoverable	The interpreter's tables can sometimes become cluttered with unnecessary information, particularly when extensive editing has been done. The tables can be cleaned by saving the program in ASCII format (through the SAVE LIST command) and then issuing a GET command. The value displayed is an estimate of the number of words in the subunit space that would be recovered by a SAVE LIST and GET.
COMMON Space	The spaced required by all variables declared in COM statements.
Local Space	Space for locally declared (or undeclared) variables and parameters. This number is broken down into Numeric Space and String Space fields.

Software Metrics This section provides statistics about the program. Many techniques exist for measuring the size and complexity of software and the amount of structure it contains. The Program Analyst can measure these features quickly and accurately. The significance of the numbers is left up to the programmer. The fact that the numbers are generated should not be considered an endorsement of a particular technique, nor is
it the intent to pass judgement on the user's code. This section contains the following information:

Field	**Contents**
Source Lines	The number of lines in the subunit. This is the number of unique line numbers. Continuation lines are not counted.
Comments	The total number of comments including REM statements, comment lines, and comments placed at the end of other program lines.
NCSS	Non-Comment Source Statements. The number of program lines that do not consist entirely of a comment or REM.
Code Volume (Halstead)	An attempt to quantify the information content of a group of source statements, in this case a subunit or program. Code volume is a number calculated from the number of unique identifiers and operators and the number of occurrences of these identifiers and operators.
Complexity (McCabe)	This value is the number of decision points in the subunit or program, plus one.
Structure Compliance (DeMarco)	Indicates the percentage of branches that are accomplished without explicit GOTO statements. All of the structured statements are counted as branches, including each value or range in a CASE statement.

Statement Frequency The Statement Frequency section lists the 21 most frequently used statements in the subunit. The statement names are displayed with the frequency count.

The Optimize Screens. The Program Analyst contains four screens that are specifically designed to help improve the efficiency of the current program. The next four sections describe each of these screens.

The Data Types Screen. The purpose of this screen is to provide information to minimize run-time conversions through more efficient definitions of variable data types.

This screen displays information for the entire MAIN subunit. The name of the subunit and the first and last line numbers are displayed at the top of the screen. A predicted number of conversions is below the subunit information is shown in a table. The lines with the most conversions are indicated to the right of the table. The Program Analyst predicts static conversion numbers. If a statement (such as a FOR statement) is to be executed multiple times, the Program Analyst counts the conversion only once. If the Program Analyst finds a control variable that is a floating-point type, or a default REAL or DECIMAL type and the Program Analyst can determine that the starting value, limit, and step (if present) are all integers, the line number of the FOR loop is reported. A FOR loop that meets those criteria is very inefficient because the control variable has to be converted. In addition, the FOR loop can produce incorrect results.

After the information about the MAIN subunit is display, the cursor is at the beginning of the Line Range field at the top of the screen. To move to a different part of the program, enter a line range or a single line number, or press a softkey to move to the previous or next subunit. If you select a single line, the line itself is listed at the bottom of the screen. The data type of each numeric variable is displayed.

The Program Analyst cannot accurately detect conversions that occur during a call to an external routine, an intrinsic, a subprogram, or a function. Conversions may be generated if an actual parameter does not match the declared parameter. Also, any conversions that take place to determine which CASE to execute in a SELECT structure are not reported.

If your program uses GLOBAL OPTION DECIMAL and does not have OPTION DECIMAL or OPTION REAL statements in each of the subunits, issue the VERIFY command before entering the Program Analyst . This notifies all the subunits that they will be executing in DECIMAL mode and ensures that you get the most accurate information about conversions.

The Suggest COPTIONs Screen. The purpose of this screen is to predict the benefit of each of the compiler options (COPTIONs) in terms of generated code savings.

The Suggest COPTIONs screen estimates the number of words of compiled code that would be eliminated through the use of each compiler option. The information is displayed in four columns. The first column contains the names of all the compiler options. If the subunit or program currently being displayed has a compiler option in effect, the code savings is displayed in the Current column. If the program or subunit is not current taking advantage of that option, the potential savings is displayed in the Potential column. The final column indicates whether any features that would prevent the use of an option are being used. The Program Analyst cannot estimate the actual size of a compiled subunit.

When you enter the Suggest COPTIONs screen, the cursor is positioned under the first letter in the subunit or program name. You can type a subunit name or line number or press a softkey to move to another subunit or line.

The Replace GOSUBs Screen. The purpose of this screen is to replace GOSUB statement with the subroutines the reference. This improves performance in some situations.

If a program has GOSUBs that execute many thousands of times, the overhead associated with them becomes noticeable. The Replace GOSUBs screen can, with certain restrictions, be used to replace a GOSUB with the body of the referenced subroutine. This is possible when there are enough available line numbers after the GOSUB to insert the entire subroutine before the next line, and the subroutine does not contain and GOTO statements or any lines that are targets of branches.

The Replace GOSUBs screen prompts you for a line range to use to search for eligible GOSUBs. It also asks for the maximum number of lines to be inserted for each GOSUB. The Program Analyst then creates an ASCII file containing a copy of the referenced subroutine. If you choose to replace the GOSUB with the subroutine, the GOSUB statement is deleted and the copy of the subroutine is inserted in its place. If the subroutine contains consecutive assignment statements, the Program Analyst combines them into fewer lines if possible.

The existence of this screen should not suggest that all GOSUBs should be eliminated. Only those that are used very frequently should be considered for replacement.

The Replace GOSUBs screen is useful in conjunction with the Extract Subunit screen described later. Replace GOSUBs can help to resolve subroutine references that might otherwise prevent a successful extraction.

The Optimize PACKFMTs Screen. The purpose of this screen is to generate SKIP clauses for PACKFMT statement to minimize packing and unpacking of variables.

Often a subunit needs to access only a few items specified in a PACKFMT statement. In a PACK or UNPACK statement, the entire record will be transferred between the string buffer and the variable, even if the PACKFMT statement has information for many more items than the subunit is using.

To improve efficiency, the PACKFMT specification can contain one or more SKIP clauses. A SKIP specifies that a certain number of bytes in the string buffer are to be ignored during an UNPACK and skipped over during a PACK.

The Optimize PACKFMTs screen determines which items can be skipped, calculates the number of bytes to skip, and then modifies a PACKFMT accordingly.

When you enter this screen, you are prompted for the line number of a PACKFMT statement. You can enter a line number or use a softkey to list the line numbers of all of the PACKFMT statements in your program. If the correct line number of a PACKFMT is entered, each item in the PACKFMT statement is examined. If the PACKFMT statement is used anywhere else in the subunit, or if it is in common or is a parameter to the subunit, the Program Analyst assumes that it is needed and cannot be skipped. Otherwise, the Program Analyst determines the number of bytes in that item and generates a SKIP. If adjacent items can be skipped, the Program Analyst creates only one SKIP clause to cover those items. The items that can be skipped are highlighted.

If skippable items are found, pressing RETURN will cause the Program Analyst to create an ASCII file containing the following statements:

- The original PACKFMT line with the actual statement commented out with a !.
- The new PACKFMT statement with a line number one greater than the original.

When this file is merged (using the MERGE command) back into the program, the original line serves as documentation of the complete record layout. If you are referencing a PACKFMT statement by a line number instead of a label, you need to modify the appropriate PACK or UNPACK statement.

When determining the number of bytes in a string variable, the Program Analyst uses the maximum declared length (or 18 if the string is undeclared). The Program Analyst assumes that strings are always padded out to their maximum length before packing them. This is recommended practice because the maximum length is always used during an UNPACK.

The Optimize PACKFMTs screen can also be used to optimize IN DATASET statements. You can modify a IN DATASET statement so that it looks like a PACKFMT statement, optimize the modified statement, merge the statement
into the program, and then modify it back to an IN DATASET statement.

The Extract Subunit Screen. The purpose of this screen is to assist in dividing large subunits into smaller subunits.

Manually removing lines form one part of a program and using the lines to create a new subunit can be tedious. There are many dependencies that must be examined. The Program Analyst can be very useful for extracting subunits. Using the information in the interpreter, the Program Analyst can detect all of the following dependencies:

- Branches that will be broken if lines are removed.
- Variables that have been shared between the old and new subunits that must become parameters or placed in common.
- Single-line functions definitions and OPTION, IMAGE, and PACKFMT statements that will be needed in the new subunit.

On entry of a line range, the Program Analyst displays the effects of removing this line range from its subunit. The Program Analyst examines all GOTOs, GOSUBs, and structured statements to determine whether they would be affected. It also determines whether variables (including files) need to be passed as parameters to the proposed new subunit. Once you have identified a line range that can be extracted without breaking any branches or structures, the Program Analyst creates the new subunit and the CALL to it.

NOTE The Program Analyst allows you to extract a subunit even if there are broken branches. Manual editing may be required before the new program can run.

Factors Affecting Extraction. The following factors that can affect extraction cannot be fully analyzed:

* DATA, READ, and RESTORE statements.
* Report Writer usage.
* SORT USING statements.
* THREAD statements.
* RETURN statements.
* ON ERROR, ON KEY, ON HALT, etc.

The Program Analyst may produce warnings if these features are used in the subunit being divided. However, the Program Analyst continues to create the subunit. If will be up to you to make any necessary manual changes.

When the Program Analyst creates a new subunit, it handles the following requirements automatically:

* Creation of parameter lists.
* Copying of necessary common block declarations.
* Copying of necessary single-line function definitions.
* Copying of necessary IMAGE and PACKFMT statements.
* Moving declarations of variables that are only using in the new subunit into that subunit.
* Copying of OPTION statements.

The Program Analyst does not alter your program. It produces two ASCII files that you can GET and MERGE to create an altered program. If you are planning to extract more than one subunit, you must do the GET and MERGE for the first subunits before analyzing subsequent subunits.

Extracting a Subunit. The following is a list of the steps required to extract a subunit:

1. Renumber the program to allow space between lines so the Program Analyst can insert any necessary lines.

2. Get and examine a listing. The Program Analyst normally assumes lines are to be extracted from the MAIN subunit. Also, the Program Analyst cannot create a multi-line function this way. Look at the listing to find logical subunit material.

3. Enter the Program Analyst and go to the Extract Subunit screen. Enter the line range that you want to analyze. Use existing line

numbers.

4. When you enter a line range, the Program Analyst displays information about branches, structured statement, variables, and detectable problem conditions. Try different line ranges until you find one that will not cause many broken branches.

5. Press the softkey labeled Extract. Use any legal HP Business BASIC/XL identifier. The Program Analyst creates the two files used to produce the altered program. The first file is the original program with the extracted lines replaced by a CALL statement, and the second is the new subunit.

6. Exit the Program Analyst.

7. GET the original program file (the one whose name begins with g) and then MERGE the new subunit file (the one whose name begins with m). Use a line number on the MERGE command that will add the new subunit to the end of the program.

8. Renumber the program, and SAVE it. During the SAVE, the Program Analyst will identify any remaining broken structures that will need manual correction.

INFO Command

The INFO command prints information about the current interpreter environment. The INFO command is a command-only statement. That is, it can only be issued at the interpreter prompt and cannot be placed in a program. Default values can be changed by running the HP Business BASIC/XL configuration utility program as described in Appendix C.

Syntax

INFO

Example

The following shows the results of the INFO command:

```
 >INFO
Current settings:
   Numeric format                      STANDARD
   Angular units                       RAD
   DEFAULT expression evaluation       OFF
   Options                             REAL, INIT, BASE 0,
                                       NO DECLARE, TRACE
   Home group (FILES ARE IN)           None defined
   Destination of SEND OUTPUT TO       DISPLAY
   Destination of SEND SYSTEM OUTPUT TO   DISPLAY
   Destination of COPY ALL OUTPUT TO   DISPLAY
   Subunit size                        499 words
   Native Language                     0 (Native-3000)
   DATE$ default                       0
```

HELP Command

The HELP command displays information about one or more HP Business BASIC/XL commands, statements, and errors. Because a HELP command can end in an unquoted string literal, it cannot be followed by a comment. The HELP command is a command-only statement.

Syntax

```
     [unquoted_str_lit ]
HELP [str_lit          ]
```

Parameters

unquoted_str_lit String of the form

topic,subtopic

where *topic* and *subtopic* are unquoted string literals (for example, AUTO,SYNTAX). Characters beyond the fortieth are ignored.

str_lit Its value is a string of the form

"topic,subtopic"

(for example, "AUTO,SYNTAX").

If neither *unquoted_str_lit* nor *str_lit* is specified, the HELP command enters the HELP subsystem, where it prompts for them (the prompt is Help>). After displaying information on one topic, the HELP subsystem prompts for another.

If the HELP command recognizes *unquoted_str_lit* or *str_lit* as a subtopic, but not as a topic, it uses the last topic.

When the HELP command cannot recognize a spelling, the HELP command displays information on the topic or subtopic with the closest spelling (the first letter must be correct).

The information on a command or statement includes a list of similar commands and statements; for example, the information on INPUT lists ACCEPT, ENTER, and LINPUT.

If the information for a command, statement, or error fills more than one terminal screen, the HELP command displays one screen of information and then asks:

```
     Do you want to see more?  (Y/N) Y
```

To answer yes, do one of the following:

- Press RETURN.
- Type Y over the last Y and press RETURN.

To answer no, do one of the following:

Type N over the last Y and press RETURN.

To exit the HELP subsystem, type one of the following in uppercase, lowercase, or a combination of uppercase and lowercase:

```
     EXIT
     EXI
     EX
     E
```

Accessing the Operating System

The operating system can be accessed from HP Business BASIC/XL with the SYSTEM, SYSTEMRUN, or EXIT commands as follows:

SYSTEM — Executes an operating system command, a UDC, a program, or a command file from HP Business BASIC/XL (and returns to HP Business BASIC/XL). You can also type

```
     :
```

instead of SYSTEM.

SYSTEMRUN — Runs another program from HP Business BASIC/XL (and returns to HP Business BASIC/XL).

EXIT — Exits HP Business BASIC/XL (and does not return).

The SYSTEM and SYSTEMRUN commands are executable from within a program, and are discussed in chapter 4.

EXIT Command

The EXIT command exits HP Business BASIC/XL. It is a command-only statement and cannot appear in a program.

Syntax

```
{EXIT [BASIC]}
{::          }
```

Typing EXIT after changing your program without saving the complete program will result in the following question:

```
     UnSAVEd source modifications will be lost. Do you really want to
EXIT? Y
```

To exit, press RETURN or Y RETURN. To return to HP Business BASIC/XL, press any other character key followed by RETURN. Function keys and other special keys will not return you to HP Business BASIC/XL.

The :: form of the EXIT command does not check for source modifications before exiting.

Example

```
EXIT
::
```

The Calculator

The HP Business BASIC/XL interpreter can be used as a calculator. If you type in a numeric expression without a line number, HP Business BASIC/XL will return the value of that expression. Table 2-12 summarizes the what happens with each response to the interpreter prompt. Note that at the end of each line, you can type either RETURN or CONTROL E.

Table 2-12. The Calculator

In Response to the Interpreter Prompt (>), Type:	**Then press:**	**Effect**
Expression (except a numeric literal)	RETURN	HP Business BASIC/XL displays the value of the expression.
num_lit	RETURN	HP Business BASIC/XL deletes the program line that is numbered *num_lit*.
Program line	RETURN	HP Business BASIC/XL adds the program line to the program.
Anything else	RETURN	RETURN is treated as CONTROL E
Anything	CONTROL E	HP Business BASIC/XL executes the statement that would result from putting "DISP" before what was typed.
Nothing	CONTROL E	HP Business BASIC/XL executes the STEP command. Refer to "Debugging a Program" earlier in this chapter for more information.

Example

```
 >2+2  RETURN             (numeric expression)
4
>(5*(27/3))  RETURN      (numeric expression)
45
>Index1=12  RETURN       (assignment)
12
>Index2=3  RETURN        (assignment)
3
>Index1*Index2  RETURN   (numeric expression)
```

```
36
>10 PRINT  RETURN          (program line)
>10                        (line number with nothing else deletes line)
>Index1*Index2  RETURN     (numeric expression)
36
>Index1=12  RETURN         (assignment)
12
>Index2=Index1  RETURN     (assignment)
12
>10 PRNIT  RETURN          (syntax error)
Error
```

Chapter 3 Language Elements

Introduction

This chapter describes elements of the HP Business BASIC/XL language. It covers executable input, statements, variables, operators, and subunits.

Executable Input Units

An executable input unit can be input and executed without being part of a larger structure. The following are the three executable input units in HP Business BASIC/XL:

* Expressions.
* Commands.
* Programs.

Table 3-1 compares them.

Table 3-1. Executable Input Units

Executable Input Units	Composed of	For More Information
Expression	Operands and operator, or function and arguments. Operands and arguments are variables or expressions.	On expressions in general: Chapter 3. On executing expressions: "Calculator Mode" in Chapter 2. On variables: Chapter 3.
Command	Most statement elements, except line number and line label.	On commands in general: Chapter 3.
Program	Numbered program lines (statements that are not commands).	On program lines in general: Chapters 3 and 4. On program development and execution: Chapter 2.

Statements and Their Elements

This section contains the following information:

- Gives the syntax of the general statement and briefly explains each statement element.
- Explains the different types of statements.
- Provides further information on the statement elements keyword and identifier.
- Lists the places in a program where spaces are required or are illegal.
- Compares remarks and comments.
- Explains how statements form a program.

Statement Syntax

Every HP Business BASIC/XL statement has the following syntax, although not every statement can have all of the optional elements shown. See "Statement Types" in this chapter for restrictions on the general syntax.

Syntax

{*line_num* [*line_label*:]} [*Statement_body*] [*comment*]

Parameters

line_num Integer in the range[1, 999999]. Leading zeros are not significant. Each program line in a program must have a unique line number. Program lines are executed in line number order unless control statements specify otherwise.

line_label Identifier. For a description of an identifier, see "Identifiers". If a program line has a label, it can be referenced by either its label or line number.

statement_body The part of the statement that is specified by its syntax specification. Syntax specifications for individual statements appear in chapter 4. The *statement-body* is composed of the following statement elements:

Table 3-2. Statement Elements

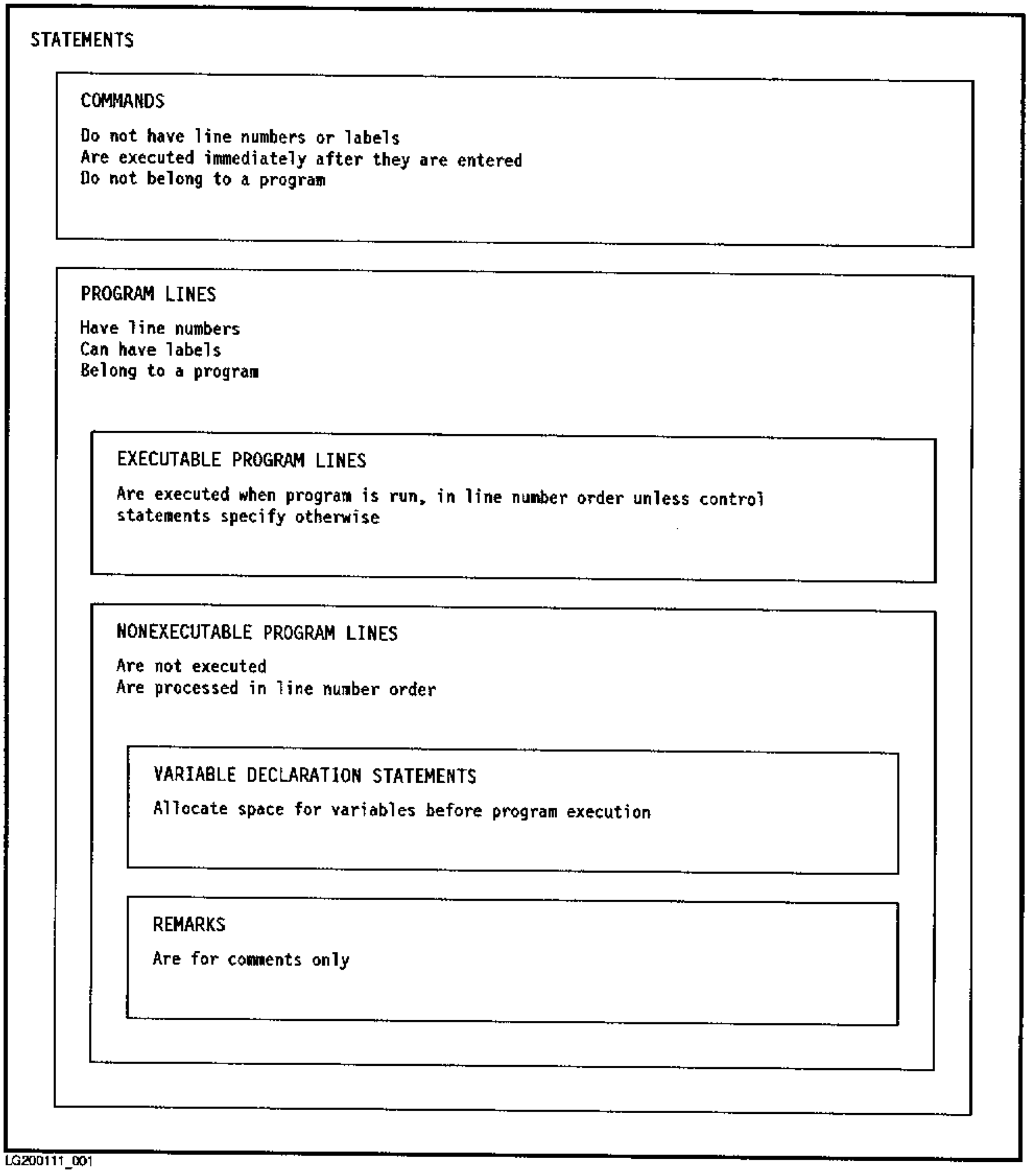
Statement Element	**Explained in**
Keyword	"Keywords".
Variable name	"Variable Names".
Spaces	"Spaces".
Literal	"Numeric Literals" and "String Literals".
Expression	"Operators" or "Subunits".

comment Any character string, including the null string. A *comment* cannot follow a HELP command or an IMAGE or DATA statement.

Statement Types

The HP Business BASIC/XL statement types and their relations are shown in Figure 3-1. Figure 3-1 also lists the characteristics of each statement type.

Figure 3-1. Statement Types



STATEMENTS

COMMANDS

Do not have line numbers or labels
Are executed immediately after they are entered
Do not belong to a program

PROGRAM LINES

Have line numbers
Can have labels
Belong to a program

EXECUTABLE PROGRAM LINES

Are executed when program is run, in line number order unless control statements specify otherwise

NONEXECUTABLE PROGRAM LINES

Are not executed
Are processed in line number order

VARIABLE DECLARATION STATEMENTS

Allocate space for variables before program execution

REMARKS

Are for comments only

The following are command-only statements:

ANALYST

AUTO

CALLS

CHANGE

COMPGO

COMPILE

COMPLINK

CONTINUE

COPY

CWARNINGS

EXIT(or ::)

FILES

FIND

HELP

HOP

INFO

LIST

LIST SUBS

MODIFY

MOVE

NAME

REDO

RENUMBER

RUN

STEP

VERIFY

XREF

A command-only statement cannot be a program line. Every other Business BASIC\XL statement can be a command or a program line. There are also some statements that cannot be a command. That is, they can only appear in program lines.

The following are statements that can appear as part of a program line, but can not be issued as a command:

ACCEPT

CASE

CLEAR FORM

CLOSE FORM

COPTION

DATA

DECIMAL

DEF FN

DIM

ELSE

END LOOP

END SELECT

END WHILE

END IF

ENTER

EXIT IF

EXTERNAL

FLUSH INPUT

FNEND

FOR

GLOBAL COPTION

GLOBAL EXTERNAL

GLOBAL INTRINSIC

GLOBAL OPTION

IF THEN ELSE

IMAGE

IN DATASET

INPUT

INTEGER

INTRINSIC

LENTER

LINPUT

LOOP

MAT INPUT

MAT READ

NEXT

OPEN FORM

OPTION

PACKFMT

PAUSE

READ

READ FORM

REAL

REPEAT

SELECT UNTIL

SHORT DECIMAL

SHORT INTEGER

SHORT REAL

SUBEND

SUBEXIT

SUBPROGRAM

THREAD IS

TINPUT

WHILE

WORKFILE IS

WRITE FORM

All other statements can be issued as a command, and can appear in a program line.

Keywords

Keywords are the basis of statements.

A keyword can be entered in the following ways:

 * All uppercase letters (for example, PRINT).
 * All lowercase letters (for example, print).

A keyword cannot be entered using a combination of uppercase and lowercase letters; for example, PrINt, or Print.

Regardless of how keywords are entered, HP Business BASIC/XL lists them in uppercase.

Examples

```
        10  LET B$="Chocolate"       !LET is a keyword
        20  PRINT X                  !PRINT is a keyword
        30  ON I GOTO 100,200,300    !ON and GOTO are keywords
```

Identifiers

An identifier is a character string that has the following characteristics:

- Begins with a letter.
- Contains any combination of letters, digits, and underscores (_).
- Has 63 or fewer characters.

HP Business BASIC/XL uses identifiers for several purposes. Table 3-3 shows those uses.

Table 3-3. Identifier Uses

Use of Identifier	Required Modifier	Example
Numeric variable name	None	Total
Line label	None	Return_point:
User-defined subprogram name	None	Routine
User-defined function name	Prefix FN	FNAdd
String variable name	Suffix $	Name$

An identifier can be entered in the following ways:

- All uppercase letters; for example, NAMES (See note below).
- All lowercase letters; for example, names.
- A combination of uppercase and lowercase letters; for example NaMeS.

NOTE If an identifier has the same spelling as a keyword, it must be typed in a combination of uppercase and lowercase letters. For example, "Print" or "pRiNt" is an identifier, but "print" or "PRINT" is a keyword. If such an identifier appears where the keyword is illegal, Business BASIC\XL recognizes it as an identifier. For example, HP Business BASIC/XL interprets "PRINT IF" as "PRINT If", where If is an identifier.

In general, identifiers should not have the same name and spelling as keywords. This can be very confusing, especially when attempting to debug a program.

Regardless of how an identifier is entered, HP Business BASIC/XL prints it with the first character upshifted and the others downshifted. For example, "NAMes" and "NAMES" become "Names", and both refer to the same entity.

Examples

Table 3-4. Legal Identifiers

Legal Identifier	Printed
X	X
grand_total	Grand_total
Sub_total_123	Sub_total_123
i	I
A_	A_
variablename	Variablename
LEGAL_IDENTIFIER	Legal_identifier

Table 3-5. Illegal Identifier

Illegal Identifier	Reason It Is Illegal
1XYZ	First character is not a letter.
#illegal	First character is not a letter.
sub'total	Contains a character that is not a letter, digit or underscore.

Spaces

One of the following must separate a keyword and an identifier:

- Space.
- Comma.
- Parenthesis.
- Operator.

With few exceptions, spaces can appear anywhere in a program. Table 3-6 lists the places where spaces cannot appear and gives examples.

Table 3-6. Places Where Spaces Are Not Allowed

Space Is Not Allowed	Correct Example	Incorrect Example
Within a line number.	1020	10 20
Within a keyword.	PRINT	PR INT
Within an identifier.	Grandtotal	Grand total
Within a numeric literal.	10000	10 000
Within a multicharacter relational operator symbol.	<>	< >
Between the identifier and $ in a string variable name.	Astring$	Astring $
Between the FN and the identifier in a function name.	FNAdd	FN Add

When a keyword has an alternate spelling that contains an embedded space, the two parts can be separated by more than one space. The keywords in the left column below can also be initially spelled as shown in the right column. The space in the second column can be replaced with more than one space. HP Business BASIC/XL prints these keywords as shown in the left column.

Table 3-7. Keywords with Alternate Spellings

Keyword	Alternate Spelling
ENDIF	END IF
ENDLOOP	END LOOP
ENDSELECT	END SELECT
ENDWHILE	END WHILE
FNEND	FN END
GOSUB	GO SUB
GOT0	GO TO
SUBEND	SUB END
SUBEXIT	SUB EXIT

Comments versus Remarks

Both comments and remarks are used for documentation only. Table 3-8 summarizes their similarities and differences.

Table 3-8. Comments vs Remarks

	Relationship to Program	How Recognized	Effect on Run-time Efficiency
Comment	Optional part of program line.	Follows ! on program line.	None (HP Business BASIC/XL ignores everything after !).
Remark	Nonexecutable program line.	Begins with the keyword REM.	Slight (HP Business BASIC/XL must read the word REM to determine that it does not need to do anything else for that line).

A comment can follow an empty statement as shown in line 400:

```
     300  REM This is a remark.
     400  !This comment follows an empty statement.
     500  PRINT "Hello"   !This comment is part of an executable program line.
```

HP Business BASIC/XL lists a remark with one blank between the keyword REM and the text of the remark, as in line 300 above. HP Business BASIC/XL lists a comment with one space before the exclamation point, as in line 400 above. A comment cannot follow a HELP command.

Program Structure

A program is a sequence of program lines. It is good programming practice to end a program with an END statement and use STOP statements within the program if the program must be stopped before it ends. However, the END statement can appear more than once in a program, and it need not be the last line.

Syntax

program_line [*program_line*]...[*END*] [*program_line*]...

Parameters

program_line Any statement except a command (that is, any statement with a line number).

The order that program lines are executed in is determined by line numbers and control statements. Program lines are executed in line number order unless control statements specify otherwise. Control statements are in chapter 4.

The lines of a program can be entered in any order. HP Business BASIC/XL arranges them in line number order before listing them or executing the program. Chapter 2 explains how to enter program lines.

A program can be divided into program units, one main program and one or more subunits. Execution begins with the first line of the main program unit. Subunits are covered later in this chapter.

Variables

In HP Business BASIC/XL, a variable can be numeric or string, scalar or array, local parameter, or common. This section explains declaring and using variables.

Certain characteristics of variables can be changed in a program unit. The OPTION and GLOBAL OPTION statements can change the following program unit characteristics:

* Default numeric type.
* Initialization of numeric variables to zero.
* Implicit variable declaration.
* Default lower bound of arrays.
* Trace statement output control.

- * Whether program main is the outer block for a multiprogram application.

The OPTION and GLOBAL OPTION statements are explained in chapter 4.

Variable Declaration

A variable can be declared as **local** to one program unit or **common** to two or more program units. A local variable can be accessed only by the program unit in which it is declared, whereas a common variable can be accessed by every program unit that declares it.

A local variable can be declared explicitly by a variable declaration statement, or implicitly the first time it is used. A common variable must be explicitly declared with the COM statement in every program unit that uses it.

Table 3-9 lists the variable declaration statements and the characteristics of the variables that they can declare.

Table 3-9. Variable Declaration Statements

Variable Declaration Statement	Type of Variables Declared
COM	Any
DIM	Any
SHORT INTEGER SHORT INT	Short Integer
INTEGER INT	Integer
SHORT REAL SHORT	Short Real
REAL	Real
SHORT DECIMAL SHORT DEC	Short Decimal
DECIMAL	Decimal

DEC		

Variable declaration statements can appear anywhere in a program. In the interpreter, before the main procedure or function is executed, HP Business BASIC/XL allocates space for both explicitly and implicitly declared variables. A variable cannot be explicitly declared more than once in a program unit.

If a variable appears in a program line, but not in a declaration statement, its name determines whether it is a numeric or string variable and the context determines whether it is a scalar or an array.

Variable Names

Table 3-10 gives examples of numeric and string variables names.

Table 3-10. Variable Names

Variable Type	Variable Name	Examples
Numeric	Identifier	Sum Grand_total X
String	Identifier with $ appended	Name$ Date1$ A$

A variable name is recognized throughout the program unit that declares it.

Within a program unit, the following items can have the same name:

* Scalar numeric variable.
* Scalar string variable.
* Numeric array variable.
* String array variable.
* Line label.
* Common area name.

Context determines whether the name refers to a scalar variable, an array variable or a line label.

Example

The following are examples of declaring variables:

```
100 INTEGER B       !Declares scalar numeric variable B
110 INTEGER B(5)    !Declares numeric array variable B
120 DIM B$[15]      !Declares scalar string variable B$
130 DIM B$(3)[15]   !Declares string array variable B$
140 B: STOP         !This line has a label, line label B
150 PRINT B         !B refers to scalar numeric variable
160 PRINT B(1)      !B refers to numeric array variable
170 PRINT B$        !B$ refers to scalar string variable
180 PRINT B$(3)     !B$ refers to string array variable
190 GOTO B          !B refers to line label
999 END
```

Numeric Variable Declaration Statements. Each numeric variable declaration statement explicitly declares one or more numeric scalar or array variables. The type of the variables depends on the statement.

Table 3-11 lists the numeric variable types, the number of bits used to store the value, the range of each type, the precision of each type, and the declaration statement that declares variables of that type. HP Business BASIC/XL accepts the character "D" as well as "E" to indicate scientific notation.

Table 3-11. Numeric Variable Data Types

Numeric Type	**Size (in bits)**	**Range**	**Precision**	**Declaration Statement**
Short integer	16	[-32768, 32767]	Exact	SHORT INTEGER
Integer	32	[-2147483648, 2147483647]	Exact	INTEGER
Short decimal	32	[-9.99999 E63, -9.99999 E-63], 0, [9.99999 E-63, 9.99999 E63]	Exact (6 digits)	SHORT DECIMAL
Decimal	64	[-9.99999999999 E511, -1.00000000000 E-511], 0, 1.00000000000 E-511, 9.99999999999 E511]	Exact (12 digits	DECIMAL
Short real	32	[-3.40282 E38, -1.17549 E-45], 0, [1.17549 E-45, 3.40282 E38]	Not Exact (6 digits)	SHORT REAL
Real	64	[-1.79769313486231 E308, -4.94065645841247 E-324], 0, [4.94065645841247 E-324, 1.79769313486231 E308]	Not Exact (15 digits	REAL

The syntax for each of these declaration statements is in chapter 4.

Array Variables. An array is an ordered collection of variables of the same type. If the array elements are string variables, they have the same maximum length.

An array element is legal wherever a scalar variable is legal.

An array variable is declared with a DIM, COM, or numeric declaration statement. The syntax for each of these is in chapter 4.

Implicit Declaration. If a program unit does not contain an OPTION DECLARE statement, its local variables can be declared implicitly, that is, the first time they are used, rather than with a COM, DIM, or numeric declaration statement.

Table 3-12 shows the characteristics that HP Business BASIC/XL gives to implicitly declared variables.

Table 3-12. Characteristics of Implicitly Declared Variables

Syntax of First Variable Reference	**Kind and Type of Variable**	**Size of Variable**
identifier	Scalar variable of default numeric type.	Not applicable
identifier (*i1,...,in*)	Array variable of default numeric type.	Dimensions:*n* where 1<= *n* <=6 Lower bound: default Upper bound: 10
identifier $	Scalar string variable.	Maximum length: 18 characters
identifier $(*i1,...,in*)	String array variable.	Dimensions: *n* where 1 <= *n* <= 6 Lower bound: default Upper bound: 10 Maximum length of each element: 18 characters

Variable Initialization

Before executing a program unit, HP Business BASIC/XL allocates space for its local variables. When HP Business BASIC/XL exits the program unit, it deallocates local variable space.

Table 3-13 shows how and when HP Business BASIC/XL initializes local and common variables.

Table 3-13. Variable Initialization

	Local Variable	**Common Variable**
Initialized to	Numeric: zero. String: null string.	Numeric:zero. String: null string.
When	Before HP Business BASIC/XL executes the program unit that declares the variable (for the main program unit, this is only when a RUN or GET command is executed).	Before HP Business BASIC/XL executes the first program unit that declares the variable.
How Often	Each time HP Business BASIC/XL executes that program unit.	Once.
Unless	OPTION NOINIT applies to that program unit.	GLOBAL OPTION NOINIT was specified.

Variable Reference

HP Business BASIC/XL can reference an entire scalar or array variable, a single array element, or a substring of a string variable. A substring reference can be made to a scalar string variable or a string array element.

Table 3-14 explains how to reference variables and variable parts, and gives examples.

Table 3-14. Variable References

Variable	**Reference by**	**Examples**
Scalar	*var_name*	X A$
Entire array	*var_name* or *var_name* (*[,*]...)	B(*), S$(*)
Array element	*var_name* (*num_expr* [,*num_expr*]...) One *num_expr* per dimension. For each dimension, *num_expr* is in the range [*lower_bound*, *upper_bound*]	B(1) S$(2,4,6)

Substring *	*str_name* [*num_expr1,num_expr2*]	A$[1,5]
	str_name [*num_expr1;num_expr2*]	A$[5;3]
	str_name [*num_expr1*]	S$[5]

Table 3-14 Note

* If the substring reference belongs to a string array variable, *str_name* must be an array element reference.

Variable Assignment

Numeric values must be assigned to numeric variables and string values must be assigned to string variables. A substring reference results in a string value that can be assigned to a string variable. Also, a string value can be assigned to a substring on the left hand side of the assignment statement.

Table 3-15 lists the types of values that can be assigned to variables.

Table 3-15. Possible Variable Assignments

Variable	Can Be Assigned Value of	Example
Numeric	Numeric variable	A=B
	Numeric expression	A=(B+3)*(C/5)
	Numeric literal	A=358
String	String variable	A$=B$
	String expression	A$=B$+C$
	String literal	A$="abcde"
	Substring	A$=B$[1,10]
Substring	String variable	A$[1,5]=C$
	String expression	A$[5]=C$+D$
	String literal	A$[1;3]="abc"
	Substring	A$[1;3]=B$[1;3]

When a string value is assigned to a string variable, the length of the string variable becomes the current length of the string value. The current length cannot exceed the maximum length of the string variable.

Several statements can assign values to variables. They are the following:

ACCEPT

ENTER

INPUT

LENTER

LET

LINPUT

READ

TINPUT

Numeric Literals

Numeric literals are real numbers or integers.

Syntax

For literal integers (*lit_integer*):

digit [*digit*]

For literal real numbers:

lit_integer [.[*lit_integer*]][E[+,-]*lit_integer*] .*lit_integer*

[E[+,-]*lit_integer*]

Parameters

digit A single digit 0..9.

lit_integer A number consisting of any combination of the digits 0..9.

Examples

Table 3-16. Examples of Numeric Literals

Literal Integers	Literal Real Numbers
8	9.00
123	35.9E+6
406903	.74E-3

A literal integer is stored as an integer or a short integer, depending on the range required.

Context determines the data type used to store a literal fixed-point or floating-point number. A literal floating-point number is stored as a real or decimal, depending on the precision required.

A literal that is beyond the range of the data type that it is to be stored in is rounded. If it is beyond the range of the largest data type, an error occurs.

String Literals

String literals are quoted string literals or special character string literals.

Syntax

Quoted string literal:

```
 {nonquote }
"{""      }..."
```

Special character string literal:

'*integer*

Parameters

nonquote Nonquoted string literal. Any character except a double quote(").

integer Special character string literal. Must be in the range [0,255]. Represents an ASCII character.

Examples

The quoted string literals in the left column below are printed as shown in the right column. The fourth example is the null string.

Table 3-17. Examples of String Literals

String Literal	**Printed**
"cat"	cat
"black cat"	black cat
"12345"	12345
""	Nothing
""""	"
"say 'hi'"	say 'hi'
"say ""hi"""	say "hi"

The following are special character string literals representing ASCII characters

Character	**What It Represents**	**Note**
'0	NUL	null

'7	BEL	entry rings terminal's bell
'13	CR	carriage return
'48	0	zero

String Lengths

Associated with every string variables is a maximum length and a current length. Table 3-18 explains the two types of length. The units are the number of eight bit characters.

Table 3-18. Maximum vs Current String Length

	Maximum Length	**Current Length**
Definition	Length of longest string value that can be assigned to string variable.	Length of string value that is currently assigned to string variable.
Range	[1, 32767].	[0,*curlength*] where *curlength* is current length.
Assigned	Once, when string variable is declared.	Each time a value is assigned to string variable.
Default	18 (for implicitly declared string variable).	Zero (length of the null string).

When program execution begins in a main, procedure or function in which a local string variable is declared either explicitly or implicitly and OPTION INIT is active, the current length is initialized to zero. The effect of initializing a string variable to zero is to set the value of the string to the null string.

Strings declared in a common area are initialized to the null string if the INIT option is active when the main procedure or function that contains the first occurrence of that common area begins execution.

Examples

```
 100  DIM A$           !Maximum length of A$ is 18 by default
110  DIM B$[5]        !Maximum length of B$ is 5
120  A$="Cat"         !Current length of A$ is 3
130  B$="Birds"       !Current length of B$ is 5
```

```
    140  C$="Elephants"   !Current length of C$ is 9, implicit definition
    150  A$="Caterpillar" !Now the current length of A$ is 11
    999  END
```

Substrings

Substring Operations. Substring operations are classified into two types; references and assignments. Substring references are specifications of a string of characters that are to be extracted from a string variable. The value of the string with the substring reference is never changed. Substring references can occur alone on the right hand side of assignment statements, in PRINT and PACK statements, and as arguments to some built-in string functions, for example, UPC$. Execution of a statement which contains a substring assignment results in a possible change to the value of the string variable. Substring assignment can occur as the target of an assignment statement on the left hand side, in INPUT, TINPUT, LENTER, and other input statements and in the UNPACK statement.

Substring References. A substring reference is a user-specified string of characters that begins at a character specified by an index for a string variable and has a length. By definition, the index of the first character in a string is one. The length of a substring determines the index of the last character in the string. If the index of the substring's last character in the string is greater than the actual length of the string variable then spaces are added to the characters referenced until a string of characters with the appropriate length is built.

There are two methods for specifying the substring value to be referenced. The first is specification of the start index alone. The second is specification of the start index and either the index of the last character or the length.

Start Index Only. Syntax

str_var [*start*]

Parameters

str_var A valid string variable name or string variable array element reference.

Start A numeric literal or expression that evaluates to a value between 1 and LEN(*str_var*)+1, inclusive.

Example

Consider the following substring reference:

```
     10  PRINT A$[Start]
```

The statement references the substring starting at the character at index Start in the string, A$. If LEN(A$) = 0 then the value is a null string. Otherwise, it is that string beginning at character index start of A$ and ending at character index LEN(A$). (LEN is a function that returns the length of a string. It is described in chapter 5.) If start = (LEN(A$)+1) then the value is the null string.

Start Index and End Index or Length.

Syntax

str_var [*start*,*end*] *str_var* [*start*;*length*]

Parameters

str_var A valid string variable name or string variable array element reference.

start A numeric literal or expression that evaluates to a value between 1 and LEN(*str_var*)+1, inclusive.

end A numeric literal or expression that evaluates to a value between *start* -1 and 32767, inclusive.

length A numeric literal or expression that evaluates to a value between 0- and 32770, inclusive.

Example

Consider the following two statements:

```
10 PRINT A$[Start,End]
20 PRINT A$[Start;Length]
```

Both statements reference the substring starting at the character at index Start in the string, A$.

If End = (Start-1) or Length =0, then the value is the null string.

If Start = LEN(A$)+1), then the value is a string of (End-Start+1) or Length spaces.

If End or (Start+Length-1) > MAXLEN(A$) then an error occurs.

For statement 10, if LEN(A$) >= End, then the value is the string beginning at character index Start of A$ and ending at character index End. Otherwise, the value is all of the characters from character index Start of A$ until character index LEN(A$) with spaces appended to the end of the value for a total of (End-Start+1) characters.

For statement 20, if LEN(A$) >= (Start+Length-1) then the value is the string beginning at character index Start of A$ and ending at character index (Start+Length-1). Otherwise, the value is all characters from character index Start of A$ until character index LEN(A$) with spaces appended to the end of the value for a total of Length characters.

```
10 A$="basic"     !Substring values on the RHS of assignment
```

```
     20 B$=A$[1]       !Assigns "basic" to B$ - characters 1 to LEN(B$)
     30 B$=A$[2,3]     !Assigns "as" to B$ - characters 2,3
     40 B$=A$[2;3]     !Assigns "asi" to B$ - characters 2,3,4
     50 B$=A$[4,7]     !Assigns "ic  " to B$ - characters 4,5 + 2 spaces
     60 B$=A$[6;2]     !Assigns "  " to B$ - a null string + 2 spaces
     70 B$=A$[7,10]    !Range error because 7 > LEN(A$)+1
```

Substring Assignment. A substring assignment begins at a user-specified index corresponding to a character position in a string variable and has a length. By definition, the index of the first character in a string is one. The length of the substring determines the index of the last character in the string to which a value is assigned. If the number of characters assigned to the string is less than the length of the substring specified, then spaces are assigned to the remaining characters in the string variable until the number of characters assigned is equal to the length of the substring.

There are two methods for specifying the target substring. The first is the specification of the starting index alone, and the second is specification of the starting index and either the index of the last character or the length.

Start Index Only. Syntax

str_var [*start*]

Parameters

str_var A valid string variable name or string variable array element reference.

start A numeric literal or expression that evaluates to a value between 1 and LEN(*str_var*)+1, inclusive.

Example

Consider the following assignment statement:

```
     10  A$[Start]=B$
```

Execution of this statement assigns the value of B$ to A$ beginning at character Start.

If Start= (LEN(A$)+1), then a string append is done.

If the LEN of the string following assignment is greater than that before assignment, then the actual length of A$ is reset.

If (Start+LEN(B$)-1) <= MAXLEN(A$), then the value of B$ is assigned to A$. Otherwise, (MAXLEN(A$)-Start+1) characters from the value of B$ are assigned to A$.

Note that as long as 1 <= Start <= LEN(A$)+1, then regardless of the length of B$, no bounds violation occurs during the string assignment.

Start Index and End Index or Length.

Syntax

str_var [*start*,*end*] *str_var* [*start*;*length*]

Parameters

str_var A valid string variable name or string variable array element reference.

start A numeric literal or expression that evaluates to a value between 1 and LEN(*str_var*)+1, inclusive.

end A numeric literal or expression that evaluates to a value between *start* -1 and 32767, inclusive.

length A numeric literal or expression that evaluates to a value between 0- and 32767, inclusive.

Example

Consider the following two assignment statements:

```
     10 A$[Start,End]=B$
     20 A$[Start;Length]=B$
```

Execution of either of these statements assigns the value of B$ to A$ beginning at character Start.

If Start = (LEN(A$)+1) then a string append is done.

If the LEN of the string after the assignment is greater than that before assignment then the actual length of A$ is reset.

If LEN(B$) >= (End-Start+1) or Length then the number of characters from B$ assigned to A$ is equal to (End-Start+1) or Length, respectively. If LEN(B$) < (End-Start+1) or Length, then the value of B$ is assigned to A$ beginning at character position Start. Spaces assigned to each remaining character in A$ up to and including the character with index End or until a total of Length characters has been assigned.

If End or (Start+Length-1) > MAXLEN(A$), then a bounds violation occurs.

```
     10 B$="basic"       !Assigns a value to B$
     20 A$[1]=B$         !Value of A$ is "basic"
     30 A$[2,6]=B$       !Value of A$ is now "bbasic"
     40 A$[1;5]=B$       !Value of A$ is now "basicc"
     50 A$[4,6]=B$       !Value of A$ is now "basbas"
     60 A$[7]=B$         !Value of A$ is "basbasbasic" - string append
     70 A$[1,6]=B$       !Value of A$ is "basic basic" - 1 space was assigned
     80 A$[13;5]=B$      !Range error because 13 &> LNE(A$)+1
     90 A$[LEN(A$)+1]=B$ !Value of A$ is "basic basicbasic" - string append
```

Expressions

An expression is an operator with its operands or a function call. HP Business BASIC/XL evaluates an expression and returns a result.

Syntax

```
                     {operand                              }
[operand ] operator  {func_name  [(parameter  [,parameter ]...)]}
```

Parameters

operand First operand is required if operator is binary and not allowed if operator is unary.

Each operand is one of the following:

- A literal.
- A variable name.
- An expression.

Operand type restriction depends on the operator.

operator Determines how the value(s) of the operands(s) produce(s) the result.

func_name Function name. HP Business BASIC/XL supports the following types of functions:

- Predefined function.
- Single-line user-defined function.
- Multi-line user-defined function.

parameter The number of parameters depends on the function. Each parameter is one of the following:

- A literal.
- A variable name.
- An expression.

Parameter type restrictions are dependent on the types of the function's formal parameters.

The result of an operation or a predefined function depends on the values of the operands or parameters, but the values of the operands and parameters do not change.

A user-defined function can change the values of its parameters if the parameters are passed by reference.

Operators

HP Business BASIC/XL has three unary operators: unary plus(+), unary minus (-), and NOT. All other HP Business BASIC/XL operators are binary. Also, each HP Business BASIC/XL operator is either an arithmetic, relational, Boolean, or string operator, depending on the types of its operands and result.

For each operator category, Table 3-19 gives the operand and result types.

Table 3-19. Operands and Result Types of Operators

Operator Category	**Operand Type**	**Result Type**

Arithmetic	Numeric	Numeric
Relational	Numeric or string	Boolean*
Boolean	Boolean*	Boolean*
String	String	String

Table 3-19 Note

* A Boolean value is actually a numeric value. TRUE is one and FALSE is zero.

Arithmetic Operators

An arithmetic operator has numeric operands and a numeric result.

Table 3-20 identifies each arithmetic operator as unary or binary and gives its name and an example.

Table 3-20. Arithmetic Operators

Operator	Unary or Binary	Operation Name	Example (expression=result)
+	Unary	Unary plus	+5=5
+	Binary	Addition	1+2=3
-	Unary	Unary minus	-(4+4)=-8
-	Binary	Subtraction	8-4=4
*	Binary	Multiplication	9*7=63
/	Binary	Real division	36/4=9.0
DIV	Binary	Integer division	37 DIV 4=9

MOD	Binary	Modulus	37 MOD 4=1
^	Binary	Exponentiation	2^3=8
**	Binary	Exponentiation	2**3=8
MIN	Binary	Minimum	5 MIN 4=4
MAX	Binary	Maximum	5 MAX 4=5

The result of real division is of the default numeric type. The result of integer division is truncated to a whole number. If the result is within range, the type is integer. Otherwise, it is decimal or real.

Examples

The following examples show the results of division on different data types:

```
     3 DIV 2 = 1     -10 DIV 5 = -2     9.999999999 DIV 1 = 9
     3/2 = 1.5       -10/5 = -2         9.999999999/1 = 9.999999999
```

The result of the operation

num_expr1 MOD *num_expr2*

is

num_expr1 -(*num_expr2* *INT(*num_expr1* /*num_expr2*))

where INT(*x*) returns the largest integer less than or equal to *x*, for any numeric expression *x*. By definition, *x* MOD 0 = *x* for any numeric expression *x*. The result of the MOD operation is of the default numeric type, DECIMAL or REAL.

Examples

The following are examples of the result of the MOD statement. Each example shows the math required to determine the result.

```
      38 MOD 6   = 38 - (6*INT(38/6))
                 = 38 - (6*6)
                 = 38 - 36
                 = 2

     13 MOD -2  = 13 - (-2*INT(13/-2))
                = 13 - (-2*-7)
                = 13 - 14
                = -1

     -13 MOD 2  = -13 - (2*INT(-13/2))
                = -13 - (2*-7)
                = -13 - (-14)
                = -13 + 14
                = 1

     -13 MOD -2 = -13 - (-2*INT(-13/-2))
                = -13 - (-2*6)
                = -13 - (-12)
                = -13 +12
                = -1
     3 MOD 5    = 3 - (5*INT(3/5))
                = 3 - (5*0)
                = 3 - 0
                = 3
```

Relational Operators

A relational operator has either two numeric operands, two ASCII string operands, or the result of another relational expression and a Boolean result. Every relational operator is binary.

Table 3-21 gives the name and an example of each relational operator.

Table 3-21. Relational Operators

Operator	Operation Name	Example (expression=result)
<	Less Than	(1<2)=TRUE
<=	Less Than or Equal	(2<=1)=FALSE
=	Equal	(9=7)=FALSE
>=	Greater Than or Equal	(9>=4)=TRUE
<>	Not Equal	(36<>45)=TRUE
#	Not Equal	12#(6+6)=FALSE

String Comparisons. String comparisons are made by comparing each string operand character by character from left to right. The characters are compared based on each character's ordinal value in the ASCII character set. The ordinal value of a character is the value in the decimal code column in the ASCII character code table presented in Appendix D. To compare two strings when using a native language other than NATIVE-3000(language #0), the language the system uses before the introduction of Native Language Support, use the LEX function (for more information on Native Language Support refer to "Native Language Support" in chapter 6, or the *Native Language Programmer's Guide*).

The null string ("") is less than every string except itself, to which it is equal. The following explanation does not apply to the null string.

HP Business BASIC/XL compares the strings S1$ and S2$ as follows (S1$[*c*;1] and S2$[*c*;1] are corresponding characters).

1. *c* =1

2. If CHR$(S1$[*c*;1])<CHR$(S2$[*c*;1]), then S1$ is less than S2$. Stop.

3. If CHR$(S1$[*c*;1])>CHR$(S2$[*c*;1]), then S1$ is greater than S2$. Stop.

4. CHR$(S1$[*c*;1])=CHR$(S2$[*c*;1]). If *c* +1 is in the range [1, MIN(LEN(S1$), LEN(S2$))], then *c* =*c* +1 and return to step 2.

5. If LEN(S1$) = LEN(s2$) then S1$ is equal to S2$. Stop.

6. If LEN(S1$) > LEN(s2$) then S1$ is greater than S2$. Stop.

7. If LEN(S1$) < LEN(s2$) then S1$ is less than S2$. Stop.

(MIN and LEN are the predefined minimum and length functions.)

Examples

The following expressions are TRUE:

```
     "Abc" = "Abc"      "Cat" <> "Cats"      "Bird" < "Cats"
     "Abc" <= "Abc"     "Cat" < "Cats"       "Ears" > "Early"
    "Abc" >= "Abc"    "Cat" <= "Cats"     "Bird " + "Dog" = "Bird Dog"
```

The following expressions are FALSE:

```
     "Abc" # "Abc"      "Cat" =  "Cats"      "Bird" >= "Cats"
     "Abc" < "Abc"      "Cat" < "Bats"       "Ears" < "Early"
    "Abc" > "Abc"      "BAT" = "bat"       "Bird" + "Dog" = "Bird Dog"
```

Boolean Operators

A Boolean operator has one or two operands and a Boolean result.

The Boolean values TRUE and FALSE are represented by the numeric values one and zero. The operands of a Boolean expression can be Boolean or numeric values. A numeric operand is considered TRUE if it is nonzero and FALSE if it is zero.

HP Business BASIC/XL also provides the two keywords TRUE and FALSE. TRUE is a numeric constant of short integer type equal to one. FALSE is a numeric constant of short integer type equal to zero. Depending on the operator, HP Business BASIC/XL evaluates a Boolean expression either completely or partially.

Logical evaluation HP Business BASIC/XL always evaluates both operands.

Partial evaluation HP Business BASIC/XL always evaluates the first operand, but evaluates the second operand only if its value could change the value of the expression.

Table 3-22 identifies each Boolean operator as unary or binary, gives its name, and tells whether it is evaluated logically or partially.

Table 3-22. Boolean Operators

Operator	**Unary or Binary**	**Operation Name**	**Logical or Partial Evaluation**
NOT	Unary	Negation	Logical
LAND	Binary	Logical AND	Logical
AND	Binary	AND	Partial
LOR	Binary	Logical OR	Logical
OR	Binary	OR	Partial
XOR	Binary	Exclusive OR	Logical

Table 3-23 is the truth table for the NOT operator.

Table 3-23. NOT Truth Table

X	**NOT X**
TRUE	FALSE
FALSE	TRUE

Examples

These expressions are TRUE:

```
NOT 0     NOT (X-X)     NOT (5 = 3)     NOT ("HP" < "Competitors)
```

These expressions are FALSE:

```
NOT 1     NOT 3600      NOT (5 > 3)     NOT("HP" # "Hewlett Packard")
```

Table 3-24 is the truth table for the LAND and AND operators. The AND operator evaluates the first operand, and if it is FALSE the result is FALSE and the second operand is not evaluated. The LAND operator evaluates both operands regardless of the value of the first one.

Table 3-24. LAND/AND Truth Table

X	Y	X {LAND AND} Y
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Examples

These expressions are TRUE:

```
 (2 > 1) AND (1 > 0)                (2 > 1) LAND (1 > 0)
((X-1) <= X) AND (X <= (X+1))     ((X-1) <= X) LAND (X <= (X+1))
1 AND (1+0)                       1 LAND (1+0)
(3*5) AND ((1+2)/5)               (3*5) LAND ((1+2)/5)
("a" < "b") AND ("ant" < "bug")   ("a" < "b") LAND ("ant" < "bug")
```

These expressions are FALSE:

```
 (2 = 1) AND (1 > 0)                 (2 = 1) LAND (1 > 0)
((X-1) <= X) AND (X > (X+1))       ((X-1) <= X) LAND (X > (X+1))
(3*5) AND (0/5)                    (3*5) LAND (0/5)
("a" = "ant") AND ("b" = "bug")    ("a" = "ant") LAND ("b" =  "bug")
```

The program on the left below evaluates FNI(I); the program on the right does not. If the function FNI adds one to its argument, then the program on the left prints "0 1" and the program on the right prints "0 0".

```
10 I=0                            10 I=0
20 PRINT (I LAND FNI(I)); I       20 PRINT (I AND FNI(I)); I
```

```
          99 END                           99 END
```

If array A has Maxindex elements, and Index is greater than Maxindex, then the statement

```
     100 IF (Index <= Maxindex) AND (A(Index) =5) THEN GOTO 500
```

does not evaluate A(Index), and an error does not occur. The statement

```
     200 IF (Index <= Maxindex) LAND (A(Index) = 5) THEN GOTO 600
```

does evaluate A(Index), and an error occurs (subscript out of range).

Table 3-25 is the truth table for the LOR and OR operators. The OR operator evaluates the first operand, and if it is TRUE, the result is TRUE and the second operand is not evaluated. The LOR operator evaluates both operands regardless of the value of the first.

Table 3-25. LOR/OR Truth Table

X	**Y**	**X {OR LOR} Y**
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Examples

These expressions are TRUE:

```
     (X < (X+1)) OR (2 < 3)         (X < (X+1)) LOR (2 < 3)
     (X <= (X+1)) OR (5 = 3)        (X <= (X+1)) LOR (5 = 3)
     (9-(3**2)) OR ("a" < "z")      (9-(3**2)) LOR ("a" < "z")
```

These expressions are FALSE:

```
     0 OR (5-5)                     0 LOR (5-5)
     (9-(3**2)) OR (9-(6+3))        (9-(3**2)) LOR (9-(6+3))
     (X-X) OR ("a" > "z")           (X-X) LOR ("a" > "z")
```

The program on the left below evaluates FNI(I); the program on the right does not. If the function FNI subtracts one from its argument, then the program on the left prints "1 0" and the program on the right prints "1 1".

```
 10 I=1                         10 I=1
20 PRINT (I LOR FNI(I)); I     20 PRINT (I OR FNI(I)); I
99 END                         99 END
```

If array A has Maxindex elements, and Index is greater than Maxindex, then the statement

```
100 IF (Index &> Maxindex) OR (A(Index) =5) THEN GOTO 500
```

does not evaluate A(Index), and an error does not occur. The statement

```
200 IF (Index &> Maxindex) LOR (A(Index) =5) THEN GOTO 600
```

does evaluate A(Index), and an error occurs (subscript out of range).

Table 3-26 is the truth table for the XOR operator. XOR is different from the OR and LOR operators in that it returns TRUE only when the first or the second operator is TRUE, but the operators are not both TRUE. The OR and LOR operators return TRUE if one or both operands are TRUE.

Table 3-26. XOR Truth Table

X	**Y**	**X OR Y**
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Examples These expressions are TRUE:

```
0 XOR 1     (3-(2+1)) XOR 85       (6 <=5) XOR (7+3)
1 XOR 0     35677 XOR (9-(3*3))    (X < (X-1) XOR ("A" = "A")
```

These expressions are FALSE:

```
0 XOR 0     ("cat" = "dog") XOR ("a" = "b")     (X = (X+1)) XOR (X-X)
1 XOR 1     ("cat" < "dog") XOR ("a" < "b")     365 XOR 366
```

String Concatenation Operator

The string concatenation operator has two string operands and a string result.

Syntax

str_expr1 + *str_expr2*

The resulting string is the value of *str_expr1* with the value of *str_expr2* appended to it. The length of the resulting string is the sum of the two lengths.

Example

```
     10 Mystery1$="hot"+"dog"        !Mystery1$'s length is set to 6
     20 Mystery2$="base"+"ball"      !Mystery2$'s length is set to 8
     30 PRINT Mystery1$+"s"+" and "+Mystery2$
     40 ! Line 30 prints  -- hotdogs and baseball
```

Evaluation of Expressions

HP Business BASIC/XL evaluates a simple (one operator) expression by evaluating its operands or actual parameters from left to right, and then performing the operation or function.

Examples

```
     10 A=2
     20 B=7
     30 C=A+B
     99 END
```

In line 30 of the above program, HP Business BASIC/XL evaluates A and B (in that order) and then adds their values (2 and 7, respectively) to produce the result, 9.

```
     100 X=10
     110 Y=15
     120 Z=20
     130 Max_xyz=MAX(X,Y,Z)
     999 END
```

In line 130 of the above program, HP Business BASIC/XL evaluates the expression MAX(X,Y,Z) by first evaluating X, Y, and Z (in that order) and then comparing their values (10,15, and 20, respectively) and returning the largest value (20).

More complex expressions can be constructed by substituting expressions for the operands or parameters. For example, the expressions A+B and MAX(X,Y,Z) are operands of the addition operator in the expression (A+B)+MAX(X,Y,Z). HP Business BASIC/XL evaluates (A+B)+MAX(X,Y,Z) by first evaluating A+B and MAX(X,Y,Z) (in that order) as explained above, and then adding their values (nine and 20, respectively) to produce the result, 29.

When an expression has expressions for operands or parameters, operator hierarchy determines the order in which the component operations are performed. The general rule of left to right expression evaluation applies to the evaluation of each subexpression. For example, operator hierarchy dictates that the expression 2*3+4*5 is evaluated as (2*3) +

(4*5), where the expressions in parentheses are evaluated first.

Operator Hierarchy

When an expression contains several operators, operator precedence is used to determine the evaluation order. The operator hierarchy establishes the precedence relationship among the HP Business BASIC/XL operators. Expressions with operators of equal precedence are evaluated from left to right.

Table 3-27 shows the HP Business BASIC/XL operator hierarchy. An operator takes precedence over those below it in the table. Operators on the same line of the table have equal precedence.

Table 3-27. Operator Hierarchy

Operator or Operator Category	**Symbol(s)**
Subexpressions within Parentheses	()
Exponentiation Operator *	**, ^
Unary Operators	+, -, NOT
Multiplication and Division Operators	*, MOD, /, DIV
Addition and Subtraction Operators	+, -
Minimum and Maximum Operators	MIN, MAX
Relational Operators	<, <=, =, =>, >, <>, #
Boolean AND Operators	LAND, AND
Boolean OR Operators	LOR, OR, XOR

Table 3-27 Note

* A unary operator is applied to the exponent before the exponentiation operator is applied to its arguments. For example, -2**-2 is equivalent to -(2**(-2)).

Examples

```
 4+7*2 = 4+(7*2) = 4+14 = 18
(4+7)*2 = 11*2 = 22
3-2+1 = (3-2)+1 = 1+1 = 2
3-(2+1) = 3-3 = 0
NOT A**3 MOD 12 + 75 = B AND C OR D =
   (((((NOT(A**3)) MOD 12) + 75) = B) AND C) OR D
```

Result Type

If an arithmetic operation has two operands of the same type, the operation is performed using that type. The intermediate result is of that type, and an error occurs if the intermediate result is out of the range of the final result type. The following are exceptions:

- Short integer arithmetic, performed in integer arithmetic.

- Exponentiation in which the base is converted to a real for all types. The exponent is converted to a real for decimal, short decimal, and short real. The exponents for integers and short integers are not converted. That is, a short integer remains a short integer, and an integer remains an integer.

Examples

```
10 INTEGER A,B
20 REAL C
30 C=A+B
99 END
```

In line 30 of the above program, the intermediate result of A+B is an integer. It is converted to a real number when it is assigned to the real variable, C.

If an arithmetic operation has two operands of different types, one or both operands are converted to one type before the operation. The type that they are converted to depends on the default numeric type.

Precision can be lost when numbers are converted between real and decimal types. Overflow can occur when numbers are converted to a type with a smaller range (for example, real to short real).

Subunits

A program can be divided into program units consisting of one main program unit followed by one or more subunits. In this section, the main program unit is called the main program.

A subunit is a series of program lines that can be called with parameters, by another program unit. The calling program unit transfers control to the subunit; the subunit executes and returns control to the calling program unit. The calling program unit can be the main program or another subunit.

A subunit can contain any program lines that are valid in a main program, including variable declaration statements. Except for common variables, the variables that are defined in a subunit, including formal parameters, are local to that subunit. All variable names in the subunit represent variables that are distinct from variables with the same names in other program units. HP Business BASIC/XL allocates space for local variables when it enters a subunit, and releases that space to memory when it returns to the calling program.

When HP Business BASIC/XL enters a subunit, it suspends the ON ERROR, ON END, and ON HALT specifications form the last program unit until control returns to that program unit. Exceptions to this rule are those "ON" conditions that specify subunit calls, for example, ON ERROR CALL Error_Routine.

A subunit is either a subprogram or a user-defined multi-line function. A subprogram performs a task, but does not return a value to the calling program unit. A multi-line function returns a value to the calling program unit unless it is called as a subprogram, in which case the result is discarded.

Table 3-28 summarizes the differences between subprograms and multi-line functions.

Table 3-28. Subprograms vs Multi-line Functions

Subunit Type	**Subprogram**	**Multi-line Function**
Begins with	SUBPROGRAM or SUB statement.	DEF FN statement.
Ends with	SUBEND statement.	FNEND statement.
Returns to	Line following subprogram call.	Line containing function call.
Returns via	SUBEXIT or SUBEND statement.	RETURN statement.
Returns with value	No.	Yes.

Subprograms

A subprogram is a subunit that performs a task and returns control to the program unit that called it. It does not return a value to the calling program unit.

Syntax

SUB_stmt [*stmt*]...*SUBEND_stmt*

Parameters

SUB_stmt SUBPROGRAM or SUB statement. Not executable. Indicates that the lines that follow are a subprogram.

stmt Can be a SUBEXIT statement that returns control to the calling program unit before the SUBEND statement is executed, or can be any executable statement. These statements constitute the body of the subprogram.

SUBEND_stmt SUBEND statement. Indicates the end of the subprogram.

A subprogram follows the editing procedure described in chapter 2.

A program unit calls a subprogram with a CALL statement. The subprogram returns control to the statement following the CALL statement.

Example

```
  10 READ A,B               !Main program begins
 15 DATA 48,50
 20 CALL Sub1(A,B)         !Main program calls Sub1; go to line 100
 30 PRINT A
 40 PRINT B
 99 END                    !Main program ends
100 SUB Sub1 (X,Y)         !Subprogram Sub1 begins
105   DIM String$[1]
110   IF X<0 THEN SUBEXIT  !If X<0, Sub1 ends early; go to line 30
115   String$=CHR$(X+Y)    !If X=>0, Sub1 continues
120   PRINT String$
999 SUBEND                 !Subprogram Sub1 ends; go to line 30
```

User-Defined Multi-Line Functions

A user-defined multi-line function is a subunit that returns a value to the calling program unit. The value returned by a function has a specific type. A function that returns a numeric value is called a numeric function; A function that returns a string value is called a string function.

Syntax

DEFFN_stmt *stmt* [*stmt*] . . . *FNEND_stmnt*

Parameters

DEFFN_stmnt DEF FN statement. Not executable. Indicates that the lines that follow are a multi-line function.

stmt Executable statements that make up the body of the function. At least one *stmt* must be a RETURN statement that returns a value and control to the calling program unit.

FNEND_stmt FNEND statement. Indicates the end of the function.

A function is edited using the procedures described in chapter 2.

A program unit calls a multi-line function the same way it calls a predefined or single-line function: by its name, followed by an actual parameter list if it has one. The list of actual parameters is enclosed in parentheses, and the individual parameters are separated by commas.

Example

```
  10 READ A,B              !Main program begins
 15 DATA 48, 50
 20 C$=FNFunc$(A,B)       !Main program calls FNFunc$; go to line 100
 30 PRINT C$
 99 END                   !Main program ends
100 DEF FNFunc$(X,Y)      !Function FNFunc$ begins
105   DIM String$[1]
115   String$=CHR$(X+Y)
120   RETURN String$      !FNFunc$ returns value to line 20
999 FNEND                 !Function FNFunc$ ends
```

A multi-line function can also be called as a subprogram with the CALL statement. In this case, the value returned by the function is discarded.

If a program has more than one subunit with the same name, the name references the first subunit that it finds. The following is the search order:

1. Single-line function.
2. Local external or intrinsic subunit.
3. Internal multi-line function (one defined by the program).
4. Global external or intrinsic subunit.

Parameter Passing

An actual parameter can be passed to a subprogram by reference or by value. Actual parameters are passed by reference unless the individual actual parameter is enclosed in parentheses or is an expression or substring. Enclosing the actual parameter in parentheses specifies that the actual parameter is to be passed by value.

Table 3-29 compares the two methods. String or numeric literals are always passed by value. Arrays are always passed by reference.

Table 3-29. Parameter Passing Methods

	Actual Parameter Passed by Reference	**Actual Parameter Passed by Value**
Formal parameter is	The actual parameter itself.	Assigned the value of the actual parameter.
Subprogram can	If it changes corresponding formal	No.

change actual parameter	parameter.	
Variables Passed This Way	File designators*. Arrays. Array elements. Scalar numeric variables. Unsubscripted scalar string variables.	All not mentioned to the left. Scalar variables enclosed in parentheses. String literals. Numeric literals. Expressions Substrings
Corresponding parameters must be	Exactly the same type and both scalar or both array.	Compatible types.**

Table 3-29 Notes

* An actual parameter that corresponds to a formal file designator parameter must have a value that can be converted to a short integer in the range [1, 32767].

** An actual and formal parameter are compatible if the parameters are both string or both numeric (they must also be scalar, because whole arrays cannot be passed by value). If the parameters are of different numeric types, HP Business BASIC/XL converts the value of the actual parameter to the numeric type of the formal parameter before assigning it to the formal parameter.

Example

```
 10 A,B=0
 20 CALL Sub(A,(B))        !A is passed by reference
 25 REM                    !B is passed by value
 30 PRINT A                !Prints 1 (Sub changed A)
 40 PRINT B                !Prints 0 (Sub did not change B)
 99 END
100 SUB Sub (X,Y)          !A corresponds to X; B corresponds to Y
110   X=X+1
120   Y=Y+2
130   PRINT X              !Prints 1
140   PRINT Y              !Prints 2
150 SUBEND
```

The number of actual parameters in a subprogram call must be the same as the number of formal parameters in the SUBPROGRAM or DEF FN statement that defines the beginning of the subprogram or function. The actual parameters are evaluated and assigned to the corresponding formal parameters from left to right.

Initial Subprogram Environment

Every program unit has its own operating environment. When HP Business

BASIC/XL enters a subprogram, it initializes the environment. When control returns to the calling program unit, HP Business BASIC/XL reinstates the environment of the calling program unit.

Table 3-30 lists the characteristics that define the operating environment of a program unit and explains how each characteristic is initialized.

Table 3-30. Program Unit Operating Environment

Operating Environment Characteristic	**Initial Value of Characteristic Upon Program Unit Entry**
Data pointer position.	First datum in first DATA statement in program unit.
Accessible files.	Files passed as parameters and common files that program unit declares.
Trigonometric unit.	Radians.
Print format for numeric data.	Standard.
Default lower bound for arrays.	Depends on OPTION BASE.
ON ERROR specifications.	ON ERROR GOTO and ON ERROR GOSUB specifications that were active in the calling program unit are inactive; ON ERROR CALL specifications that were active in the calling program unit are active.
ON END specifications.	ON END specifications that were active in the calling program unit are inactive.

Using Common Variables in Subunits

A subunit can declare an entire common area or an initial subset of a common area that is declared in the main program. It can only access the common variables that it declares.

A program unit declares common areas with COM statements. A subprogram cannot contain common variables with the same names as its formal parameters or local variables.

Example

```
10 COM A(4,4), B, INTEGER C, D(3,3), E$[28], F$(2,4)[56]
```

```
        .
        .
        .
     99 END
    100 SUB Payroll
    110 COM X(*,*), Y, INTEGER Z,Q()
        .
        .
        .
    199 SUBEND
    200 DEF FNAccounts (X,Y,Z)
    210 COM I()
        .
        .
        .
    299 FNEND
```

The following table shows the correspondence between common variable names in the above program.

Table 3-31. Common Variable Names Correspondence

Name of Common Variable in Main Program	**Name of Common Variable in Payroll**	**Name of Common Variable in FNAccounts**
A	X	I
B	Y	None
C	Z	None
D	Q	None
E$	None	None
F$	None	None

VERIFY Command

The VERIFY command *verifies* specified program units; that is, it checks that they are *well-formed* and prints messages if it finds errors. The VERIFY command is a command-only statement, and it cannot be executed when the program is running.

A program unit is *well-formed* if it has the following characteristics:

- Properly matched constructs.
- Consistent array references.
- No incorrectly placed statements (for example, SUBEXIT in a

function).
- No undeclared variables under OPTION DECLARE.

Syntax

```
       [ALL                        ]
VERIFY [          [{,}         ]   ]
       [progunit  [{;} progunit ]...]
```

Parameters

ALL — Specifies all program units in the program, including the main program unit. ALL is the default.

progunit — One of the following:
[SUB]*subunit_name*.
[SUB]*function_name*.
[SUB]MAIN.

A program unit cannot execute unless it is *well-formed*. For this reason, HP Business BASIC/XL verifies a program unit at the following times:

- At run time, if it was modified since its last call.
- Before saving it in a BASIC Save file.

Therefore, you do not need to issue the VERIFY command to check a program before you run it, because HP Business BASIC/XL will issue it automatically. The purpose of the VERIFY command is to allow you to VERIFY a program as you develop it, without having to RUN or SAVE it.

Example

The following example shows what happens when a program is not *well-formed*. The example below shows the results of the VERIFY command. HP Business BASIC/XL has issued the VERIFY command when the programmer typed RUN.

```
 >10 OPTION DECLARE    !This specifies that all variables must be declared
>20 WHILE A           !A is not declared, and the WHILE statement
>25                   !is not closed
>30 PRINT A
>RUN
Error 179
Structured constant on line 20 not properly closed.
Error 1403
Undeclared variable A found in subunit MAIN.
Error 157
VERIFY error(s) in program.
```

Calling External Subunits

External routines fall into the following categories:

- Procedures (routines that do not return values).
- Functions (routines that return values).

An external routine is called with the CALL statement. An external function can be called with either the CALL statement or the FNCALL function; the method depends on the function name and whether its result can be discarded. Table 3-32 tells how to call each type of external subunit.

Table 3-32. External Subunit Calls

External Routine	Dependency	How to Call
Subprogram	None.	Use CALL statement.
Function	Return value can be thrown away.	Use CALL statement.
Function	Internal name is a legal HP Business BASIC/XL function name.	Call as a user-defined function is called.
Function	Internal name is not a legal HP Business BASIC/XL function name.	Call FNCALL function.

FNCALL is a predefined function that takes a function call as its parameter. Executing an FNCALL call is equivalent to executing the parameter (a function call). An internal function (a predefined function or function defined by the program) cannot be called with FNCALL. An external function with an illegal HP Business BASIC/XL function name must be called with FNCALL. An FNCALL call can appear wherever a user-defined function call can appear.

Examples

```
10 INTRINSIC ("Isubs") Sub1             !Declares intrinsic subprogram
15 CALL Sub1                            !Calls intrinsic Sub1
20 EXTERNAL Sub2                        !Declares external subprogram
25 CALL Sub2                            !Calls external Sub2
30 INTRINSIC Irr_result1                !Function with irrelevant result
35 CALL Irr_result1
40 EXTERNAL REAL Irr_result2            !Function with irrelevant result
45 CALL Irr_result2
50 INTRINSIC FNRead                     !Function with legal name
55 C$=FNRead
60 EXTERNAL REAL FNWrite ALIAS "Write"  !Function with legal name
65 Real1=FNWrite
70 EXTERNAL INTEGER Store (REAL X)      !Function with illegal name
71                                      !to show use of FNCALL
75 Int1=FNCALL(Store(Real1))
80 INTRINSIC Getfile ALIAS "Get_file"   !Function with illegal name
81                                      !aliased to legal name
85 IF FNCALL(Getfile("File2")) THEN CALL Sub1
99 END
```

External Parameter Type Correspondence

When a program calls an external routine, the types of the actual parameters must correspond to the types of the formal parameters.

When a program declares an external function that is not declared as INTRINSIC, the return type in the EXTERNAL statement must correspond to the return type in the function's original definition.

Table 3-33 shows the correspondence between parameter types in HP Business BASIC/XL, HP Pascal/XL, and HP C/XL.

Table 3-33. Parameter Type Correspondence

HP Business BASIC/XL Actual Parameter Type	**Formal Parameter Typed Declared in EXTERNAL Declaration**	**Formal Parameter Type in HP Business BASIC/XL**	**Formal Parameter Type in HP Pascal/XL**	**Formal Parameter Type in HP C/XL**
String$	String$	String$	STRING	Not supported
String$	BYTE STRING$	Not supported	Packed array of char	char
Any type except String$	BYTE	Not supported	Any type requiring exactly 8 bits of storage	char
SHORT INTEGER	SHORT INTEGER	SHORT INTEGER	SHORTINT	short
INTEGER	INTEGER	INTEGER	INTEGER	int
SHORTREAL	SHORTREAL	SHORTREAL	REAL	float
REAL	REAL	REAL	LONGREAL	double
SHORT DECIMAL	SHORT DECIMAL	SHORT DECIMAL	Not supported*	Not supported*
DECIMAL	DECIMAL	DECIMAL	Not supported*	Not supported*

Table 3-33 Note

* Decimal parameters can be passed to an external routine written in any language by defining an appropriate type in that language.

Parameter type correspondence for numeric arrays is the same as that for scalar numeric parameters. The corresponding formal parameter for string array parameters in the procedure or function header for the procedure or function must be a string array that conforms to the type expected by An HP Business BASIC/XL procedure or function.

All arrays are passed by reference.

Examples

If the external HP Pascal/XL function func is defined:

```
function func (c:  color;
               var s:  str5;
               var i1: int1;
               var i2: integer;
               r:  real;
               var l:  longreal):  real;
```

And the types color, str5, and int1 are defined:

```
color = (red,blue,yellow);
str5  = packed array [1..5] of char;
int1  = shortint;
int2  = integer;
```

Then the following EXTERNAL statement is correct:

```
100 EXTERNAL PASCAL SHORT REAL FNFunc ALIAS "func" &
    (BYTE VALUE C, BYTE S$, SHORT INTEGER I1, INTEGER I2, &
    SHORT REAL VALUE R, REAL L)
```

Chapter 4 Statements

Introduction

This chapter contains descriptions of each statement that can be used to form programs in HP Business BASIC/XL. The statements are arranged in alphabetical order. Each description contains the complete syntax of the statement, examples, and other necessary information.

ACCEPT

The ACCEPT statement obtains a string of characters from the designated input device without echoing those characters to the display as they are entered. If a string variable is included in the ACCEPT statement, the value of the string of characters is assigned to the string variable. The characters in the entered string must be from the ASCII or default foreign character set. Otherwise, the terminal beeps.

No line feed is generated following statement execution, so the cursor remains on the same line.

Syntax

```
                 [                          [separator     ]]
ACCEPT [str_var ] [[separator ] option_clause  [option_clause ]]...
```

Parameters

str_var The string variable that the input string is assigned to. Characters are assigned to the variable when you type RETURN. Characters, such as a comma or a double quote, are not considered to be a data item separator or terminator within the input string. An ACCEPT statement without a *str_var* discards the input.

option_clause One of the following:

{TIMEOUT [=] *timeout_num_expr* }
{ELAPSED [=] *elapsed_num_var* }
{CHARS [=] *chars_num_expr* }

timeout_num_ expr Numeric expression for the maximum amount of time, in seconds, allowed for you to enter input. The input time limit is determined as follows:

Value of *timeout_num_ expr*	Input Time Limit
Zero or less	Unlimited
In the range (0,255)	That number of seconds rounded to nearest second
Greater than 255	Set to 255 seconds

If input time is limited through the use of the TIMEOUT option, HP Business BASIC/XL transfers control to the next program statement when the time limit is exceeded without assigning a new value to the specified *str_var*.

elapsed_num_var A numeric variable that the time, in seconds, used to enter the input is returned to. If the TIMEOUT option is also specified, and that time limit is exceeded, *elapsed_num_var* is set to -256.

If the ELAPSED option is not selected, the elapsed time is not measured.

chars_num_expr A numeric expression that evaluates to the maximum number of characters that can be input. Typing this number of characters causes the generation of a carriage return and assignment of the value to the specified *str_var*. Then the program begins execution of the next statement in the program.

separator One of the following:

```
{WITH}
{,   }
{;   }
```

Each *option_clause* can occur only once in an ACCEPT statement.

Examples

The following examples show the use of the ACCEPT statement. Lines 10 - 60 will assign the input string to a string variable, whole lines 70 - 110 discard the input.

```
10  ACCEPT String_var1$
20  ACCEPT String_var2$, TIMEOUT Time_limit
30  ACCEPT String_var3$ WITH TIMEOUT=Time_limit
40  ACCEPT String_var4$ WITH TIMEOUT Time_limit, ELAPSED Elapsed_time
50  ACCEPT String_var5$, CHARS Num_chars, ELAPSED Elapsed_time
60  ACCEPT String_var6$, ELAPSED Elapsed_time, CHARS 5, TIMEOUT 3
70  ACCEPT
80  ACCEPT TIMEOUT 5
90  ACCEPT ELAPSED Elapsed_time
100 ACCEPT CHARS 1
110 ACCEPT TIMEOUT 1, CHARS 1
```

ADVANCE

The ADVANCE statement moves the datum pointer of a specified BASIC DATA file a given number of datum from its current position. Use of any other file type with this statement results in an error.

Syntax

```
                          [{,}                  ]
ADVANCE #fnum; num_expr  [{;} STATUS[=]num_var ]
```

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. *fnum* is a numeric expression that evaluates to a positive short integer.

num_expr A numeric expression that indicates the number of datum and the direction that the pointer will move. The absolute value of this expression is the number of datum in the file that the ADVANCE statement moves the datum pointer. If *num_expr* is positive, the datum pointer moves ahead. If *num_expr* is negative, the datum pointer moves back. Consider the first datum in the file to be labeled number one. If the current position in the file plus the value of *num_expr* is either less than zero or greater than the total number of datum in the BASIC DATA file, an end of file error occurs.

num_var *num_var* is a numeric variable that returns the status of the ADVANCE operation. The value assigned to *num_var* is zero if ADVANCE is successful. If either the beginning or the end-of-file marker is passed, the difference between *num_expr* and the number of items actually skipped prior to reaching the file delimiter is returned. Note that if *num_expr* is negative, the value returned to *num_var* in the event of trying to advance past the beginning of file marker is negative.

Examples

The following program shows the use of the ADVANCE statement. Line 15 positions the datum pointer at datum 1, that is, the first datum in the file. Line 20 advances that pointer 6 datums, to datum 7. Lines 30-40 read and print that record. Datum 7 is the first field in record 4. Line 50 positions the back at datum 4. (The READ in line 30 advanced the pointer to datum 8). Lines 60-70 read and print that datum.

```
>list
 !  ADVANCE
     5 DIM A$[30]
    10 ASSIGN #1 TO "Datafile"
    15 POSITION #1;BEGIN
    20 ADVANCE #1;6
    30 READ #1;A$,Rec_no
    40 PRINT A$,Rec_no
    50 ADVANCE #1;-4
    60 READ #1;Rec_no
    70 PRINT Rec_no
    80 ASSIGN * TO #1
>run
This is record number                    4
 2
>
```

ASSIGN

The ASSIGN statement opens a file (makes it accessible) or closes a file (makes it inaccessible) in the program executing the statement. The file is opened by HP Business BASIC/XL when the program assigns the file a file number. HP Business BASIC/XL uses the file number to identify the file for reading and writing information. The ASSIGN statement disassociates a file from its file number and closes the file. When HP Business BASIC/XL closes a file, it releases the buffer space that was allocated to it.

Syntax

To open a file:

```
       {fname  TO #fnum }                     [,RESTRICT[=]ioaccess ]
ASSIGN {#fnum  TO fname } [,STATUS[=]num_var ] [[,useraccess ]       ]
```

[,MASK[=]*str_expr*]

To close a file:

```
       {* TO #fnum }
ASSIGN {#fnum  TO *}
```

Parameters

fname A string literal or string expression that contains the file name. It must include the lockword used when the file was created, if any. This parameter can be back referenced to a file equation.

fnum The file number that HP Business BASIC/XL uses to identify the file. It evaluates to a positive short integer. If *fnum* is associated with another open file, the ASSIGN statement opening the file first closes the open file before opening the one specified by *fname*.

If you attempt to close an already closed *fnum*, the ASSIGN statement does nothing.

num_var A variable that returns the status of the ASSIGN statement. The ASSIGN statement sets the value of this variable to zero if it opens the file successfully; otherwise, it sets it to a nonzero value.

A nonzero value represents the file error code returned by the file subsystem of the MPE XL operating system. The error number can be translated to an MPE XL file system error message by looking up the table of file system error codes in the *MPE XL Intrinsics Reference Manual* under the FCHECK intrinsic.

ioaccess If a file is opened by a program, the *ioaccess* specification determines how the program can access the file. The value of *ioaccess* is one of the following keywords:

READ The program can read from the file, but cannot write to it.

WRITE The program can write to the file, but cannot read from it.

APPEND The program can perform sequential writes to the file starting after the last record. It cannot read from the file or perform direct writes to the file.

READWRITE The program can read from and write to the file. This is the default I/O access if the RESTRICT option is not specified.

useraccess If a file is open to one program, *useraccess* determines how other programs can access the file. It also determines whether the program that opened the file can open it again without closing it first. The value of *useraccess* is one of the following keywords:

EXCLUSIVE Other programs cannot access the file. The program that opened it must close it before opening it again. The sequence:

```
ASSIGN fname  TO #fnum1,RESTRICT=READ,
  EXCLUSIVE
ASSIGN fname  TO #fnum2
```

is illegal. The sequence must be:

```
ASSIGN fname  TO #fnum1,RESTRICT=READ,
  EXCLUSIVE
ASSIGN * TO fnum1
ASSIGN fname  TO #fnum2
```

SINGLEUSER Other programs cannot write to the file, but the program that opened it can open it again without closing it first. The sequence:

```
ASSIGN fname  TO #fnum1,RESTRICT=READ,
  SINGLEUSER
ASSIGN fname  TO #fnum2
```

is legal; it opens the file *fname* twice in the same program, at the same time. SINGLEUSER is the default if this parameter is not specified.

You must have LOCK capabilities at both the account and group level in order to open the file multiple times. If you do not have those capabilities, then the default access is EXCLUSIVE.

SHARED Other programs can access the file.

SHAREREAD — Other programs can read the file, but cannot write to it.

str_expr — A string expression that evaluates to a string with a length of six characters. The string serves as a mask used to scramble and unscramble file data, excluding format words, EOR marks, and EOF marks. If a mask is specified the first time a file is assigned, the same mask must be specified each time the file is assigned; otherwise, the data cannot be properly unscrambled.

Examples

The following examples show the use of the ASSIGN statement to open and close files. Line 30 assigns a file with read access, allowing other programs to use it. File1 also has a mask. Line 40 assigns a file with append access. Line 50 assigns a file with readwrite access (default). Line 60 assigns a file with write access, allowing other programs to read it, and has a mask. Line 70 assigns the file with readwrite access and line 80 assigns the file for read access, allowing no one else to access the program. Line 90 assigns the file using a back referenced file equation.

```
     10 ASSIGN * TO #1    !Closes file designated as #1
     20 ASSIGN #2 TO *    !Closes file designated as #2
     30 ASSIGN "File1" TO #3,STATUS=S,RESTRICT=READ,SHARED,MASK="ScRmBL"
     40 ASSIGN #4 TO "File2",STATUS X,RESTRICT APPEND,SINGLEUSER
     50 ASSIGN "File3.lab" TO #5,STATUS Open
     60 ASSIGN "F4.mktg.hp" TO #6,RESTRICT=WRITE,SHAREREAD,MASK="zzypdq"
     70 ASSIGN #7 TO "File5",RESTRICT READWRITE
     80 ASSIGN #8 TO "File6",RESTRICT=READ,EXCLUSIVE
     90 ASSIGN "*file3" to #9
```

BEEP

When HP Business BASIC/XL is running interactively, the BEEP statement sends a CONTROL G (ASCII character 7) to the terminal, causing it to beep. When HP Business BASIC/XL is running in a job stream, the BEEP statement does nothing.

Syntax

BEEP

Example

When the following program is run, the terminal will beep once.

```
     10 BEEP
```

BEGIN REPORT

The BEGIN REPORT statement activates a report, but does not start report output. The report description is verified and some Report Writer expressions are evaluated. The report is not activated unless BEGIN REPORT executes correctly. This statement can not appear within a report description.

Syntax

BEGIN REPORT *line_id*

Parameters

line_id — The line number or line label of the REPORT HEADER for the report to use. The line indicated can be a comment, provided that only comments occur between the given line and the REPORT HEADER statement.

Examples

```
     100 BEGIN REPORT 500
     100 BEGIN REPORT Report_1
```

An error occurs if a report is active when BEGIN REPORT executes. This statement searches for a REPORT HEADER statement starting with the line indicated. Only comments can occur between the given line and the REPORT HEADER statement.

Once the REPORT HEADER is found, the Report Writer scans the report description. The report scan uses two passes. The first pass determines what sections are valid, and then the second pass evaluates necessary expressions. The following actions take place during the scanning process:

First Pass:

- Section statements are made busy. In addition, the TOTALS, GRAND TOTALS, PRINT DETAIL IF, BREAK IF, and BREAK WHEN statements are made busy. Busy lines cannot be deleted or modified (See "Busy Lines and Subunits" in chapter 2).

- All level expressions are evaluated. This affects HEADER, TRAILER, BREAK IF, and BREAK WHEN statements. TOTALS statements are indirectly affected, as they are ignored if the last HEADER or TRAILER section has a level expression equal to zero.

Second Pass:

- The PAGE LENGTH, LEFT MARGIN, PAUSE EVERY, SUPPRESS AT, and SUPPRESS FOR statements are evaluated.

- The TOTALS and GRAND TOTALS are set to zero.

- BREAK IF and BREAK WHEN statements are evaluated. This includes evaluation of the control expressions and the BY clause values. The OLDCV and OLDCV$ values are initialized. For BREAK WHEN statements with a BY clause, the initial limit and multiple values are set up.

- The WITH clauses of the PAGE HEADER and PAGE TRAILER sections are evaluated if present. This determines the usable page size. A check is made to ensure that there are lines left on the page after the PAGE sections are counted.

If any error occurs during BEGIN REPORT, the report is not activated.

BEGIN TRANSACTION

The BEGIN TRANSACTION statement defines the beginning of a sequence of TurboIMAGE procedure calls that are to be regarded as a single logical transaction for the purposes of logging and recovery. The MSG parameter allows you to log additional information in the log file. TurboIMAGE logs database transactions on the transaction log file if any of the following are true:

- The database is open in one of the following modes:
  - Modify with enforced locking.
  - Update.
  - Exclusive modify.
  - Modify.

- The database is enabled for logging by the database administrator.

- The system console has enabled a logging process.

The transaction log file is explained in the *TurboIMAGE/XL Database Management System*.

Syntax

BEGIN TRANSACTION *dbname* $, MSG[=]*str_expr*, [, STATUS[=]*status_array* (*)]

Parameters

dbname $ A string variable, whose value is a TurboIMAGE database name. This must be the *dbname* $ returned by a successful DBOPEN statement.

str_expr A string of ASCII characters of up to 512 characters in length to be written as part of the BEGIN TRANSACTION log record.

status_array A 10-element short integer array to which TurboIMAGE returns any error codes or other status information. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

The following shows the use of the BEGIN TRANSACTION statement.

```
     100 BEGIN TRANSACTION Db$,MSG=Message$,STATUS=S(*)
     110 BEGIN TRANSACTION Db$,MSG Message$,STATUS S(*)
```

BREAK IF

The BREAK IF statement provides a general mechanism for automatic summary level breaks. The DETAIL LINE statement causes the execution of the statement. If the break condition is true, all summary levels from the BREAK level and up are triggered. This causes TRAILER and HEADER sections to be printed. The BREAK IF statement can occur anywhere in the report description. There can only be one BREAK statement per summary level, either BREAK IF or BREAK WHEN. There is no BREAK statement for the report level.

Syntax

BREAK *break_level* IF *boolean_expr*

Parameters

break_level The summary level that is triggered if the break condition is satisfied. This value must be in the range [0, 9]; a level of zero causes the statement to be ignored.

boolean_expr An expression that evaluates to a numeric value. If the value is nonzero, a break is triggered at this level.

Examples

The following examples show the use of the BREAK IF statement.

```
     100 BREAK 3 IF Abc > Def or Abc < Ghi
     100 BREAK 5 IF Last_name$<> Old_last$ AND &
         First_name$ <> Old_first$
```

The BEGIN REPORT statement sets all BREAK IF statements to busy, unless the *break_level* is zero. When the report ends, the lines are no longer busy. The level expression is evaluated only during BEGIN REPORT. The Boolean expression is evaluated during DETAIL LINE, TRIGGER BREAK, and BEGIN REPORT.

The DETAIL LINE statement checks all BREAK statements when its *total flag* is nonzero. All BREAK statements are checked in this case. BREAK statements are evaluated from level one to level nine, in order. For BREAK IF, the Boolean expression is evaluated. If the expression is true (nonzero), a break is triggered at the given level. The value of the LASTBREAK built-in function is changed immediately. DETAIL LINE remembers the lowest broken level and triggers all the TRAILER and HEADER sections from that level through nine.

BREAK WHEN

The BREAK WHEN statement provides a general mechanism for automatic summary level breaks. The DETAIL LINE statement causes the execution of the statement. If the break condition is true, all summary levels from the BREAK level and up are triggered. This causes TRAILER and HEADER sections to be printed.

The BREAK WHEN statement can occur anywhere in the report description. There can only be one BREAK statement per summary level, either BREAK IF or BREAK WHEN. There is no BREAK statement for the report level.

Syntax

BREAK *break_level* WHEN *control_expr* [CHANGES]

BREAK *break_level* WHEN *num_ctl_expr* [CHANGES] BY *num_by_expr*

Parameters

break_level	The summary level triggered if the break condition is satisfied. This value must be in the range [0, 9]; a level of zero causes the statement to be ignored.
control_expr *num_ctl_expr*	A numeric or string expression. When BREAK WHEN is evaluated, the value of this expression is recorded. Then this value is compared at the next DETAIL LINE to see if any change has occurred. A break occurs occur if a change takes place.
num_by_expr	A numeric expression indicating how much the control expression must change before a break occurs. See below for exact details about how this works. The control expression must be numeric to use a BY clause.

Examples

```
100 BREAK 1 WHEN Salesman$ CHANGES
100 BREAK 3 WHEN Region CHANGES
100 BREAK N WHEN Product CHANGES BY Base_product_num
```

The BEGIN REPORT statement sets all BREAK WHEN statements to busy, unless the *break_level* is zero. When the report ends, the lines are no longer busy. The level expression is evaluated only during BEGIN REPORT. In addition, the BY clause value is evaluated only during BEGIN REPORT. The control expression is evaluated during DETAIL LINE, TRIGGER BREAK, and BEGIN REPORT.

The DETAIL LINE statement checks all BREAK statements when its *total-flag* is nonzero. The BREAK statements are evaluated in summary level order, from one to nine. The control expression of the BREAK WHEN statement is evaluated at this time. Conditions for satisfying a break are given below. The LASTBREAK function is set as soon as a break condition is found. DETAIL LINE remembers the lowest level broken and triggers the TRAILER and HEADER sections from that level through nine. First, the TRAILERS are triggered from the highest existing level descending to the lowest level broken. Next, the HEADERS are triggered from the lowest level broken up to the highest existing level.

When BEGIN REPORT executes, the level expressions for all BREAK statements are evaluated first. A second pass is made for BREAK WHEN statements. During this pass, the control expression is evaluated and the result put into OLDCV (or OLDCV$) for the break level. Then, if present, the BY clause is evaluated and its value recorded. This value is used when a break occurs at the current level.

The TRIGGER BREAK statement also evaluates the BREAK WHEN control expressions for all broken levels. This is to update the OLDCV and OLDCV$ values for all broken levels. These evaluations are done before the actual break occurs.

All OLDCV values are updated when a break occurs. The values are updated between the printing of the TRAILER sections and the HEADER sections.

Satisfying a BREAK WHEN Condition

There are two forms of the BREAK WHEN statement; both have a *control* expression. The statements differ in what changes can be specified for the control expression.

String Control Variables. When the report is activated via BEGIN REPORT, the value of the control expression is recorded. With each DETAIL LINE, the current value of the control expression is compared to the recorded value in OLDCV$. If the two values are not the same, the break level is triggered.

After any break at the specified level, the new value of the control expression is recorded in place of the old value, OLDCV$. This process takes place after all trailers have been output, but before headers are printed.

Examples

```
        BREAK 3 WHEN Sales_Office$ CHANGES
```

In this example, a break at level 3 occurs whenever the control expression Sales_Office$ changes value.

Numeric Control Variables. Numeric control expressions have an optional BY clause in the BREAK WHEN statement. If the BY clause is not present or evaluates to zero, the statement works exactly as it does with a string control expression. That is, a break is triggered whenever the control expression in OLDCV changes value.

The BY clause establishes a limit value that the control expression must exceed before a break occurs. The numeric expression in the BY clause determines the increment by which the limit changes after a break. However, the limit is NOT set by adding the BY expression to the control expression.

When a BEGIN REPORT executes, the control expression is recorded and the BY clause is evaluated. At this time, a break limit is set up as well. This limit is set up in the following manner:

- If the BY expression is positive, the limit is set to the multiple of the BY clause closest to, but still greater than, the control expression.

- If the BY expression is negative, the limit is set to the multiple of the BY clause closest to, but still less than, the control expression.

At each DETAIL LINE, the control expression is compared to the limit value. If the control expression is greater than or equal to the limit (less than or equal if BY was negative), a break is triggered. After the trailers print, but before the headers are output, a new limit is established using the rules above. The BY clause is not reevaluated; only the limit is changed.

The break limit is reevaluated at any break at the BREAK WHEN level. This can be caused by breaking at this level or a lower level from a DETAIL LINE or a TRIGGER BREAK statement.

Examples

```
     BREAK N WHEN Product_no CHANGES
     BREAK 8 WHEN Profits CHANGES BY 100000
     BREAK A(1) WHEN Sales CHANGES BY -50
```

In the first example, a break takes place when the control PRODUCT_NO changes value. It does not matter how much it changes, nor whether it gets larger or smaller.

In the second example, a break occurs when the variable PROFITS exceeds a multiple of one hundred thousand. Assuming that PROFITS has an initial value of 50000, the first break limit is 100000. If PROFITS then changes to 235000, break level 8 is triggered; the next break limit is set to 300000, the next multiple larger than PROFITS.

The third example is similar to the second, except that the BY clause is negative. If SALES has an initial value of 480, the break limit is set to 450 (not 430). If SALES gets larger, no break ever occurs. Only when SALES becomes 450 or less does the break occur. For example, if SALES drops to 220, a break occurs and the new limit is set to 200. It is important to remember that this is the multiple of a BY clause.

Control Expression Storage Requirements

The control expression for BREAK WHEN statements is kept by the OLDCV function. The data space required to contain this expression is determined during BEGIN REPORT, when the values are first examined. The space for OLDCV and OLDCV$ are allocated as follows:

* For numeric variables and array elements, space is allocated based on the data type of the variable. There should be no way to get an error with OLDCV in this case.

* For other numeric space is allocated based on the data type returned when the expression is initially examined. Thus, if BEGIN REPORT finds an INTEGER expression, space is allocated for an INTEGER. If the expression later returns a REAL outside the INTEGER range, an error occurs.

* For other string variables and array elements, space for OLDCV$ is allocated based on the maximum length of the string variable. Thus, the value of the string may be shorter than the space allocated.

* For string expressions including substrings, space for OLDCV$ is allocated based on the actual length of the evaluated expression. This could cause a string overflow if a later evaluation returns a longer string.

A BY clause stores two values: the BY value itself, and the limit, which causes a break. Both of these values are stored in REAL or DECIMAL, depending on the option in the report subunit.

CALL

The CALL statement transfers control from the program unit that the statement occurs in to a specified subprogram. The subprogram that control is transferred to must be defined in the program or a run-time error occurs.

The CALL statement can also transfer control to a user-defined multi-line function. When used in this manner, the function is actually called as a subprogram. The value returned by the function is discarded.

Syntax

CALL *sub_name* [(*a_param* [, *a_param*]...)]

Parameters

sub_name Subprogram that control is transferred to.

a_param Actual parameter - a value, a variable or an expression, This parameter has a value of the appropriate type to be assigned to the corresponding formal parameter in the SUB statement that begins the subprogram or multi-line function *sub_name*.

The CALL statement assigns the values of the actual parameters to the corresponding formal parameters and transfers control to the subprogram.

Execution of a SUBEXIT or SUBEND statement in the subprogram returns control to the statement following the CALL statement provided there are no pending softkey interrupt requests.

Example

```
      10 READ A,B$
      15 DATA 1,"Sample"
      20 CALL Subrtn(A,B$)                !Control goes to line 100
```

```
 30 PRINT "Done"
 99 END
100 SUBPROGRAM Subrtn(Number,String$)
110   IF Number<1 THEN SUBEXIT
120   FOR I=1 TO Number
130     PRINT RPT$(String$,Number)
140   NEXT I
150 SUBEND                            !Returns control to line 30
```

If a program has more than one subunit with the same name, the CALL statement calls the first one that it finds. The following is the search order:

1. Single-line function
2. Local external or intrinsic function
3. Internal multi-line function (one defined by the program)
4. Global external or intrinsic subunit

If the program is using softkey handling, the program checks for the key after the subend statement, but before execution of the next main program line. Thus, control can not go to the next line following the CALL, but to a line specified by an ON KEY statement.

CASE and CASE ELSE

The CASE and CASE ELSE statements are part of the SELECT construct. Refer to the SELECT statement for more information.

CATALOG

The CATALOG statement displays directory information about specified files. The format of the directory information displayed depends on the operating system.

Syntax

```
{CATALOG}
{CAT    }[option_list ]
```

Parameters

option_list *option* , [*option*] [, *option*]

option
{FILE[=] *filename_or_fileset* }
{TYPE[=] *file_code* }
{COUNT[=] *num_var* }

Each option can occur only once in a CATALOG statement.

filename_or_ fileset An *fname* as described chapter 6 or a set of files specified by incorporating "wild card" characters. This is the set of files that directory information is displayed for. Wild card characters represent a set of characters and are operating system dependent. On the HP3000 operating with the MPE XL operating system, information on the use of wild card characters can be obtained by typing ":help listf parms". For example, the "@" symbol specifies zero or more alphanumeric characters. Thus, the filename, "ab@" specifies the file "ab" and all additional files that have the "ab" prefix. If the FILE option is not selected, the default value for a *filename_or_fileset* is the user's group and account or the group and account specified in the most recent FILES ARE IN statement.

file_code A string expression of up to five characters in length specifying the type of file and indicated by a file code. If the TYPE option is selected, then directory information is displayed about only those files with the designated *file_code*. The values of *file-code* available to the user are operating system dependent. The values for MPE XL are available in the *MPE XL Commands*

Reference Manual under the BUILD command's file code mnemonics. Valid values include "BSVXL", "BDTXL", "JL", or "1200". If the TYPE option is not specified, then TYPE information is not used as a selection criteria for determining which file's directory information is displayed.

num_var A numeric variable to which the total number of files found is returned.

The CATALOG statement lists its information on the standard list device or on the device specified by the most recently executed SEND SYSTEM OUTPUT TO statement. Table 4-1 shows how specifying *file_code* or *fname*, both, or neither, determines CATALOG statement output on the HP 3000.

Table 4-1. CATALOG Statement Output

FILE and TYPE option selected	**Files That the CATALOG Statement Lists Directory Information For**
Neither	All files.
TYPE only	Files that have an MPE file code matching *file_code*.
FILE only	Files that match the *filename_or_fileset* specification.
FILE and TYPE	Files that match the *filename_or_fileset* specification that also have the MPE file code matching *file_code*.

Examples

```
         CAT
         CAT file1
         CAT FILE ="File1"
         CAT TYPE = "BDTXL"

     10  CAT FILE ="@BB@",TYPE="BSVXL",COUNT=Count
     20  CAT FILE ="@.PUB.SYS",COUNT=System_count
```

CAUSE ERROR

The CAUSE ERROR statement causes an HP Business BASIC/XL program to behave as though the specified error had occurred. If an ON ERROR statement has been issued, then the user-specified recovery action is executed.

Syntax

CAUSE ERROR *error_number*

Parameters

error_number A numeric value that is the same as an HP Business BASIC/XL error number.

Example

```
     10 ON ERROR GOTO 200
     20 CAUSE ERROR 2      !This causes an error 2, memory overflow
     30                    !Control transfers to line 200
     .
     .
     .
     200 Error handler:    !Start of error handling routine
     .
     .
     .
```

CLEAR FORM

The CLEAR FORM statement sets the contents of all fields on the currently displayed form to blanks or another default value. This statement is used with JOINFORM as well as VPLUS, but the DEFAULT clause is ignored when using JOINFORM. If there is no active form, executing a CLEAR form causes a run-time error.

Syntax

CLEAR FORM [*default_clause*]

Parameters

default_clause The optional keyword DEFAULT assigns the initial values from the VPLUS form file to each field. The correct syntax is:

```
       {DEFAULT }
[WITH] {DEFAULTS}
```

Examples

The following examples show the use of the CLEAR FORM statement.

```
     500 CLEAR FORM
     510 CLEAR FORM DEFAULT
     530 CLEAR FORM WITH DEFAULT
```

CLOSE FORM

CLOSE FORM closes the currently active form. This statement is used with both JOINFORM and VPLUS.

When an VPLUS form is active, the form file is also closed. CLOSE FORM does not close JOINFORM files. When execution of the CLOSE FORM statement is complete, the cursor is at the top of display memory and memory lock, format lock, and block mode are off.

When CLEARALL, CLEARREST, or REMAIN are not specified, the form is closed by deleting the individual lines of the form. The contents of display memory above and below the form are not deleted. When the form is deleted, the contents of display memory that follows the form are scrolled into the area of display memory that previously contained the form.

Syntax

```
           [{;}         ]
           [{,} CLEARALL]
CLOSE FORM [CLEARREST   ]
           [REMAIN      ]
```

Parameters

CLEARALL Specifies that the form should be deleted from the screen by placing the cursor at the home position and clearing all of display memory.

CLEARREST Specifies that the form should be deleted from the screen by placing the cursor at the first line of the form, and clearing display memory from that position to the end. The area of display memory above the form is not affected.

REMAIN — Specifies that the form should be left on the screen. It is unprotected after it is closed.

Examples

The following statements show the use of the CLOSE FORM statement.

```
     200 CLOSE FORM             !FORM is cleared from the screen
     210 CLOSE FORM ;REMAIN     !FORM is left on the screen
     220 CLOSE FORM ,REMAIN     !FORM is left on the screen
     230 CLOSE FORM             !FORM is cleared from the screen
     240 CLOSE FORM ;CLEARREST  !Display memory is cleared from the
     245                        !first line of the form to the end
     250 CLOSE FORM :CLEARALL   !All of display memory is cleared
```

If REMAIN is entered preceded by a "," HP Business BASIC/XL will replace it with a ";".

COM

The COM statement declares a common area. The common area is a global data area that is first declared in the main program. One or more variables can be declared in each declared common area. Each common variable in a COM area declared in the main program unit is accessible within the main program unit and in all called procedures or functions that declare the common area in which the variable occurs. Unlike local variables, the value of a common variable is retained following the exit from a called procedure or function. A new common area can also be declared in a called procedure or function if the GLOBAL OPTION SUBPROGRAM NEWCOM or GLOBAL OPTION MAIN NEWCOM is used in the main program area preceding the procedure or function. New common areas declared in these routines are allocated when first encountered during program execution and can be referenced in any routine called from that routine. The common area is deallocated when the routine in which it was allocated completes execution.

Syntax

COM [/*identifier* /] *type_list* [, *type_list*]...

Parameters

identifier — Name of common area. If an identifier is specified, the declared common area is "labeled" with the name of the identifier. If an identifier is not specified, then the common area is referred to as the unnamed common area. You can have a maximum of ten named commons and one unnamed common.

COM statements in different program units with the same label refer to the same common area and unnamed COM statements refer to the unnamed common area. This identifier is truncated to eight characters.

type_list —
{[*type*]*num_com_item* [, *num_com_item*]...}
{*non_num_com_item* }

type — One of the following:
SHORT INTEGER
INTEGER
SHORT DECIMAL
DECIMAL
SHORT REAL
SHORT
REAL
unspecified

If a type is not specified, implicit declaration rules apply. After *type*, each *num_com_item* is of that type until another *type* or a *non_num_com_item* appears.

num_com_item

Numeric variable declaration (for a scalar or array variable).

If the COM statement is in a subunit, *num_com_item* must represent a numeric array with the abbreviation

identifier ([*[,*]...])

with one asterisk per dimension or without asterisks. Not using asterisks specifies any number of dimensions. Either format is legal, but the format without asterisks is noncompilable. To facilitate program documentation, numeric values can be used in place of the asterisks, but these values are ignored during program execution.

non_num_com_item

String variable declaration (for a scalar or array variable) or file designator. If maximum length is not specified for a string variable, it is 18.

Maximum length is not specified if the COM statement is in a subunit.

If the COM statement is in a subunit, *non_num_com_item* must represent a string array with the abbreviation

identifier $([*[,*]...])

with one asterisk per dimension or without asterisks. Not using asterisks specifies any number of dimensions. Either format is legal, but the format without asterisks is noncompilable. The maximum length of each element is the same as declared in the main program. To facilitate program documentation, numeric values can be used in place of the asterisks, but these values are ignored during program execution.

The syntax of an HP Business BASIC/XL file number is:

#*numeric_literal*

numeric_literal is a positive integer in the range [1, 32767]. The file designated by the actual parameter in the COM area is referenced by #*numeric_literal* within the subunit declaring the com area. If the main procedure or function in which the HP Business BASIC/XL file number occurs is to be compiled, the numeric_literal must be a positive integer in the range [1, 16].

To make it easier to copy com area to subunits, the declaration of the com area in the main program can be copied directly to the subunit. The numeric values specifying the range of subscripts for a dimension for either numeric or string array variables do not need to be changed to asterisks. However, HP Business BASIC/XL interprets the values as place holders for each dimension. The dimension information in the common area in the program unit in which the common area is declared, usually the main, is used to determine the array dimensionality and the subscript bounds.

Example 1: Common Declarations

```
     10 COM INTEGER A,B, REAL C,D, A$[7], P,Q, DECIMAL X,Y,Z, #2
```

Variable(s)	**Type**

A,B	Integer
C,D	Real
A$	String with maximum length of 7 characters
P,Q	Default numeric type
X,Y,Z	Decimal
#2	File designator

```
     100 COM N,S$,N_array(1:5),S_array$(1:2,1:4)[6]
```

Variable(s)	**Type**
N	Default numeric type
S$	String with default maximum length (18)
N_array	Array of default numeric type
S_array$	Array of strings with maximum length of 6

Example 2: Concatenation of Common Variable Lists If two COM statements in the same program unit have the same area name, their variable lists are concatenated.

Lines 200 and 210 are equivalent to line 300. Common area Area 3 contains the same variables whether the program unit contains lines 200 and 210 or line 300.

```
     200 COM /Area3/ SHORT INTEGER J,K,L
     210 COM /Area3/ REAL M,N,O, DECIMAL P,Q

     300 COM /Area3/ SHORT INTEGER J,K,L, REAL N,N,O, DECIMAL P,Q
```

Example 3: Correspondence of Common Variables in Main and Subunit When two program units declare the same common area, corresponding common items refer to the same entities. The entities (for example, variables or files) can have different names in different program units, however, because the different names refer to the same areas in memory, they must have the following:

* The same type.
* The same number of dimensions.

If the main program unit contains the statements:

```
     10 COM /Area4/ REAL A,B$[60], INTEGER C, #8
     20 COM /Area4/ DECIMAL E(1:25,1:50), F$(0:4,0:4,0:4)[12]
```

Then a subunit can contain the statements:

```
     350 COM /Area4/ REAL X,Y$
     360 COM /Area4/ INTEGER C, #10, DECIMAL E()
     370 COM /Area4/ F$(*,*,*)
```

Corresponding variables are compatible:

Main Program Unit	**Program Subunit**
REAL A	REAL X
B$[60]	Y$
INTEGER C	INTEGER C
# 8	# 10
DECIMAL E(1:25,1:50)	DECIMAL E()
F$(0:4,0:4,0:4)[12]	F$(*,*,*)

If the main program unit assigns a value to the variable that it calls A, and then calls the program subunit, the value of X is the same as that assigned to A in the main because A and X are different names for the same variable.

If the main program unit contains the statements:

```
     10 COM /Area4/ SHORT REAL A, B$[60], INTEGER C, #15
     20 COM /Area4/ DECIMAL E, F$(0:4,0:4,0:4)[12]
```

Then a procedure in the same program cannot contain the statements:

```
     450 COM /Area4/ REAL Num, String$, SHORT INTEGER D
     460 COM /Area4/ Q$, DECIMAL A(*,*), B()
```

The conflict in type and /or dimension for each variable is:

Main Program Unit	**Program Subunit**
SHORT REAL A	REAL Num
INTEGER C	SHORT INTEGER D
#15	Q$
DECIMAL E	DECIMAL A(*,*)
F$(0:4,0:4,0:4)[12]	B()

Within a program unit, the following variables cannot have the same name:

 * A common scalar variable and a local scalar variable.
 * A common array variable and a local array variable.

In most cases, the main program declares every common area that the program uses, whether the main program uses it or not. Before HP Business BASIC/XL executes the main program unit, it allocates space for all common variables, using the default numeric type and default lower bound set by GLOBAL OPTION statements.

A procedure need only declare the common areas that it uses. A procedure can declare all or part of the defined common area (starting at the beginning), but cannot add items to it.

Exceptions to the foregoing occur if the NEWCOM suboption of the MAIN/SUBPROGRAM global option is used. If NEWCOM is specified in the current procedure, then, when the procedure begins execution, new common areas in the subunit that are not declared in the main program are allocated space. Also, the space for common areas declared in the main program that are not used in the procedure is deallocated.

Examples

If the main program unit contains the COM statements:

```
     10 COM /Area5/ INTEGER A,B, REAL C,D, DECIMAL E,F
     20 COM /Area5/ A$,B$,C$
```

Then a procedure can declare all of Area5:

```
     100 COM /Area5/ INTEGER X,Y
     110 COM /Area5/ REAL R1,R2
     120 COM /Area5/ DECIMAL D1,D2
     130 COM /Area5/ A$, B$, C$
```

Or part of Area5, starting at the beginning:

```
     200 COM /Area5/ INTEGER Part1,Part2, REAL Part3, Part4
     210 COM /Area5/ DECIMAL D
```

But a procedure cannot omit the beginning of Area5:

```
     300 COM /Area5/ REAL P,Q
     310 COM /Area5/ DECIMAL D1,D2
```

And it cannot add to Area5:

```
     400 COM /Area5/ INTEGER Int1,Int2, REAL Real1,Real2
     410 COM /Area5/ DECIMAL Dec1, Dec2, A$, B$, C$
     420 COM /Area5/ SHORT Sh1, Sh2, Sh3
```

Common variables are initialized as explained in "Initializing Variables," in chapter 3.

COMMAND

The COMMAND statement executes a string expression as if its value were a program line.

Syntax

COMMAND *str_expr*

Parameters

str_expr Its value must be an executable statement with 500 or fewer characters. If it is not, an error occurs.

The executable statement cannot be any of the following:

- A command-only statement (for example, LIST).
- A program-only statement (for example, INPUT).
- The COMMAND statement.
- Any statement that defines a construct (for example: WHILE, END WHILE, or REPEAT).

The statement cannot be a declaration statement, because declaration statements are not executable.

Examples

```
100 READ I
105 DATA 1
110 IF I THEN
120   Routine$="Routine1"
130 ELSE
140   Routine$="Routine2"
150 ENDIF
160 COMMAND "GOSUB " + Routine$   !Issue the GOSUB statement
170 STOP
180 Routine1: I=I+(2*I)+(3*I)
190 RETURN
200 Routine2: I=I*(2+I)*(3+I)
210 RETURN
220 END
```

The COMMAND statement is not compilable.

CONVERT

This statement converts a string into a number, with an option for specifying a line number or label to branch to if an error occurs. Similar to the VAL function, the CONVERT statement translates a string of ASCII characters into a numeric value that is assigned to a supplied numeric variable. Unlike VAL, the CONVERT statement converts the numeric value to the type of the numeric variable supplied. If a line label or number is supplied and an error occurs, CONVERT branches to the designated line without requiring an ON ERROR statement.

Syntax

```
                 {TO}        [{,}        ]
CONVERT str_expr  {, } num_var  [{;} line_ref ]
                 {; }
```

Parameters

Str_expr A string, substring, or other string expression.

The string representation is CONVERTed using the following syntax:

```
          ['+']
[space]...['-'] num_char  [num_char ]...['.']

             [{'e'}['+']                  ]
[num_char ]...[{'l'}['-']num_char [num_char ]...]
             [{'d'}                       ]
```

num_char is a numeric character in the range [0, 9]. If a syntax error occurs before conversion of the first numeric character, error 32 is generated if *line_ref* is not supplied. Once the first numeric character has been converted, if a syntax error occurs, then the value assigned to *num_var* is the value converted immediately prior to the syntax error. An HP Business BASIC/XL error number is not generated for this condition. The string is deblanked before it is converted.

num_var A numeric variable.

line_ref A line label or line number that is in the same procedure as the CONVERT statement. Specification of a *line_ref* supersedes the error handling action specified in an ON ERROR statement.

Examples

```
10  A$="123"
20  CONVERT A$ TO A     !A is now 123, and its type is the default numeric.
99  END
```

```
10  A$="123abc"
20  CONVERT A$ TO A     !A is now 123, and its type is the default numeric.
99  END
```

COPY ALL OUTPUT TO

The COPY ALL OUTPUT TO statement copies interpreter and program output on the file or device specified by *dev_spec*. If the *dev_spec* is a disk file that already exists, additional information is appended to the file.

Syntax

COPY ALL OUTPUT [TO] *dev_spec*

Parameters

dev_spec A device specification statement. It includes a destination device and can also have the MARGIN and FIELD keywords. If the device is a disk file, you can also specify a FILESIZE. See chapter 6 for more information.

Examples

The examples below show some of the different ways to combine parameters in the COPY ALL OUTPUT TO statement.

```
100 COPY ALL OUTPUT TO "MYFILE"
110 COPY ALL OUTPUT TO "LISTFILE", MARGIN 20
120 COPY ALL OUTPUT TO Filename$, FILESIZE Num_records
130 COPY ALL OUTPUT TO Filename$, FILESIZE Num_records, MARGIN Z, FIELD N+1
140 COPY ALL OUTPUT TO A$+B$, FIELD 10
150 COPY ALL OUTPUT TO DISPLAY, FIELD X+2, MARGIN 10
160 COPY ALL OUTPUT TO NULL
170 COPY ALL OUTPUT TO PRINTER
```

The SEND OUTPUT TO statement overrides the COPY ALL OUTPUT TO statement. If a program contains both statements, then PRINT statement output is displayed only on the device that the SEND OUTPUT TO statement specifies.

Between the initiation of report writer output with the DETAIL LINE, TRIGGER BREAK, TRIGGER PAGE BREAK or END REPORT statement and termination of the report, execution of a COPY ALL OUTPUT TO statement generates an error.

COPYFILE

The COPYFILE statement copies one file to another file or to a device referenced by a file operation. It does not affect the original file.

Syntax

```
                            [            {,}                 ]
COPYFILE fname1  [TO fname2 ] [, lock_word  {;} STATUS[=]num_var ]
```

Parameters

fname1

fname of the file to be copied.

fname2

fname of the copy of the file that is to be created. The name must not be the name of a file that already exists, otherwise a duplicate file name error occurs. The COPYFILE statement creates a file with this name and gives it the attributes of the original file. If this parameter is not specified, the COPYFILE statement copies the original file to the standard list device or to the device specified by the most recently executed SEND OUTPUT TO statement.

Similar to the *formal_designator* described in the Device Specification Syntax in chapter 6, *fname2* can also be one of "$STDLIST", "$NULL" or "*_fname_ ". The *_fname_ syntax is used to reference device files that have been previously defined using file equations.

lock_word

String expression that evaluates to the lockword for *fname1*. It is required if *fname1* has a lockword.

The lockword is not added to the copied file.

num_var

The COPYFILE statement assigns zero to *num_var* if the copy is completed successfully; otherwise, the value is set to a nonzero value.

Examples

```
100 CREATE ASCII "File1", RECSIZE=100, FILESIZE=1200
120 CREATE ASCII "File3", RECSIZE=100, FILESIZE=2400
130 PROTECT "File1", "zzxyz"            !add lockword "zzxyz" to File1
140 PROTECT "File3", "pqpqqp"           !add lockword "pqpqqp" to File3
150 COPYFILE "File1" TO "File2", "zzxyz" !lockword required for access
155                                     !to File1 - File2 is created
160 COPYFILE "File2" TO "File4"         !File2 has no lockword.
170 COPYFILE "File2" TO "File6"
180 COPYFILE "File2"                    !displays the contents of File2
185                                     !on the terminal display
190 SEND OUTPUT TO "File5"
200 COPYFILE "File2"                    !writes the contents of File2
999 END                                 !to File5
```

```
10 COPYFILE "File1/Lock1" TO "File2"     !File2 does not have a lockword
20 COPYFILE "File3" TO "File4", "Lock3"  !File4 does not have a lockword
```

```
100 SYSTEM "FILE LINE; DEV=LP"
120 SYSTEM "FILE LASER; DEV=PP,8;ENV=LP602.ENV2680A.SYS;CCTL"
140 COPYFILE TEXT TO "*LASER"
160 COPYFILE WORK TO "*LINE"
```

CREATE

The CREATE statement creates a BASIC DATA, binary, or ASCII data file.

Syntax

CREATE [*file_type*] *fname* [,RECSIZE [=] *num_expr1*]

[,FILESIZE [=] *num_expr2*] [,STATUS [=] *num_var*]

Parameters

file_type

The value of *file_type* can be either the keyword ASCII or binary. Specifying either keyword results in the creation of a file of the corresponding type. If no *file_type* is specified, then a BASIC DATA file is created.

fname

String literal or string expression containing the name of the file.

RECSIZE
FILESIZE STATUS — These clauses can be in any order.

num_expr1 — Record length. If positive, each record has *r* words. If *r* is negative, each record has *r* bytes. If not specified, and file is type BASIC DATA, each record has 256 bytes (128 words).

num_expr2 — File size; maximum number of records in file. Cannot change after the file is created. The default is established by the operating system.

num_var — If the CREATE statement successfully creates the file, it sets this variable to zero; otherwise, it sets it to a nonzero value.

Examples

The following examples show the use of the CREATE statement.

```
     10 CREATE ASCII "File1",RECSIZE=-100,FILESIZE=1000,STATUS=File1stat
     20 CREATE "File2.mktg",FILESIZE=2500
     30 CREATE "File3.lab.hp",RECSIZE 300,FILESIZE 5000,STATUS=Created
     40 CREATE ASCII "File4",STATUS=Success
     50 CREATE BINARY Binfile
     60 CREATE BINARY File5,RECSIZE=-80,FILESIZE=5000,STATUS=Created
```

CURSOR

The CURSOR statement is used to position the cursor and to set display enhancements. The actions specified in the cursor-item list are carried out left to right. Any error in a list of actions causes execution to terminate. There are three pointers that JOINFORM maintains. These are the input, output, and cursor pointers. Setting the input pointer also sets the cursor pointer. Setting the cursor pointer does not change the input or output pointers. Reading a variable from a JOINFORM with the INPUT or ENTER statement advances the input pointer. The order of the input and output fields is defined when the form is created with the JOINEDIT program. The IFLD, OFLD, CFLD, SETIFLD, SETOFLD, and SETCFLD functions are allowed only while a JOINFORM is active.

Syntax

CURSOR *cursor_item_list*

Parameters

cursor_item_list — A list containing one or more unique selections from the following options, separated by commas or semicolons.

```
   [{,}    ] {,}                     {,}
Row [{;} Col ] {;} Col  (Enhance_string  {;} num_chars )

{IFLD   }
{OFLD   }
{CFLD   }
{SETIFLD} (field )
{SETOFLD}
{SETCFLD}
```

Row — Specifies the row display memory coordinate. This coordinate must be a numeric expression, variable, or constant.

Col — Specifies the column display memory coordinate. This coordinate must be a numeric expression, variable, or constant.

enhance_string — A quoted string of characters, or a string variable specifying the display enhancement:

- h or H: Half-Bright
- i or I: Inverse
- b or B: Blinking

* u or U: Underline

num_chars A numeric expression, variable, or constant that specifies the length of the display enhancement.

IFLD This function moves the cursor to the *field* output field in a JOINFORM. The current input field number is set to *field*.

OFLD This function moves the cursor to the *field* output field in a JOINFORM. The current output field number is set to *field*.

CFLD This function moves the cursor to the *field* input field in a JOINFORM. The current input and output field numbers are not modified when this function is executed. A subsequent INPUT statement will position the cursor to this field, but will read data beginning at the field specified input pointer.

SETIFLD This function sets the current input field number to *field*. The cursor is not moved.

SETOFLD This function sets the current output field number to *field*. The cursor is not moved.

SETCFLD This function sets the current cursor field number to *field*. The cursor is not moved. The current input and output field numbers are not modified. A subsequent INPUT statement will position the cursor to *field*, but will read data beginning at the input pointer.

field A numeric expression, variable or constant that evaluates to a numeric value that specifies the number of a field on the JOINFORM.

Cursor Position on the Terminal Screen

For the purpose of CURSOR positioning, the first row in display memory is row 1. The leftmost column in display memory is column 1. When specifying the cursor position with the CURSOR statement in conjunction with the row and column, at least one of row and column must be specified. An error occurs when the value for a row or column is greater than 999. If a row number greater than the number of lines in the display memory is specified the cursor is positioned to the last row. If a column number greater than 80 is specified one line is skipped. Regardless of where the cursor is in display memory, it always remains visible on the display screen. Therefore, the CURSOR statement can be used to scroll or page through display memory.

The functions SETIFLD, SETOFLD, and SETCFLD set the internal field pointer, but do not move the cursor. A subsequent INPUT or OUTPUT statement positions the cursor to the current cursor field or output field. This is more efficient than using IFLD, OFLD, or CFLD, because those functions set the internal field pointer and position the cursor. The cursor would then have been moved twice.

Screen Enhancements

A screen enhancement is set beginning at the current position of the cursor for a length that is determined by *num_chars*. Legal enhancement strings contain the characters h or H for "Half-bright", i or I for "Inverse", b or B for "Blinking" or u or U for "Underline". The empty string (" ") indicates that enhancements are to be turned off. The enhancement string may contain any of the characters above and use a blank (" "), comma (","), or semicolon (";") as a separator between characters for visual clarity.

An enhancement string that contains only blanks, commas, or semicolons is treated as an empty string that turns off enhancements. The legal values for *num_chars* are -999..999. Depending on the value of *num_char*, one of the following will occur:

* *num_chars* evaluates to a positive value; *num_chars* characters are set to the specified enhancement one character at a time. Each of the individual characters is prefixed with the appropriate escape sequence required for the enhancement. The escape sequence prefixing the character following the last character to be enhanced contains the enhancement terminator.

* *num_chars* evaluates to zero; the escape characters that turn on the specified enhancement prefix the characters at the current cursor position. The enhancement is terminated as follows. If there is an enhancement on the line at a point following the current cursor position then that enhancement terminates the specified enhancement. Otherwise, the specified enhancement extends only to (and including) the last non-blank character on the line.

* *num_chars* evaluates to a negative value; the character at the current cursor position is prefixed with the escape sequence for the specified enhancement. The character at the position in screen memory that is *num_chars* to the right of the current cursor position is followed by the enhancement terminator.

  The enhancement terminator causes the next character to not have enhancements. This may cause strange results when putting an enhancement on a line with existing enhancements. Enhancements that go past the end of the line may also cause some strange results. Thus, characters between the current cursor position and the enhancement terminator are not individually enhanced.

  For example, 'CURSOR (,1),("HI",50)' performs exactly the same function as 'CURSOR (,1),("HI",-50)' if, in the latter case, the current line contains no enhancement terminators. Execution of the second statement is faster.

The use of positive and non-positive enhancement lengths intermixed on a single screen produces unpredictable results. Therefore, we do not recommend this.

Examples

```
     10  DISP '27"H"'27"J";
     20  OPEN FORM "main:jfmain"  !Simple form with five 18 character fields
     30  FOR I = 1 TO 5           !Start output to each field
     40    PRINT RPT$(VAL$(I),3)  !Set fld1 to 111, fld2 to 222, etc
     50  NEXT I
     60  CURSOR IFLD(4)           !Positions the cursor to the fourth field
     70  INPUT A$                 !a carriage return here will read 444
     71                           !into A$ and increment IFLD(x) to the fifth field
     80  CURSOR CFLD(1)           !Positions the cursor to the first field
     81                           !on the form
     90  INPUT B$                 !A carriage return here will read 555
     91                           !into B$, not the 111 from the first field
     92                           !The first field is where the cursor is
     93                           !located when the carriage return was pressed
     95  CURSOR SETIFD(2),SETCFLD(2)  !Sets the cursor and input pointers
     96                           !to the second field, but does not move
     97                           !the cursor
     98  INPUT C$                 !The cursor is now moved to the cursor
                                  !field specified in line 95 and reads
     99                           !the contents of the current input
     100                          !field into C$
     101  CURSOR SETIFLD(5)       !Sets the input field to field 5
     102  ENTER D$                !Reads the contents of input field
     103                          ! 5 into D$
     105  CLOSE FORM;REMAIN
     110  PRINT A$,LIN(1),B$      !A$=444, B$=555
     999  END
```

DATA

The DATA statement lists data for the READ statement. The data that the DATA statement provides is assigned to variables by the READ statement.

Syntax

DATA *datum* [, *datum*]...

Parameters

datum Numeric or string literal. A string literal can be enclosed in quotes, but doesn't have to be. If it is enclosed in quotes it is called a quoted string literal; if not, it is called an unquoted string literal. Leading and trailing spaces are not part of an unquoted string literal, but embedded spaces are.

A DATA statement is not executable. When the program reaches a DATA statement, it proceeds to the next line following it.

A data pointer points to the datum that is assigned to the next variable. Before a program unit is executed, the data pointer is set to point to the first datum in the program unit's first or lowest-numbered DATA statement. The data in a DATA statement are read from left to right. When all the data in one DATA statement are read, the data pointer is positioned at the first datum in the next DATA statement. Within a program unit, DATA statements are used in line number order.

When one program unit calls another, the data pointer points to the first data item in the called program unit. When the called program unit returns control to the calling program unit, the data pointer returns to its position in the calling program unit at the time of the call.

Examples

```
10 DATA "2", truffles, "four", A B C, 56
```

Datum	**Description**
"2"	Quoted string literal
truffles	Unquoted string literal
"four"	string literal
A B C	Unquoted string literal
56	Numeric literal

The following program shows the use of the data statement. It reads, and then prints three variables.

```
>LIST
 !  DATAEX
     10 READ A,B,C$
     20 DATA 1,2,"THREE"
     30 PRINT A
     40 PRINT B
     50 PRINT C$
     60 END
>RUN
 1
 2
THREE
>
```

DBASE IS

The DBASE IS statement identifies the database to be searched or sorted. The statement is global to the entire program. Once specified, it remains in effect until another DBASE IS statement is executed. The database identified in this statement must be open. If it is not, an error occurs.

Syntax

DBASE IS *dbname* $

Parameters

dbname $ A string variable, whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

Examples

```
     100    DBASE IS Db_name$
```

When *dbname* $ is a null string, the DBASE IS specification is reset to nothing. It is not an error to specify a null string. An error occurs if a string with all blanks is specified.

DBCLOSE

The DBCLOSE statement terminates database access, makes a data set temporarily or permanently inaccessible, or rewinds a data set.

Syntax

```
     DBCLOSE dbname $[,MODE[=]dbclose_mode ]
                    [,DATASET[=]dataset ]
                    [,STATUS[=]status_array(*) ]
```

Parameters

dbname $ — A string variable, whose value is a TurboIMAGE database name. *dbname* $ must be the variable that was passed to a successful DBOPEN.

dbclose_mode — A numeric expression that evaluates to one of the following TurboImage database modes:

Mode	Effect
1	Terminate access to entire database and ignore the *dataset* parameter. (Default)
2	Terminate access to *dataset*, but leave database open.
3	Rewind the data set.

dataset — A string expression with a maximum length of 16 characters. Its value is the name of a data set. The name must be left-justified and if shorter than 16 characters must be terminated by a semicolon or blank. This parameter can also be an integer or short integer corresponding to the desired dataset number.

status_array — A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

```
     100 DBCLOSE Data_base$,STATUS status(*)
     110 DBCLOSE Data_base$,MODE=1,STATUS=Status(*)
     120 DBCLOSE Data_base$,MODE=2,DATASET Dataset$,STATUS status(*)
```

DBDELETE

The DBDELETE statement deletes a record from a manual master or detail data set.

The database must be open in mode one, three, or four. See the DBOPEN statement for the meaning of these modes. If mode one is selected, a covering lock is required.

Syntax

DBDELETE *dbname* $, DATASET[=]*dataset* [, Status[=]*status_array(*)*]

Parameters

dbname $ A string variable, whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

dataset A string expression with a maximum length of 16 characters. Its value is the name of a data set. The name must be left-justified and if shorter than 16 characters must be terminated by a semicolon or blank. This parameter can also be an integer or short integer corresponding to the desired dataset number.

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

```
     110 DBDELETE Data_base$,DATASET=Data_set$,STATUS=Status(*)
     120 DBDELETE Data_base$,DATASET Data_set$,STATUS Status(*)
```

DBERROR

The DBERROR statement moves a database error message as an ASCII string to a string variable specified using the RETURN parameter. The conversion of the error number in the *status_array* is as listed in the DBERROR message table in the section describing the DBERROR library procedure in the *TurboImage/XL Database Management System*.

Syntax

DBERROR STATUS[=] *status_array(*)*, RETURN[=]*str_var*

Parameters

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

str_var A string variable at least 72 characters in length that serves as the buffer to which the multi-line error message is returned.

Examples

```
     110 DBERROR STATUS=Status(*),RETURN=Message$
```

DBEXPLAIN

DBEXPLAIN prints a multi-line message on MPE's standard list device, usually a terminal, which describes the most recent TurboIMAGE library procedure call. Information about the results of the call are explained on the basis of the information contained in the *status_array* parameter. In the event of an error, the message printed is more detailed than the message returned by the DBERROR statement. This statement must be placed immediately after the library procedure call.

Syntax

DBEXPLAIN STATUS[=]*status_array* (*)

Parameters

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

```
     100 DBEXPLAIN STATUS=Status(*)
```

DBFIND

The DBFIND statement locates the master set entry that matches a specified search item value. It sets up pointers to the first and last entries of the detail data set chain in preparation for chained access to data entries which are numbers of the chain. The path is determined and chain pointers located on the basis of a specified search item and its value.

Syntax

```
                                            {str_expr1 }
DBFIND dbname $, DATASET[=]dataset  , ITEMS[=]{num_expr1 } ,

      {str_expr2 }
KEY[=]{num_expr2 } , [STATUS[=]status_array (*)]
```

Parameters

dbname $ A string variable, whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

dataset A string expression with a maximum length of 16 characters. Its value is the name of a detail data set. The name must be left-justified and, if shorter than 16 characters, must be terminated by a semicolon or blank. This parameter can also be an integer or short integer corresponding to the desired dataset number.

str_expr1 A string expression that evaluates to the left-justified name of a detail data set search item that has a maximum length of 16 characters. If shorter than 16 characters, the value must be terminated by a semicolon or blank.

num_expr1 A numeric expression that evaluates to a short integer referencing the search item number that defines the path containing the desired chain.

str_expr2 If the dataset is to be read in CALCULATED mode (mode seven), *str_expr2* evaluates to the name of the search item to be used in calculated access to locate the desired chain head in the master data set. The maximum string length is 16 characters.

num_expr2 If the dataset is to be read in CALCULATED mode (mode seven), *num_expr2* evaluates to a short integer referencing the search item number to be used in calculated access to locate the desired chain head in the master data set. This can also be a short integer numeric array.

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

```
100 DBFIND Db$,DATASET Ds$,ITEMS K$,KEY A$
110 DBFIND Db$,DATASET Ds$,ITEMS=N,KEY=A$
120 DBFIND Db$,DATASET Ds$,ITEMS N1,KEY N2
130 DBFIND Db$,DATASET Ds$,ITEMS=K$,KEY=N
140 DBFIND Db$,DATASET Ds$,ITEMS K$,KEY A$,STATUS S(*)
150 DBFIND Db$,DATASET Ds$,ITEMS=N,KEY=A$,STATUS=S(*)
160 DBFIND Db$,DATASET Ds$,ITEMS N1,KEY N2,STATUS S(*)
170 DBFIND Db$,DATASET Ds$,ITEMS=K$,KEY=N,STATUS=S(*)
```

DBGET

DBGET reads an entire record or specified data items from a data set. The DBGET statement can be used in the following ways:

DBGET...USING — reads data into an internal buffer that is used as a source for unpacking into a list of local variables.

DBGET...INTO — reads data into the buffer specified.

DBGET...USING...INTO — reads data into the buffer specified by the INTO clause that is used as a source for unpacking into a list of local variables.

Syntax

```
              {INTO str_var                               }
              {USING line_id                              }
DBGET dbname $ {                           {,}            }
              {USING line_id  INTO str_var  {;} DATASET[=]dataset }

[, MODE[=]read_mode ]

[, ITEMS=item_list ]

[      {str_expr }]
[, KEY={num_expr }]

[, STATUS[=]status_array (*)]
```

Parameters

dbname $ — A string variable whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

str_var — The string variable buffer that the values of the data items specified in the *item_list* are moved into. The values in *str_var* must be assigned to HP Business BASIC/XL variables using HP Business BASIC/XL's UNPACK statement.

line_id — A line number or label for a PACKFMT or IN DATASET statement. The referenced statement is used to unpack data automatically into program variables.

dataset — A string expression with a maximum length of 16 characters. Its value is the name of a data set. The name must be left-justified and if shorter than 16 characters must be terminated by a semicolon or blank. This parameter can also be an integer or short integer corresponding to the desired dataset number.

read_mode Either a numeric expression that evaluates to one of the following or a string expression that evaluates to one of the equivalent mnemonics:

Value	Mnemonic	TurboIMAGE Mode
1	READ	Reread
2	SERIAL	Serial forward read
3	SERIALBACK	Serial backward read
4	DIRECT	Direct read
5	CHAIN	Chained forward read
6	CHAINBACK	Chained backward read
7	CALCULATED	Calculated read
8	PRIMARY	Primary calculated read

If this parameter is not specified, the default value is two, (serial).

item_list The *list* parameter for the DBGET TurboIMAGE library procedure. The name of a string or an array of one-word integers containing an ordered set of data item identifiers. The value for each element in the ordered list of data item identifiers is packed into *str_var* in the same order that they appear in the list.

If the *item_list* is a string variable, then the names of the data items must be left-justified, separated by commas and terminated with a semicolon or blank. The embedded blanks are allowed and no name can appear more than once.

The data *item_list* can contain special symbols such as @, which specifies all data items in the data set. Consult the Special List Parameter Constructs table in the explanation of the DBPUT library procedure in the *TurboIMAGE/XL Database Management System* for additional special symbols and their usage.

If referencing data items by number, the first word in the short integer array must be the total number of elements in the array. This number is followed by that number of unique data item numbers contained in the first word of the array.

The *item_list* specified is returned internally by TurboIMAGE as the *current list*. Consult the *TurboIMAGE/XL Database Management System* for details about the benefits of using TurboIMAGE's *current list*. If the ITEMS option is not specified, HP Business BASIC/XL sets the *item_list* to "@;".

num_expr Used only with DIRECT read mode, mode 4. Its value is the integer record number of the entry to be read.

str_expr Used only with CALCULATED or PRIMARY read mode, modes 7 and 8, respectively. Its value is a search item value for the master data set referenced in *dataset*.

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

The following examples show the use of the DBGET statement.

```
100 DBGET Db$ INTO S$,DATASET=Ds$,STATUS=S(*)
110 DBGET Db$ INTO S$,DATASET=Ds$,MODE=1,STATUS=S(*)
120 DBGET Db$ INTO S$,DATASET=Ds$,MODE=2,STATUS=S(*)
130 DBGET Db$ INTO S$,DATASET=Ds$,MODE=3,ITEMS=I$,STATUS=S(*)
140 DBGET Db$ INTO S$,DATASET=Ds$,MODE=4,ITEMS=I$,KEY=K$,STATUS=S(*)
150 DBGET Db$ INTO S$,DATASET=Ds$,MODE=7,STATUS=S(*),ITEMS=I$,KEY=N
160 DBGET Db$ INTO S$,DATASET Ds$,MODE 8,KEY N,STATUS S(*)
170 DBGET Db$ INTO S$,DATASET Ds$,STATUS S(*)
180 DBGET Db$ INTO S$,DATASET Ds$,STATUS S(*),ITEMS I$
190 DBGET Db$ INTO S$,DATASET Ds$,ITEMS I$,STATUS S(*)
200 DBGET Db$ INTO S$,DATASET Ds$,ITEMS I$,KEY N,STATUS S(*)
210 DBGET Db$ INTO S$,DATASET Ds$,KEY K$,STATUS S(*)
220 DBGET Db$ USING 400; DATASET Ds$
230 DBGET Db$ USING Pack1; DATASET Ds$,STATUS=S(*)
240 DBGET Db$ USING Pack1 INTO S$,DATASET=Ds$
400 IN DATASET Ds$ USE A,B, SKIP 10,D$
410 Pack1: PACKFMT A,B, SKIP 10,D$
```

The following statements:

```
100 DBUPDATE Dbase$ USING 200 INTO D$; DATASET = "parts"
200 PACKFMT A,Price,Company$
```

are equivalent to:

```
100 DBGET Dbase$ INTO D$;DATASET="parts"
110 UNPACK USING 200;D$
200 PACKFMT A,Price,Company$
```

DBINFO

The DBINFO statement provides information about the database specified. The information returned is restricted by the user class number when the database is opened. Any data items, data sets, or paths of the database that are inaccessible to that user are considered to be nonexistent.

Syntax

```
                {DATASET [=] dataset }
DBINFO dbname $, {ITEMS [=] item      }

, MODE [=] mode  , RETURN [=] str_var  [, STATUS [=] status_array (*)]
```

Parameters

dbname $ A string variable whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

dataset A string expression with a maximum length of 16 characters. Its value is the name of a data set. The name must be left-justified and if shorter than 16 characters must be terminated by a semicolon or blank. This parameter can also be an integer or short integer corresponding to the desired dataset number.

item A string or numeric expression that evaluates to the name of a data item or evaluates to a numeric value referencing a data item, respectively. Whether the DATASET or ITEMS option is selected is dependent on the mode selected as described in the DBINFO library procedure in the *TurboIMAGE/XL Database Management System*.

mode A numeric expression that evaluates to a short integer indicating the type of information desired. Available modes are detailed in the explanation of the DBINFO library procedure in the *TurboIMAGE/XL Database Management System*.

str_var The name of the string to which the requested information is returned. The required length is dependent on the type of information to be returned as specified by the MODE parameter.

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

The following examples show the use of the DBINFO statement.

```
     120 DBINFO Db$,DATASET=Ds$,MODE=M,RETURN=Buf$,STATUS=S(*)
     130 DBINFO Db$,DATASET Ds$,MODE M,RETURN Buf$,STATUS S(*)
     140 DBINFO Db$,ITEMS S$,MODE M,RETURN Buf$,STATUS S(*)
     150 DBINFO Db$,ITEMS S$,MODE M,RETURN Buf$,STATUS S(*)
```

DBLOCK

The DBLOCK statement applies a logical lock to a database, a data set, or a data item value to all but one user. Then, the user can write to the locked area. The PREDICATE statement aids in locking database items in DBLOCK modes five and six. Without the PREDICATE statement, the PACK statement must be used to build a predicate string for the DBLOCK statement.

Syntax

DBLOCK *dbname* $ [, MODE [=] *lock_mode*]
[{DATASET [=] *dataset*   }]
[, {DESCRIPTOR [=] *str_expr* }]

[, STATUS [=] *status_array* (*)]

Parameters

dbname $ A string variable whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

lock_mode Evaluates to an integer indicating the type of locking desired:

Code	Effect
1 (default)	Locks database unconditionally
2	Locks database conditionally
3	Locks data set unconditionally
4	Locks data set conditionally
5	Locks data item entry unconditionally
6	Locks data item entry conditionally

If a data item is locked unconditionally in mode 5, the entry for that item does not have to exist for the lock to succeed.

dataset A string expression with a maximum length of 16 characters. Its value is the name of a data set. The name must be left-justified and if shorter than 16 characters must be terminated by a semicolon or blank. This parameter can also be an integer or short integer corresponding to the desired dataset number. Required only if *lock_mode* is three or four.

str_expr A string expression that is required only if *lock_mode* is five or six. Its value is a predicate lock string that describes the locking condition. The PREDICATE statement is used to set up the predicate lock string. The format of the PREDICATE lock descriptors is presented in the description of the DBLOCK library procedure in the *TurboIMAGE/XL Database Management System*.

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

The following examples show the use of the DBLOCK statement. In line 30, a PREDICATE statement has been issued for use with lines 150 and 160.

```
 30 PREDICATE Pred$ FROM Ds$ WITH Item$="skates"
100 DBLOCK Db$,STATUS=S(*)
110 DBLOCK Db$,MODE=1,STATUS=S(*)
120 DBLOCK Db$,MODE=2,STATUS=S(*)
130 DBLOCK Db$,MODE 3,DATASET Ds$,STATUS S(*)
140 DBLOCK Db$,MODE 4,DATASET Ds$,STATUS S(*)
150 DBLOCK Db$,MODE 5,DESCRIPTOR Pred$,STATUS S(*)
160 DBLOCK Db$,MODE 6,DESCRIPTOR Pred$,STATUS S(*)
```

DBMEMO

The DBMEMO statement sends a message to the transaction log file.

Syntax

DBMEMO *dbname* $, MSG[=]*str_expr* [, STATUS[=]*status_array* (*)]

Parameters

dbname $ A string variable whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

str_expr A string of ASCII characters of up to 512 characters in length to be written to the log file.

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

The following examples show the use of the DBMEMO statement.

```
110 DBMEMO Db$,MSG=Message$,STATUS=Stat(*)
120 DBMEMO Db$,MSG Message$,STATUS Stat(*)
```

DBOPEN

The DBOPEN statement initiates database access and sets TurboIMAGE's user class number and access mode for subsequent database operations. The first two characters in the *dbname* variable must be blanks.

Syntax

DBOPEN *dbname* $[, PASSWORD[=]*str_expr*] [, MODE[=]*open_mode*]

[, STATUS[=]*status_array* (*)]

Parameters

dbname $ A string variable whose value is a TurboIMAGE database name. The first two characters in the string must be blanks followed immediately by the actual database name. This variable must be used in all other statements that call this database.

str_expr Evaluates to the database's password. Required the if database is protected with a password.

open_mode A numeric expression that evaluates to one of the valid TurboIMAGE access modes in Table 4-2. See the description of the DBOPEN library procedure in the *TurboIMAGE/XL Database Management System* for more information. If not specified, the default is seven, exclusive read.

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Table 4-2. Database Access Modes

Open Mode	**Allows**	**And concurrent**	**Concurrent Modes Allowed**
1	Modify with enforced locking	Modify	1, 5
2	Update	Update	2, 6
3	Exclusive modify	None	None
4	Modify	Read	6
5	Read	Modify	1, 5
6	Read	Modify	6 and either 2, one 4, or 8
7	Exclusive read	None	None

8	Read	Read	6, 8

Examples

The following statements show the use of the DBOPEN statement.

```
  90 Database$ = "  Clients"  !Database name is preceded by two spaces
 100 DBOPEN Data_base$,STATUS=S(*)
 110 DBOPEN Data_base$,PASSWORD="synergy",STATUS=S(*)
 120 DBOPEN Data_base$,PASSWORD=Pw$,MODE=4,STATUS=S(*)
 130 DBOPEN Data_base$,PASSWORD=Pw$,MODE=2,STATUS=Status(*)
 140 DBOPEN Data_base$,MODE 1,STATUS Status(*)
 150 DBOPEN Data_base$,MODE 7,STATUS S(*)
 160 DBOPEN Data_base$,STATUS Status(*)
 170 DBOPEN Data_base$,PASSWORD "Quanta",STATUS Status(*)
```

DBPUT

The DBPUT statement adds new entries to a manual master or detail data set.

The database must be open in access mode one, three, or four (see Table 4-2 in "DBOPEN Statement" for the meanings of these modes). A covering lock must be in place if mode one is used.

DBPUT...USING... Data will be packed into an internal buffer from the list of local variables specified in the PACKFMT statement before writing into the data set.

DBPUT...FROM... Data will be transferred from the buffer into the data set.

DBPUT...USING...FROM... Data will be packed into the buffer specified in the FROM clause using the PACKFMT list that will then be transferred into the data set.

Syntax

```
              {USING line_id              } {,}
DBPUT dbname $ {FROM[=]str_var             } {;} DATASET[=]dataset
              {USING line_id  FROM[=]str_var }

[, ITEMS[=]item_list ] [, STATUS[=]status_array (*)]
```

Parameters

dbname $ A string variable whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

dataset A string expression with a maximum length of 16 characters. Its value is the name of a data set. The name must be left-justified and if shorter than 16 characters must be terminated by a semicolon or blank. This parameter can also be an integer or short integer corresponding to the desired dataset number.

str_var The string variable containing the data item values to be added to the database. The values must be in the same order as their data item identifiers in the *items_list* parameter. The values must be packed into *str_var* from their corresponding HP Business BASIC/XL variables using HP Business BASIC/XL's PACK statement.

line_id A line number or label for a PACKFMT or IN DATASET statement. The referenced statement is used to automatically pack data from program variables.

item_list The name of an ordered set of data item identifiers, either names or numbers. The value of each data item is in the corresponding position in the ordered set of values contained in *str_var*. Any search or sort items defined for the entry must be included in *item_list*. Fields of unreferenced items are filled with binary zeros.

If the *item_list* is a string variable, the list of data item names must be left justified in the string. Individual data item names are separated by commas and the last is followed by a semicolon or blank. Embedded blanks are not allowed and no name can appear more than once.

The data *item_list* can contain special symbols such as @, which specifies all data items in the data set. Consult the Special List Parameter Constructs table in the explanation of the DBPUT library procedure in the *TurboIMAGE/XL Database Management System* for additional special symbols and their usage.

If referencing data items by number, the first word in the short integer array must be the total number of elements in the array. This number is followed by that number of unique data item numbers.

The *item_list* specified is returned internally by TurboIMAGE as the *current list*. Consult the *TurboIMAGE/XL Database Management System* for details about the benefits of using TurboIMAGE's *current list*. If the ITEMS option is not specified, HP Business BASIC/XL sets the *item_list* to "@;".

Examples

The following examples show the use of the DBPUT statement.

```
110 DBPUT Db$ FROM S$,DATASET=Ds$,STATUS=S(*)
130 DBPUT Db$ FROM S$,DATASET=Ds$,STATUS=S(*)
150 DBPUT Db$ FROM S$,DATASET Ds$,STATUS S(*),ITEMS I$
170 DBPUT Db$ FROM S$,DATASET Ds$,ITEMS I$,STATUS S(*)
220 DBPUT Db$ USING 400; DATASET Ds$
230 DBPUT Db$ USING Pack1; DATASET Ds$,STATUS=S(*)
400 IN DATASET Ds$ USE A,B, SKIP 10,D$
410 Pack1: PACKFMT A,B, SKIP 10,D$
420 DBPUT D6$ USING 400, FROM=S$,DATASET=Ds$
```

The following statements:

```
100 DBPUT Dbase$ USING 200 FROM=D$; DATASET = "parts"
200 PACKFMT A,Price,Company$
```

are equivalent to:

```
100 PACK USING 200;D$
100 DBPUT Dbase$ FROM D$;DATASET="parts"
200 PACKFMT A,Price,Company$
```

DBUNLOCK

The DBUNLOCK statement cancels the restriction imposed by the DBLOCK statement with the same *dbname*.

Syntax

DBUNLOCK *dbname* $[, STATUS[=]*status_array* (*)]

Parameters

dbname $ A string variable, whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

status_array A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Redundant DBUNLOCK statements are ignored.

Examples

The following example shows the use of the DBUNLOCK statement.

```
100 DBUNLOCK Db$
110 DBUNLOCK Db$,STATUS=S(*)
120 DBUNLOCK Db$,STATUS S(*)
```

DBUPDATE

The DBUPDATE statement replaces the values of data items in the current address of a specified dataset.

The database must be open in access mode one, two, three, or four (see Table 4-2 in "DBOPEN Statement" for more on the meanings of these modes).

DBUPDATE...USING...	Data will be packed into an internal buffer from the list of local variables specified in the PACKFMT statement that can update the dataset.
DBUPDATE...FROM...	The buffer specified in the FROM clause is used to update the data set.
DBUPDATE...USING...FROM...	Data will be packed into the buffer specified in the FROM clause using the PACKFMT can be used to update the data set.

Syntax

```
                 {USING line_id                }{,}
DBUPDATE dbname $ {FROM[=]str_var               }{;} DATASET[=]dataset
                 {USING line_id  FROM[=]str_var }

[, ITEMS[=]item_list ] [, STATUS[=]status_array (*)]
```

Parameters

dbname $ — A string variable whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

str_var — The string variable containing the data item values to be added to the database. The values must be in the same order as their data item identifiers in the *items_list* parameter. The values must be packed into *str_var* from their corresponding HP Business BASIC/XL variables using HP Business BASIC/XL's PACK statement.

line_id — A line number or label for a PACKFMT or IN DATASET statement. The referenced statement is used to automatically pack data from program variables.

dataset — A string expression with a maximum length of 16 characters. Its value is the name of a data set. The name must be left-justified and, if shorter than 16 characters, must be terminated by a semicolon or blank. This parameter can also be an integer or short integer corresponding to the desired dataset number.

item_list — The name of an ordered set of data item identifiers, either names or numbers. The value of each data item is in the corresponding position in the ordered set of values contained in *str_var*. Any search or sort items defined for the entry must be included in *item_list*. Fields of unreferenced items are filled with binary zeros.

If the *item_list* is a string variable, the list of data item names must be left justified in the string. Individual data item names are separated by commas and the last is followed by a semicolon or blank. Embedded blanks are not allowed and names cannot appear more than once.

The data *item_list* can contain special symbols such as @ that specifies all data items in the data set. Consult the Special List Parameter Constructs table in the explanation of the DBPUT library procedure in the *TurboIMAGE/XL Database Management System* for additional special symbols and their usage.

If referencing data items by number, the first word in the short integer array must be the total number of elements in the array. This number is followed by the unique data item number.

The *item_list* specified is returned internally by TurboIMAGE as the *current list*. Consult the *TurboIMAGE/XL Database Management System* for details about the benefits of using TurboIMAGE's *current list*. If the items option is not specified, HP Business BASIC/XL sets the *item_list* to "@;".

Examples

The following examples show the use of the DBUPDATE statement.

```
     110 DBUPDATE Db$ FROM S$,DATASET=Ds$,STATUS=S(*)
     130 DBUPDATE Db$ FROM S$,DATASET=Ds$,STATUS=S(*)
     150 DBUPDATE Db$ FROM S$,DATASET Ds$,STATUS S(*),ITEMS I$
     170 DBUPDATE Db$ FROM S$,DATASET Ds$,ITEMS I$,STATUS S(*)
     220 DBUPDATE Db$ USING 400; DATASET Ds$
     230 DBUPDATE Db$ USING Pack1; DATASET Ds$,STATUS=S(*)
     400 IN DATASET Ds$ USE A,B, SKIP 10,D$
     410 Pack1: PACKFMT A,B, SKIP 10,D$
```

The following statements:

```
     100 DBGET Dbase$ USING 200 FROM D$; DATASET = "parts"
     200 PACKFMT A,Price,Company$
```

are equivalent to:

```
     100 PACK USING 200;D$
     110 DBUPDATE Dbase$ FROM D$;DATASET="parts"
     200 PACKFMT A,Price,Company$
```

DECIMAL

This statement defines a variable as a type DECIMAL. If the SHORT option is used with it, the variable is type SHORT DECIMAL.

Syntax

```
                     {num_var } [ {num_var }]
[SHORT] DEC[IMAL] {arrayd  } [,{arrayd  }]...
```

Parameters

num_var Name of scalar numeric variable to be declared.

arrayd Numeric array description. The syntax for the array is described under the DIM statement.

Examples

The following are examples of declaring variables of types DECIMAL and SHORT DECIMAL.

```
     100 SHORT DECIMAL Price
     120 SHORT DECIMAL Cost1,Cost2(7),Cost3
     130 DECIMAL Length
     140 DECIMAL D1,D2,D3(6,8),D4(3,5)
```

DEF FN

The DEF FN statement defines the beginning of a multi-line function. It is not executable.

Syntax

Numeric function:

DEF [*type*] FN*identifier* [(*f_param* [,*f_param*]...)]

String function:

DEF FN*identifier* $ [(*f_param* [,*f_param*]...)]

Parameters

type Numeric type (for example, INTEGER, SHORT REAL). If *type* is specified, the numeric function returns a numeric value of that type. If *type* is not specified, the numeric value returned has the default numeric type. A string function returns a string value.

FN*identifier*, FN*identifier* $ Function name. A blank is not allowed between FN and the identifier. For example, if the identifier is Add, the function name is FNAdd or FNAdd$.

f_param A formal parameter. Formal parameters for multi-line functions are specified like as they are for subprograms. The SUBPROGRAM statement explains the specification.

Example

```
     DEF INTEGER FNAdd (INTEGER A,B(*), REAL C,D(*), E$(*,*,*),F$,G)
     DEF FNSearch$(E$(*,*,*),F$, G, #20)
```

Each of the above statements defines the beginning of a multi-line function. FNAdd is a numeric (type INTEGER) function, and FNSearch is a string function.

Each has the following formal parameters:

Parameter	**Type**
A	Scalar integer variable
B	Integer array variable
C	Scalar real variable
D	Real array variable
E$	String array variable
F$	Scalar string variable
G	Scalar variable with the default numeric type
#20	File designator

If a program has more than one multi-line function with the same name, the name refers to the first function with that name; that is, the one that has the lowest-numbered DEF FN statement. The others cannot be called.

If a program unit has a single-line function with the same name as a multi-line function, that program unit can only call the single-line function. Other program units can still call the multi-line function.

DEFAULT OFF

Values that are out of range cause arithmetic errors, explained under the DEFAULT ON statement. The DEFAULT ON statement overrides those error messages. The DEFAULT OFF statement is used following a DEFAULT ON statement to reinstate those error messages. The DEFAULT OFF value is also set when you initially enter to the interpreter.

Syntax

DEFAULT OFF

If the DEFAULT OFF value is set, program execution is suspended when you encounter one of these errors.

DEFAULT ON

Values that are out of range cause the arithmetic errors in the following table. If a DEFAULT ON statement is executed before one of these errors occurs, the error is overridden and a default value is substituted for the value that is out of range.

Syntax

DEFAULT ON

If one of the errors in Table 4-3 occurs before a DEFAULT ON statement is executed or after a DEFAULT OFF statement is executed, program execution is suspended.

The DEFAULT OFF value is set when you initially enter the interpreter.

Table 4-3. DEFAULT ON Values

Error Number	Error Description	Default Values
20	Short integer precision overflow.	32767 or -32768
21	Short decimal precision overflow.	{+ -}9.99999E+63
22	Decimal precision overflow.	{+ -}9.99999999999E+511
24	TAN(N*PI/2) where N is an odd integer.	1.157920892373161E+77
26	Zero to negative power.	1.157920892373161E+77
29	LGT or LOG of zero.	-1.79769313486231E+308
31	Division by zero.	Default type maximum
1139	Integer precision overflow.	-2147483648 or 2147483647
1140	Real precision overflow.	{+ -}1.79769313486231E+308
1141	Short real precision overflow.	{+ -}3.40282E+38

Examples

The following examples show the result of using the DEFAULT ON statement. In the first example, the program does not have DEFAULT ON and a short integer precision overflow results. In the second example, there is a DEFAULT ON statement and the default value of 32767 is substituted for the out of range 2*A.

```
>list
     10 SHORT INTEGER A,B
     20 A=32767
     30 B=2*A
     40 PRINT A
     50 PRINT B
     60 END
>run
Error 20 in line 30
SHORT INTEGER precision overflow.
>15 DEFAULT ON
>list
     10 SHORT INTEGER A,B
     15 DEFAULT ON
     20 A=32767
     30 B=2*A
     40 PRINT A
     50 PRINT B
     60 END
>run
 32767
 32767
>
```

DEG

The DEG statement indicates that angular units will be specified in degrees. The default is Radians. One degree represents 1/360 of a circle. This statement is used with trigonometric functions.

Syntax

DEG

Example

```
10 Radius = 10
20 DEG
30 Area = PI*Radius**2
40 PRINT Area
50 END
```

DETAIL LINE

The DETAIL LINE statement is the foundation for Report Writer execution. When the DETAIL LINE statement is executed, all break conditions are tested and triggered, if appropriate, before the detail line output is produced. In addition, this statement causes all totals to be incremented.

This statement can not occur within a report definition. Also, it can not be executed while any other break condition such as a level break or a page break is being executed.

Syntax

```
                        [               [LINES]]
DETAIL LINE totals_flag  [WITH num_lines  [LINE ]]
```

[USING *image* [; *output_list*]]

Parameters

totals_flag A numeric expression in the SHORT INTEGER range. This value is used as a Boolean to determine what work must be done.

num_lines The maximum number of lines expected to be needed by this statement. This number reflects *ALL* output done before the next DETAIL LINE statement executes.

image An image string or a line reference to an IMAGE line.

output_list A list of output items that is identical to the list for the PRINT USING statement.

Examples

The following examples show the DETAIL LINE statement.

```
     100 DETAIL LINE J WITH 3 LINES
     100 DETAIL LINE True
     100 DETAIL LINE 0 WITH 2 LINES USING Image_line;A, B
```

If the report has not been activated, DETAIL LINE reports an error. If the report output has not started, this statement starts the report output. When starting the report, DETAIL LINE evaluates all BREAK statements in order to update OLDCV/OLDCV$ values, but no break takes place. Once the report output has started, all work depends on the value of the *totals_flag*.

The totals flag is always evaluated by DETAIL LINE. If its value is TRUE (nonzero), break conditions are checked and totals are automatically accumulated. If the totals flag is FALSE (zero), this work is not done.

The output of DETAIL LINE can be controlled by the PRINT DETAIL IF statement. If this statement exists in the report, it is evaluated. If the PRINT DETAIL IF statement is FALSE (zero), the DETAIL LINE statement ends. The WITH and USING clauses are not executed.

If PRINT DETAIL IF returns TRUE, or if the statement does not occur in the report description, the WITH and USING clauses are executed if they exist.

The work done by DETAIL LINE is shown in the following section. The description of executing a break condition occurs after the TRIGGER BREAK statement.

Execution of DETAIL LINE

The following is a sequential description of what happens when the DETAIL LINE statement executes.

- Checks are made to see that DETAIL LINE can be executed. For this to take place, the report must be active and that there are no Report Writer sections currently executing.
- If necessary, the report is started. This will cause the REPORT HEADER, PAGE HEADER, and all HEADER sections to execute.
- The *totals_flag* is evaluated. This value is used as a Boolean value.
- If the *totals_flag* is true (nonzero), do the following:
  - Evaluate ALL break statements, watching for the lowest numbered BREAK statement that is satisfied. The BREAK IF and BREAK WHEN statements are evaluated in summary level order, from one to nine. During these checks, the CURRENT LEVEL from the RWINFO built-in function changes to reflect the BREAK statement executed.
  - If a BREAK condition is satisfied, LASTBREAK is set to the lowest numbered BREAK statement with a satisfied BREAK condition. Take the following steps:
- Set CURRENT LEVEL to LASTBREAK.
- Trigger breaks from level LASTBREAK through nine. This causes the TRAILER and HEADER sections to execute. Some Report Writer counters

are updated, totals are reset and OLDCV values are reset. This process is described under TRIGGER BREAK.

- * Accumulate all TOTALS. GRAND TOTALS are evaluated and added first, then TOTALS are done in ascending level number order.
- * Evaluate the PRINT DETAIL IF statement. If the statement does not occur, or if the expression is true (nonzero), do the following:
  - * Evaluate the WITH clause of DETAIL LINE. If the number of lines left before the page trailer or end of page is smaller than the WITH value, automatically trigger a page break. If a WITH clause is not specified, there must be one line left on the page.
  - * If the USING clause is present on the DETAIL LINE statement, print the detailed line as indicated by the USING clause and the expressions that follow it.

DIM

The DIM statement declares one or more string or array variables.

Syntax

DIM *type_list* [,*type_list*]...

Parameters

type_list

{[*type*]*num_item* [, *num_item*]...}
{*non_num_item* }

type

One of the following:
SHORT INTEGER
INTEGER
SHORT DECIMAL
DECIMAL
SHORT REAL
SHORT (Current default type, either a REAL or a DECIMAL)
REAL
unspecified

If a type is not specified, implicit declaration rules apply. (These rules are explained in "Implicit Declaration" in chapter 3). After *type*, each *num_item* is of that type until another *type* or a *non_num_item* appears.

num_item

A numeric variable name or a numeric array declaration. A numeric array is declared as:

identifier [(*array_subscripts*)]

non_num_items

A scalar string variable name or a string array declaration. A scalar string variable can be declared as either of the following:

{*identifier* $ }
{*identifier* $[*num_expr3*]}

The first declaration results in a scalar string variable that can contain a maximum of 18 characters (default length). The second declaration specifies the maximum length that the string may contain is *num_expr3* characters.

A string array is declared as either of the following:

{*identifier* $(*array_subscripts*) }
{*identifier* $(*array_subscripts*)[*num_expr3*]}

In the first declaration, each element of the array can have a maximum length of 18 characters (default value). In the second declaration, the maximum length of each element is *num_expr3* characters.

array_subscripts A list specifying each *dimension*. Each dimension is separated from the next by commas. An array has between one and six dimensions. The default is one dimension.

dimension A single number or a pair of numbers that has the syntax:

[*num_expr1*:]*num_expr2*

num_expr1 Lower bound for the dimension. If the DIM statement is in the main program unit, the value of *num_expr1* must be constant. *num_expr1* is less than or equal to *num_expr2*. The default is default lower bound specified in the appropriate OPTION BASE statement or the HP Business BASIC/XL configuration file.

num_expr2 Upper bound for the dimension. If the DIM statement is in the main program unit, the value of *num_expr2* must be constant.

num_expr3 Maximum length of each string in the array. The default length is 18.

Examples

```
     10 !Numeric Arrays
     20 DIM Default_type_Arr(20)
     30 DIM INTEGER Int_Arr1(40)
     40 DIM REAL Real_Arr3(10:20,10,9)
     50 DIM DECIMAL Dec_Arr(40), Dec_Arr2(10,4), REAL Real_Arr2(2,2:4)

     10 !String Arrays
     20 OPTION BASE 1
     30 DIM Default_length_string$
     40 DIM Str_len_80$[80]
     50 DIM Str_arr1_len_80$(10)[80]
     60 DIM Str_arr2_len_20$(10:15,-10:0,5,3)[20]
```

DISABLE

The DISABLE statement suppresses the execution of a branch specified by pressing a branch-during-input key and places the branch into the interrupt queue. The interrupt queue contains branches that are to be executed. The key-generated branches in the queue are stored by HP Business BASIC/XL in a format that includes the key number. The branch information for each key is able to be stored only once.

If a key defined as a branch-during-input key is pressed while key generated branch processing is DISABLED, the branch is added to the interrupt queue. If the function key is subsequently redefined by an ON KEY statement and pressed again while the first branch is still in the queue, then the original branch information is overwritten by that present in the second ON KEY statement. There can be at most eight interrupts pending in the queue; one for each of the eight softkeys. The interrupt for a specific key can only be stored once. Pressing a key multiple times while DISABLE is in effect does not result in multiple executions of that key's action when interrupts are enabled.

Syntax

DISABLE

Examples

```
     100 DISABLE
```

DISP

The DISP statement outputs several values. It can use output functions to output control characters. The DISP statement is similar to the PRINT statement. The only difference between the DISP and PRINT statements is that the DISP statement uses the standard list device, and the PRINT statement uses the output device specified by the most recently executed SEND OUTPUT TO statement. If the most recently executed SEND OUTPUT TO statement specifies the standard list device, or if the program does not contain a SEND OUTPUT TO statement, then the PRINT statement is equivalent to the DISP statement.

Syntax

```
                        [,]
DISP [output_item_list ] [;]
```

Parameters

```
                                      [{[,]...}            ]
output_item_ list    [,]...output_item  [{;     } output_item ]...
```

output_item One of the following:

num_expr

str_expr

array_name (*) Array reference. See "Array References in Display List" for more information.

```
                                        {PAGE            }
                                        {{CTL}           }
                     output_function     {{LIN}           }
                                        {{SPA} (num_expr )}
                                        {{TAB}           }
```

See "Output Functions in Display List" for more information.

FOR_clause (FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *d_list*)

See the section that follows, "FOR Clause in Display List", for more information.

, A separator. This prints each new item in a separate output field.

; A separator. This prints each new item right next to the previous item.

Examples

```
     100 DISP
     110 DISP,
     120 DISP;
     130 DISP X,X+Y;A$,LWC$(A$+B$);P(*),Q$(*);PAGE,TAB(10+X),
     140 DISP Z(*), (FOR I=1 TO 10, Z(I); 2*Z(I); I*Z(I)), D$;
     150 DISP A,,B
```

The DISP statement evaluates the expressions in the display list from left to right and displays their values on the standard list device. It displays numeric values in the current numeric output format (see "Numeric Format Statements").

A DISP statement without a display list prints a carriage return and a line feed (a CRLF) on the output file or device.

FOR Clause in Display List

The display list of a DISP statement can contain a FOR clause. The FOR clause is similar to the FOR NEXT construct.

Syntax

(FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *output_item_list*)

Parameters

num_var A numeric variable assigned the sequence of values: *num_expr1*, *num_expr1* +num_expr3, *num_expr1* +(2**num_expr3*), etc. The DISP or PRINT statement prints the values of the elements of *d_list* for each value that is less than *num_expr2* if *num_expr3* is positive or greater than *num_expr2* (if *num_expr3* is negative).

num_expr1 First value assigned to *num_var*.

num_expr2 Value that *num_var* is compared to before the DISP or PRINT statement prints a value. If *num_var* > *num_expr2*, the loop is not executed.

num_expr3 Amount that *num_var* increases by if *num_expr2* is positive or decreases if *num_expr2* is negative at end of the loop. The default value is 1 if the step option is not specified.

output_item_ list Same as *d_list* in DISP or PRINT statement syntax.

Examples

The following example shows the use of a FOR clause in the display list.

```
20 DISP "Values for A are: ",(FOR I=1 TO 4, A(I);),,,"X Value: ",X
```

If each variable is assigned the following values prior to execution of line 20:

```
A(1) = 10
A(2) = 20
A(3) = 30
A(4) = 40
X    = 50
```

The output generated by line 20 is:

```
Values for A are:  10  20  30  40
X Value:           50
```

Display list FOR clauses can be nested.

```
20 DISP (FOR I=1 TO 3, (FOR J=1 TO 2 (FOR K=1 TO 2, B(I,J,K))))
```

For each combination of values of I, J, and K, the following table shows the variable value that the above statement prints.

Value of I	**Value of J**	**Value of K**	**Variable Printed**
1	1	1	B(1,1,1)
1	1	2	B(1,1,2)
1	2	1	B(1,2,1)

1	2	2	B(1,2,2)
2	1	1	B(2,1,1)
2	1	2	B(2,1,2)
2	2	1	B(2,2,1)
2	2	2	B(2,2,2)
3	1	1	B(3,1,1)
3	1	2	B(3,1,2)
3	2	1	B(3,2,1)
3	2	2	B(3,2,2)

DISP USING

The DISP USING statement dictates the format of the values that it prints by specifying either a format string or an IMAGE statement. The PRINT USING statement is similar to the DISP USING statement. Table 4-4 compares them.

Table 4-4. Comparison of DISP USING and PRINT USING

Statement	**Prints output to**
DISP USING	Standard list device.
PRINT USING	ASCII data file, if specified; otherwise, the device specified by the most recently executed SEND OUTPUT TO statement. If that device is the standard list device, PRINT USING is equivalent to DISP USING.

Syntax

DISP USING *image* [; *output_item* [, *output_item*]...]

Parameters

image Either a string expression or the line identifier of an IMAGE statement. See "Format String and IMAGE Statement" for more information.

output_item Numeric or string expression. It can be a scalar variable, an array element or a substring.

Examples

```
     110 DISP USING 100         !Uses the IMAGE statement at line 100
     120 DISP USING Image1      !Uses the IMAGE statement at the line
     125                        !contained in Image1
     130 DISP USING Image$      !Uses the IMAGE statement in Image$
     160 DISP USING "5X"        !Uses the image "5X"
     200 IMAGE1:  2A 4X
```

ELSE

The ELSE statement is used as part of the IF THEN ELSE construct. It is used to indicate what is to be executed if a specified numeric expression is zero or FALSE. Refer to the IF THEN ELSE statement for information.

ENABLE

The ENABLE statement initiates the execution of any key-generated branches in the interrupt queue that have been suppressed by a DISABLE statement. No action is taken when the interrupt queue is empty. If more than one branch is in the queue, then the branch with the highest priority is executed immediately following execution of the ENABLE statement. Subsequent branches are executed as described in the "Priority of Handling the Branch" section in chapter 8.

Syntax

ENABLE

Examples

```
     100 ENABLE
```

ENDIF

The ENDIF statement is part of the IF THEN ELSE construct. It is used indicate the end of that construct. Refer to the IF THEN ELSE statement for more information.

ENDLOOP

The ENDLOOP statement is part of the LOOP construct. It is used to indicate the end of that construct. Refer to the LOOP statement for more information.

END REPORT

The END REPORT statement closes a report normally. It causes all trailer sections to be printed, including the report trailer.

This statement can occur anywhere in the report subunit. It can be used as a command.

Syntax

END REPORT

Examples

The following example shows a line containing the END REPORT statement.

```
     100 END REPORT
```

The END REPORT statement gives an error if there is not an active report. If report output has not started, this statement starts the output.

The END REPORT statement prints all TRAILER sections in descending order from level nine to level one. After these trailers, the REPORT TRAILER section executes. Finally a PAGE TRAILER is printed and a page eject occurs to clear the last page of the report.

The END REPORT statement is guaranteed to end the report, even if an error occurs. Most errors are caught in the report sections and halt the program, but a few errors can occur during END REPORT itself. Whether there are errors or not, there are not active reports at the end of this

statement.

END REPORT DESCRIPTION

This stand-alone statement marks the end of a report description. There is no output associated with this statement.

Syntax

END REPORT DESCRIPTION

Examples

The following example shows a line containing the END REPORT DESCRIPTION statement.

```
        100 END REPORT DESCRIPTION
```

The END REPORT DESCRIPTION statement acts as a comment if there is no active report; if BEGIN REPORT has not executed.

If a report is active and this statement is executed, two possible actions may occur. If another report section is active, that section is ended. Otherwise, the statement is unexpected and an error occurs.

END SELECT

The END SELECT statement is part of the SELECT construct. It is used to indicate the end of that construct. Refer to the SELECT statement for more information.

END TRANSACTION

The END TRANSACTION statement defines the end of the sequence of TurboIMAGE procedure calls begun by the BEGIN TRANSACTION statement. The MSG parameter allows you to log additional information in the log file.

Syntax

END TRANSACTION *dbname* $, MSG[=]*str_expr*, [, MODE[=]*mode*]

[, STATUS[=]*status_array* (*)]

Parameters

dbname $ — A string variable whose value is a TurboIMAGE database name. *dbname* must be the variable that was passed to a successful DBOPEN.

str_expr — A string of ASCII characters of up to 512 characters in length to be written as part of the END TRANSACTION log record.

mode — If not specified, the value is set to one. The modes are the following:

- mode1: end logical transaction.
- mode2: end logical transaction and write contents of the logging buffer in memory to disk.

status_array — A 10-element short integer array to which TurboIMAGE returns an error code. If an HP Business BASIC/XL database statement specifies the STATUS option, an error does not abort the program. Following execution of the database statement the program can check *status_array* and handle the error. The values returned by TurboIMAGE to this array are detailed in the description of the *status* parameter of the equivalent TurboIMAGE library procedure.

Examples

The following examples show the use of the END TRANSACTION statement.

```
     110 END TRANSACTION Db$,MSG=M$,MODE=1
     120 END TRANSACTION Db$,MSG M$,MODE 2,STATUS S(*)
     130 END TRANSACTION Db$,MSG=M$,STATUS=S(*)
```

END WHILE

The END WHILE statement is part of the WHILE construct. It is used to indicate the end of that construct. Refer to the WHILE statement for more information.

ENTER

The ENTER statement assigns characters that are already present in display memory to HP Business BASIC/XL variables. User input from the keyboard is not accepted.

A value is read from the display memory starting from the cursor position until each *enter_item* has been assigned or until the end of data on the line. When there are no more data on a display memory line, the remaining variables in the ENTER statement are not assigned a new value. Commas act to separate values on the line like they do in the INPUT statement. Since any necessary conversion is performed on the data read from the display memory prior to assigning it to HP Business BASIC/XL variables, it is possible to get an error ENTERing numeric variables. For example, attempting to assign the value '99*8' to a numeric variable causes an error.

The ENTER statement can be used to read data from fields of an active JOINFORM into HP Business BASIC/XL variables. Refer to Appendix F of this manual for more information.

Syntax

```
      {enter_element }[{,}{enter_element }]
ENTER {for_clause    }[{;}{for_clause    }]...
```

Parameters

enter_item — *enter_element* or *for_clause*

enter_element — One of the following:

num_var
str_var $
array_name ([*[,*]...])
str_array_name $([*[,*]...])

The last format above has one asterisk per dimension or does not have asterisks. Not using asterisks specifies any number of dimensions. Either format is legal, but the format without asterisks is not compilable. Substrings are also allowed.

for_clause — (FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *enter_item* [, *enter_item*]...)

A *for_clause* is useful for reading array elements. Refer to "FOR Clause in Input List" in chapter 6 for more information.

Examples

```
     300 ENTER Num_var                !Enters a value for Num_var
     310 ENTER Num_var,Str_var$       !Enters a numeric and a string value
     330 ENTER (FOR I=1 to 2,A$(I))   !Enters two elements of a string array
```

EXIT IF

The EXIT IF statement is part of the LOOP construct. It is used to indicate when to exit the construct. Refer to the LOOP statement for more information.

EXTERNAL and GLOBAL EXTERNAL

The EXTERNAL or GLOBAL EXTERNAL statement defines a non-intrinsic procedure or function in an executable library so that the procedure can be called from within the HP Business BASIC/XL program. The purpose of the statement is to specify the name of the procedure or function that is called from within the HP Business BASIC/XL program. If the name in the executable library is different from that to be used within HP Business BASIC/XL, the name of the entry point in the executable library can be specified in the alias clause. The formal parameter list for the EXTERNAL statement must correspond to the formal parameter list in the procedure header of the external routine. Parameters are passed by reference unless the formal parameter is preceded by the keyword VALUE.

Since the language used to write the external procedure or function determines the size and format anticipated for the actual parameters, the language that the external procedure or function is written in must be included in the external's definition. If the external returns a value, (it is a function) then the type of the value returned must be specified if it is not the default numeric type for the main program, subprogram, or function that the definition occurs in.

Syntax

```
       {EXTERNAL}
GLOBAL {EXT     } [lang ] return_type identifier  [ALIAS quoted_& str_lit ]

[[                 [{,}                   ]    ]]
[[[ptype ] parameter  [{;} [ptype ] parameter ] ...]]
```

Parameters

GLOBAL

Allowed only if the statement is in the main block of the program. If GLOBAL appears, the statement is a GLOBAL EXTERNAL statement; if GLOBAL is omitted, the statement is an EXTERNAL statement.

A GLOBAL EXTERNAL definition can appear only in the main block of the program and allows the external to be called from either the main block or any procedure or function within that program. An EXTERNAL statement can appear in the main block or any procedure or function and allows the declared external to be called locally.

Any local EXTERNAL declaration statement takes precedence over that of a GLOBAL EXTERNAL declaration statement, but only while the main block or procedure that contains the local EXTERNAL definition is executing.

lang

One of the following terms for the language that the external procedure or function is written in:

BASIC

HP Business BASIC/XL (default if not specified)

PASCAL

Pascal/XL

PASCAL EXTENSIBLE

A PASCAL/XL routine declared using the EXTENSIBLE option. It is followed here by the numeric literal, *extensible_count*. *extensible_count* is the number of required parameters for the call to the external routine. The required parameters must be supplied for each call from the HP Business BASIC/XL calling routine. Additional non-required formal parameters can be supplied in the actual parameter list following the required parameters. Note that a call to an EXTENSIBLE routine will pass an additional "hidden" parameter to specify the number of parameters actually passed. Refer to the *HP PASCAL Reference Manual* for additional information.

HPC	HP C/XL

return_type Type of the value returned by the function. Can be any HP Business BASIC/XL type or the type BYTE (see "Calling External Subunits" in chapter 3).

identifier The name used within the HP Business BASIC/XL program to call the function or procedure. If calling a function directly without using the FNCALL keyword, this name must follow the syntax of an HP Business BASIC/XL function name; that is, the prefix 'FN' must precede the name. This name is downshifted before searching the executable library for the entry point.

quoted_str_lit The name of the procedure or function in the executable library. This name is referred to as the alias name. The string provided is the case-sensitive name of the external routine in the executable library.

ptype Parameter type. Applies to all parameters between this *ptype* specification and the next *ptype* or string parameter (as in the DIM statement). The *ptype* can be any HP Business BASIC/XL type or the type BYTE (see "Calling External Subunits" in chapter 3). Each formal parameter specified to be a BYTE String$ must be preceded by the keyword BYTE.

parameter One of the following:

[VALUE] *identifier* If *lang* is BASIC, VALUE is ignored. If *lang* is PASCAL or HPC, VALUE indicates that the parameter immediately following it is to be passed by value (rather than by reference).

#*fnum* where *fnum* is a file number as described in chapter 6.

array_name ([*[,*]...]) Gives one asterisk per dimension or does not have asterisks. No asterisks indicates an undefined number of dimensions. Either format is legal, but the format without asterisks is not compilable. The maximum length of each element is the same as declared for the actual parameter by the calling program unit.

Examples

```
100 GLOBAL EXTERNAL Calculate
110 GLOBAL EXTERNAL Add(INTEGER X,Y)
115 GLOBAL EXTERNAL PASCAL String_op(BYTE Str1$,Str2$)
120 EXTERNAL BASIC Subtract ALIAS "sub"(INTEGER X,Y;REAL Z)
130 EXTERNAL PASCAL REAL FNDiv ALIAS "DIV"(INTEGER A,B)
140 EXTERNAL PASCAL INTEGER FNDiv2 ALIAS "divide"(INTEGER A,B)
150 EXTERNAL PASCAL Blob(INTEGER VALUE A, B)
```

For a call to String_op, both actual parameters are passed by reference. The first actual parameter is passed as a packed array of character. The second actual parameter is passed as a Pascal string.

For a call to Blob, the first actual parameter is an integer passed by value, and the second actual parameter is an integer passed by reference.

The following example shows is a HP Business BASIC/XL program that calls an external Pascal program. The Pascal program is called using the PASCAL EXTENSIBLE keywords.

```
extrext2
 10 EXTERNAL PASCAL EXTENSIBLE=2 Pascal_extensible_2(SHORT INTEGER P1,&
    REAL VALUE P2, INTEGER P3, SHORT REAL VALUE P4)
 15 ! Declare and initialize the variables to be used as actual parameters
 20 SHORT INTEGER Sint1
 30 REAL Real1
 40 INTEGER Int1
 50 SHORT REAL Sreal1
 60 Sint1=1;Real1=2;Int1=3;Sreal1=4
 70 CALL Pascal_extensible_2(Sint1,Real1)             ! pass 2 parameters
 80 CALL Pascal_extensible_2(Sint1,Real1,Int1)        ! pass 3 parameters
 90 CALL Pascal_extensible_2(Sint1,Real1,Int1,Sreal1) ! pass 4 parameters
```

FILES ARE IN

The FILES ARE IN statement is used to specify a different default location for data files, for example; a group, group.account, or account other than that assigned by the operating system. Each data file resides in the newly specified default location. However, explicitly stating the group or the group.account in a data file name in a subsequent statement overrides the location specified in the FILES ARE IN statement.

Syntax

FILES [ARE] [IN] *str_expr*

Parameters

str_expr   A string expression that evaluates to a group.account.

Examples

```
100 FILES ARE IN "sfm.mktg"
110 CREATE "File1",FILESIZE=1200         !File1=FILE1.SFM.MKTG
120 CREATE "File2",FILESIZE=1500         !File2=FILE2.SFM.MKTG
130 CREATE "File3.lab.HP",FILESIZE=5000  !File3=FILE3.LAB.HP
999 END
```

FILTER

The FILTER statement starts the database retrieval process for HP Business BASIC/XL's Data Base Sort Feature. A boolean expression that can contain both built-in or user-defined functions, is used as the search condition. When the FILTER statement is executed, the data sets contained in the thread list are accessed in the order and hierarchy specified by the THREAD IS statement. The data retrieved from each data set are unpacked into the local variables as defined in the respective IN DATASET statement. For each data set from the thread list, the search condition is evaluated. If the search condition is true, the record pointers of the data set records that have been read are written out to the workfile; otherwise, they are ignored and the next data set record is searched.

If a search condition is not needed, the keyword ALL can be used to retrieve all the records.

The FILTER statement expects the workfile to be non-empty. If the workfile is empty and the FILTER statement is executed, an error occurs.

Syntax

```
                      {search_condition }
FILTER USING line_id; {ALL              }
```

Parameters

line_id   Line label on line number that identifies the line that defines the THREAD IS statement.

*search_*   Any logical expression.

condition

Examples

The following examples show the use of the FILTER statement.

```
        400  FILTER USING 300; ALL
        410  FILTER USING Thread_list; TRIM$(Name$)="widgets" AND
               Price < .25
```

FIXED

The FIXED statement sets the default numeric output format to fixed-point and specifies the number of digits to be printed to the right of the decimal point. The FLOAT and STANDARD statements also set the default numeric output format.

Syntax

FIXED *num_expr*

Parameters

num_expr Its rounded value, *n*, must be in the short integer range. When HP Business BASIC/XL outputs a number in fixed-point format, it prints *n* digits to the right of the decimal point. If *n* is less than one, HP Business BASIC/XL prints no decimal point and no decimal digits. If *n* is greater than 16, HP Business BASIC/XL prints 16 decimal digits.

A numeric literal that is expressed in scientific notation can be printed in fixed-point format, but it will be followed with E+*nn* for exponents that are less than two digits andE+*nnn* for exponents that are three digits. Each *n* is a digit.

Examples

```
     10 FIXED 2
     20 PRINT 123;.4567;-79810;-1.235E+47
     99 END
```

The above program prints:

```
     123.00   .46   -78910.00   -1.24E+47
```

If line 10 is changed to

```
     10 FIXED 3
```

then the program prints:

```
     123.000  .457  -78910.000  -1.234E+47
```

FLOAT

The FLOAT statement sets the default numeric output format to floating-point and specifies the number of digits to be printed to the right of the decimal point. The FIXED and STANDARD statements also set the default numeric output format.

Syntax

FLOAT *num_expr*

Parameters

num_expr Its rounded value, *n*, must be in the short integer range. When HP Business BASIC/XL outputs a number in fixed-point format, it prints *n* digits to the right of the decimal point. If *n* is less than one, HP Business BASIC/XL prints no decimal point and no decimal digits. If *n* is greater than 16, HP Business BASIC/XL prints 16 decimal digits.

Floating-point format is appropriate for very large and very small numbers. Floating-point format is

```
              {+}
[-]d [[d ]...]E {-}dd [d ]
```

where *d* is a numeric digit. The leftmost minus sign prints if the number is negative. The decimal point prints unless *n* is zero, and it is followed by *n* digits. To express the number in fixed-point format, raise ten to the power of the exponent (represented by {+|-}*dd* [*d*]) and multiply it by the mantissa (represented by [-]*d* [.*d* [*d*]...]).

Examples

```
     10 FLOAT 2
     20 PRINT 123;.4567;-79810;-1.235E+47
     99 END
```

The above program prints:

```
     1.23E+02    4.57E-01    -7.98E+04    -1.24E+47
```

If line 10 is changed to

```
     10 FLOAT 3
```

then the program prints:

```
     1.230E+02     4.567E-01   -7.981E+04  -1.235E+47
```

FLUSH INPUT

The FLUSH INPUT statement empties the input buffer. The input buffer is a buffer where all the data that you have typed in is stored. If an INPUT statement reads three variables, and you have typed in six, the last three remain in the input buffer. If there is data remaining in the input buffer and another INPUT statement is issued, the INPUT statement will pick up that remaining data. The FLUSH INPUT clears that buffer, so that the next INPUT statement will not use that data.

Syntax

FLUSH INPUT

Examples

```
     100 INPUT A,B,C:  !Extra input for this statement, e.g., 1,2,3,4,5,6
     110 INPUT :D,E,F  !is used by this statement, i.e., D = 4, E = 5, F = 6
     120               !but
     200 INPUT A,B,C:  !extra input for this statement
     210 FLUSH INPUT   !is flushed from the buffer
     220 INPUT :D,E,F  !and not used by this statement.
     999 END
```

FNEND

The FNEND statement defines the end of a multi-line function. A multi-line function returns both control of the execution and a value via a RETURN statement. The FNEND statement serves as a marker for the last statement in the function. Therefore, an error occurs if the FNEND statement actually executes.

Syntax

```
{FNEND }
{FN END}
```

The FNEND statement is legal only in a multi-line function. It is illegal in the main program or a subprogram.

It is good programming practice to end a multi-line function with an FNEND statement and use RETURN statements within the function. However, an FNEND statement can appear more than once in a multi-line function at either the beginning of the next subunit or at the end of the containing program. The start of the next subunit in a program does indicate the end of the multi-line function, but for program documentation purposes it is a good idea to include the FNEND.

Example

```
 10 READ A
 25 DATA 3
 20 C= FNMath(A)         !Calls the function.
 30 PRINT C
 99 END
100 DEF FNMath(X)        !Start of the function.
110 Y=X*2
120 RETURN Y             !Return from the function.
999 FNEND                !Indicates the end of the function.
```

FOR

The FOR and NEXT statements define a loop that is repeated until the value of the loop control variable is greater than or less than a specified value. The value of the loop control variable can increase or decrease.

Syntax

FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*] [*stmt*]...

NEXT *num_var*

Parameters

num_var The numeric loop control variable that assumes the values *num_expr1*, *num_expr1* +*num_expr3*, *num_expr1* +(2**num_expr3*), etc. on successive executions of the loop body. The loop body executes once for each value that is less than or equal to *num_expr2* (if *num_expr3* is positive) or greater than or equal to *num_expr2* (if *num_expr3* is negative).

The *num_var* in the FOR and NEXT statements must be the same.

num_expr1 The initial value that *num_var* is assigned.

num_expr2 Value that *num_var* is compared to before the loop body is executed. If *num_expr3* is positive and *num_var* > *num_expr2*, the loop body is not executed. Similarly, if *num_expr3* is negative and *num_var* < *num_expr2,* the loop body is not executed.

num_expr3 Amount that *num_var* is incremented by (if *num_expr2* is positive) or decremented (if *num_expr2* is negative) following execution of the statements in the loop body.

stmt If *num_expr2* is positive, this statement or statements executes each time *num_var* <= *num_expr2*. If *num_var* is negative, this statement executes each time *num_var* >= *num_expr2*. These statements constitute the loop body.

Examples

```
10  FOR I=2 TO 10 STEP 2      !This will loop 5 times.
20    PRINT "I=",I            !Values are: 2,4,6,8,10
30    PRINT "I+I=",I+I        !Values are: 4,8,12,16,20
40    PRINT "I*I=",I*I        !Values are: 4,16,36,64,100
50  NEXT I
```

```
10  FOR J=4 TO 1 STEP -1      !This will loop 4 times
20    PRINT J                 !Values 4,3,2,1
30  NEXT J
```

```
100 FOR K=1 TO 20             !This will loop 20 times
110   PRINT K                 !Values 1,2,3,...,19,20
120 NEXT K
```

If *num_expr3* is positive, and the first value is greater than the last value, or if *num_expr3* is negative, and the first value is less than the last value, then the loop is never executed.

```
     20 FOR I=10 TO 1
     30   PRINT I      !Never executed
     40 NEXT I
```

FOR constructs can be nested.

```
     100 FOR I=1 TO 3          !Outside FOR loop
     110   FOR J=1 TO 5        !Inside FOR loop
     120     PRINT "*";
     130   NEXT J
     135   PRINT
     140 NEXT I
     999 END
```

The above program prints:

```
     *****
     *****
     *****
```

NOTE An error occurs if nested loops have the same loop control variable (*num_var*).

If you use mixed mode arithmetic on the control variable, the results can be unpredictable. For example, if I is an integer, and the STEP value is .1, STEP is rounded to zero.

Entering from a statement other than the FOR statement the body of a FOR loop causes an error at the NEXT statement, because no FOR statement is active.

GET KEY

The GET KEY statement's action depends on whether a filename parameter is included in the statement. If a filename parameter is included and the file has a BKEY format, then the GET KEY statement defines the fields of the user-definable keys by obtaining all of the required information from the file. If the file specified does not have a BKEY format, an error occurs. If the filename is not specified, then the value of the fields for the user-defined keys is the value of the fields set during the previous GET KEY, SCRATCH KEY, or SAVE KEY statement. The default key set (blank key labels, and the ASCII 7 BEL function) is displayed when GET KEY is issued without first issuing SCRATCH KEY, SAVE KEY or a GET KEY.

Syntax

{GET KEY}
{GETKEY } [*fname*]

Parameters

fname The file that GET KEY uses to obtain information about the user-definable keys. A file name represented by a quoted string literal, an unquoted string literal or a string expression as described in chapter 6.

Examples

```
     GET KEY
     GETKEY "keydef"                           !Uses file Keydef
     100 GETKEY
     200 GETKEY keydef2                        !Uses file Keydef2
     300 GET KEY Filename$ + "." + Groupname$  !Uses the file named in
     310                                       !Filename$ in group Groupname$
```

GLOBAL EXTERNAL

The GLOBAL keyword modifier to the EXTERNAL statement. It allows either the main block or any procedure or function within a program to call an external. It's use and syntax are explained in the EXTERNAL statement.

GLOBAL INTRINSIC

The GLOBAL keyword is a modifier to the INTRINSIC statement. It allows an intrinsic definition to affect every program unit in the program. It's use and syntax are explained in the INTRINSIC statement.

GLOBAL OPTION

The GLOBAL keyword is a modifier to the OPTION statement. It makes selected options global to every program unit in the program. It's use and syntax are explained in the OPTION statement.

GOSUB

The GOSUB statement unconditionally transfers control to a specified line. The line must be in the same program unit as the GOSUB statement. If that line is not executable (for example, if it is a comment or declaration), control transfers to the first executable statement following it. GOSUB can be entered as GO SUB, but HP Business BASIC/XL will always list it as GOSUB.

GOSUB routines can be nested. It is however, a good programming practice to treat these routines as local subunits, that is always using the GOSUB routine to execute them and the RETURN statement to return from them. It is possible to use the GOTO statement into and out of subroutines, but this not a good structured programming practice and should be avoided. The number of levels of nesting is limited. It is set with the COPTION MAXGOSUB. The default for this COPTION is 10.

The RETURN statement returns control to the line following the GOSUB statement (see the RETURN statement in this chapter).

Syntax

{GOSUB }
{GO SUB} *line_id*

Parameters

line_id Line number or line label of the line that control transfers to. It must be in the same program unit as the GOSUB statement.

Examples

```
      10 REM Main Program Unit
      20 GOSUB 90                  !Transfer to line 90
      30 REM Print sides of square
      40 PRINT "|    |"
      50 PRINT "|    |"
      70 GO SUB 90                 !Is listed as GOSUB 90
      80 STOP
      90   REM Subroutine to print top and bottom of the square
     100   PRINT "+----+"
     120   RETURN
     999 END
```

When the above program is listed, line 70 will list as GOSUB 90.

The program prints this square:

```
     +----+
     |    |
     |    |
     +----+
```

The GOSUB statement is a local subroutine call. The local subroutine is not a separate program unit (subunit); it belongs to the program unit that contains it. Parameters cannot be passed to it, but it can access all variables declared in that program unit.

```
      10 REM Main Program Unit
      20 A = 3
      30 GOSUB 100
      40 PRINT A
      50 STOP
     100   REM Subroutine
     110   A = 1
     120   RETURN
     999 END
```

In the above program, line 40 prints 1 (not 3).

GOSUB OF

The GOSUB OF statement is one of the GOSUB corollaries of the ON GOTO and GOTO OF statements. Control transfers to the selected line, L, by "GOSUB L" rather than "GOTO L." A RETURN statement returns control to the statement that follows the GOSUB OF statement. Although the GOSUB OF statement can be input as GO SUB OF, HP Business BASIC/XL always lists it as GOSUB OF.

Syntax

```
{GOSUB }                                    [     {else_line_id }]
{GO SUB} num_expr  OF line_id  [, line_id ]...  [ELSE {CONTINUE    }]
```

Parameters

num_expr — A numeric expression that is evaluated and converted to an integer, *n*. The integer *n* must be between one and the number of *line_id* s, or an error occurs if no ELSE clause is present. Control is transferred to the *n* th *line_id*.

line_id — Line number or line label of the line that control is transferred to. The line must be in the same program unit as the GOSUB OF statement.

else_line_id — Line number or line label of the line that control is transferred to if the value of *num_expr* is not between 1 and the number of *line_ids* specified.

CONTINUE — A keyword used to specify that no branch should be made. The program continues executing at the next line

Examples

```
      1 READ I, J
      2 GOSUB I OF 10,20,30        !Go to subroutine at line 10, 20, or 30
      3 GOSUB J OF 40,50,60 ELSE 99 !Go to subroutine at line 40,50, or 60 or to
                                    !line 99
      4 STOP                        !if J < 1 or J > 3
     10 REM Subroutine for I=1
     11   PRINT "I is one"
     12   RETURN                    !Return to line 3
     20 REM Subroutine for I=2
     21   PRINT "I is two"
     22   RETURN                    !Return to line 3
     30 REM Subroutine for I=3
     31   PRINT "I is three"
     32   RETURN                    !Return to line 3
     40 REM Subroutine for J=1
     41   PRINT "J is one"
     42   RETURN                    !Return to line 4
     50 REM Subroutine for J=2
     51   PRINT "J is two"
     52   RETURN                    !Return to line 4
     60 REM Subroutine for J=3
     61   PRINT "J is three"
     62   RETURN                    !Return to line 4
     90   DATA 3,2
     99 END
```

In the above program, line 2 will be listed as GOSUB I OF 10,20,30.

The ON GOSUB statement works like the GOSUB OF statement. The following statements are equivalent:

```
     150 ON I GOSUB  10, 20, 30, Quit
     150 GOSUB I OF 10, 20, 30, Quit
```

GOTO

The GOTO statement unconditionally transfers control to a specified line. The line must be in the same program unit as the GOTO statement. If the line is not executable (a comment or declaration, for example) HP Business BASIC/XL executes the first executable statement following it.

Syntax

{GOTO }
{GO TO} *line_id*

Parameters

line_id Line number or line label of the line that control transfers to. It must be in the same program unit as the GOTO statement. Although the GOTO statement can be entered as either GO TO or GOTO, HP Business BASIC/XL will always list it as GOTO.

Examples

```
     10 GOTO 30       !Transfer control to line 30
     20 A = 1         !Never executed
     30 A = 2         !Executed immediately after line 10
     40 GO TO Remark  !Transfer control to line 60
     50 PRINT "HI"    !Never executed
     60 Remark:       !Unexecutable statement; execute next line
     70 PRINT A       !Executed after line 40
     99 END
```

Line 40 will list as GOTO Remark.

GOTO OF

The GOTO OF statement transfers control to one of several lines, depending on the value of a numeric expression. Although this statement can be entered as either GOTO OF or GO TO OF, HP Business BASIC/XL will always list it as GOTO OF.

Syntax

```
{GOTO }                                     [     {else_line_id }]
{GO TO} num_expr  OF line_id  [, line_id ]...  [ELSE {CONTINUE    }]
```

Parameters

num_expr A numeric expression that is evaluated and converted to an integer, *n*. The integer *n* must be between one and the number of *line_id* s, or an error occurs if no ELSE clause is present. Control transfers to the *n* th *line_id*.

line_id A line number or line label of a line that control can be transferred to. The line specified must be in the same program unit as the GOTO OF statement.

else line_id The ELSE clause allows the specification of a line that control transfers to if the value of *num_expr* is NOT between 1 and the number of *line_id* s specified.

CONTINUE A keyword that specifies that no branch should be made. Execution will continue at the next line.

Examples

```
100 READ I,J
110 GOTO I+J OF One,Two,Three,Four  ELSE End_program
140 One:   PRINT "One 1"
145 STOP
150 Two:   PRINT "Two 2 2"
155 STOP
160 Three: PRINT "Three 3 3 3"   !Since I+J =3, this is executed
165 STOP                         !Program then stops
170 Four:  PRINT "Four 4 4 4 4"
175 DATA  1,2
900 End_program:                 !Executed if I+J is greater than 4
999 END
```

The ON GOTO statement works like the GOTO OF statement. The following statements are equivalent:

```
150 ON I GOTO 10,200,ReInit,Quit
150 GOTO I OF 10,200,ReInit,Quit
```

GRAD

The GRAD statement indicates that angular units will be specified in Grads. The default is Radians. A Grad is 1/400 of a circle. This statement is used with trigonometric functions.

Syntax

GRAD

Example

```
10 Radius = 10
20 GRAD
30 Area = PI * Radius**2
40 PRINT Area
```

GRAND TOTALS

The GRAND TOTALS statement provides an easy means for automatic accumulation of numeric data in the Report Writer. The GRAND TOTALS statement provides totaling for an entire report.

The GRAND TOTALS statement must appear in the REPORT HEADER, REPORT TRAILER, or REPORT EXIT section. Each report description can have only one such statement.

Syntax

```
                           [{,}         ]
GRAND TOTALS [ON] num_expr  [{;} num_expr ]...
```

Parameters

num_expr Any numeric expression can be totaled. There can be as many expressions as desired. When referring to a particular total, a *sequence* number is used. The first expression is sequence number 1, the second is number 2, and so on.

Examples

```
     100 GRAND TOTALS Sales, Commission, Quantity
```

The BEGIN REPORT statement makes the GRAND TOTALS statement busy and it remains busy until an END REPORT or STOP REPORT statement executes. The GRAND TOTALS statement is used ONLY if contained in a HEADER or TRAILER section with a nonzero level number. BEGIN REPORT sets all accumulated totals to zero.

The GRAND TOTALS calculation occurs when a DETAIL LINE statement executes, but only when the *totals flag* of the DETAIL LINE is nonzero. The accumulated values are reset to zero for any summary level where a break occurs. This is done after the TRAILER sections print. After all break conditions are processed, the totals accumulate.

TOTALS statements are evaluated starting with GRAND TOTALS and working to level nine. For each statement, the expressions are evaluated from left to right.

All totals are stored in either REAL or DECIMAL data type, depending on the data type option in effect when the report started. However, the expressions themselves are evaluated like any other expression in HP Business BASIC/XL. This means that an individual expression can cause an overflow error without causing an overflow in the total.

HEADER

The HEADER statement allows you to define logical levels for separating and summarizing data printed in a report. The HEADER section is used to print headings for a particular level in the report. There are nine levels available.

In order to define a report level, there must be a HEADER or TRAILER statement in the report description. However, there can not be more than one HEADER section for a single level within the report description. If a WITH or USING clause is not present, the statement does not produce output. However, other statements in this section can produce output.

Syntax

```
                    [               [LINES]]
HEADER level_number [WITH num_lines [LINE ]]

[USING image [; output_list ]]
```

Parameters

level_number A numeric expression in the range [0, 9]. This defines the *summary* or *break* level for this header section. This number creates different summary levels for data, and causes breaks in the report at appropriate times. A level of zero causes the entire section to be ignored.

num_lines The maximum number of lines expected to be needed by the section statement. This number reflects ALL output done by the section.

image An IMAGE string or a line reference to an IMAGE line.

output-list A list of output items. This list is identical to the list used by PRINT USING.

Examples

```
     100 HEADER 1 WITH 3 LINES
     100 HEADER Order(1) USING Hd_image;Who
```

The HEADER statement generates an error if there is not an active report.

If a report section is active (executing) and encounters this statement, then that report section ends.

When BEGIN REPORT executes, the *level _number* of each HEADER statement is evaluated. HEADER sections with level numbers equal to zero are ignored. All of the level numbers are fixed by BEGIN REPORT and the HEADER statements become busy. All nonzero HEADER levels must be distinct and within the range of one to nine. The levels do not have to be contiguous. A HEADER statement can define a section without a corresponding TRAILER section and vice versa.

HEADER statements and sections execute when an automatic break occurs from BREAK IF or BREAK WHEN, or when the TRIGGER BREAK statement executes. HEADER sections are printed in ascending sequence by level number. See the DETAIL LINE statement for more details about automatic breaks.

The HEADER sections automatically execute when report output starts. The headers follow the printing of the report header and page header, printing in ascending order.

A particular HEADER section executes the HEADER statement first. This causes evaluation of the WITH clause first (which can cause a page break) followed by the execution of the USING clause. Additional statements in the HEADER section execute after the HEADER statement.

IF THEN or IF THEN ELSE

The IF THEN or IF THEN ELSE statement executes a "then clause" if the evaluated numeric expression is TRUE (nonzero). If the evaluated numeric expression is FALSE (zero), the IF THEN statement transfers control to the statement immediately following it, and the IF THEN ELSE statement executes an "else clause." Each clause is either an executable statement or a line identifier. If it is a line identifier, then execution transfers control to that line. The syntax of this statement requires that the entire statement be contained on one line. The statement can also be used as a command.

Syntax

IF *num_expr* THEN *then_clause* [ELSE *else_clause*]

Parameters

num_expr A numeric expression considered TRUE if it evaluates to nonzero and FALSE if it evaluates to zero.

then_clause Executable program statement or *line_id*. If *num_expr* is TRUE (nonzero), the IF THEN statement executes the executable program statement or transfers control to *line_id*. *line_id* is the line number or line label to which control transfers.

else_clause Executable program statement or *line_id*. If *num_expr* is FALSE (zero), the IF THEN ELSE statement executes the executable program statement or transfers control to *line_id*. *line_id* is the line number or line label to which control transfers.

Examples

```
     10 IF A=B THEN C=3          !Contains executable statement (C=3)
     20 IF X=Y THEN GOTO 40      !Contains executable stmt (GOTO 40)
     21 IF X=Y THEN 40           !Contains line identifier (40)
     30 IF J<>0 THEN Initialize  !Contains line identifier (Initialize)
```

Lines 20 and 21 above are equivalent.

IF THEN ELSE Construct

The IF THEN ELSE construct is an alternate form of the IF THEN ELSE statement. The IF THEN, ELSE, and ENDIF keywords define a construct that executes one statement or set of statements if a numeric expression is TRUE (nonzero) and another statement or set of statements if that numeric expression is FALSE (zero).

Syntax

```
                                     [ELSE              ]
                                     [[else_clause_stmt ]] {ENDIF }
IF num_expr  THEN [then_clause_stmt ]...[.                ] {END IF}
                                     [.                 ]
                                     [.                 ]
```

Parameters

num_expr A numeric expression that is considered TRUE if it evaluates to nonzero; FALSE if it evaluates to zero.

then_clause_ stmt One or more program lines that execute if *num_expr* is TRUE. These statements comprise the THEN clause.

else_clause_ stmt One or more program lines that execute if *num_expr* is FALSE. These statements comprise the ELSE clause.

After either the IF or ELSE clause executes, control transfers to the line following the ENDIF statement.

Examples

```
     10 IF I THEN                    !The IF THEN portion of the construct
     20    PRINT "I IS NOT ZERO"     !The THEN clause statement
     30 ELSE
     40    PRINT "I IS ZERO"         !The ELSE clause statement
     50 ENDIF

     20 IF A=B THEN
     30   PRINT "A=B"                !The THEN clause statements --
     40   PRINT A                    !will execute if A = B
     50 ELSE
     60   PRINT "A<>B"              !The ELSE clause statements --
     70   PRINT A,B                  !will execute if A <> B
     90 ENDIF
```

A statement in the IF or ELSE clause can transfer control out of the IF THEN ELSE construct.

```
     105 IF (K+J)*I=0 THEN
     110   PRINT "OK"
     115   GOTO 200             !Control transfers to line 200
     120 ELSE
     125   PRINT "NOT OK"
     130   GOSUB Error-routine  !Control transfers to Error-routine
     135 END IF
```

IF THEN ELSE constructs can be nested; that is, the IF or ELSE clause of one IF THEN ELSE structure can contain another IF THEN ELSE construct. The ENDIF is associated with the most recently preceding IF THEN ELSE construct.

```
     100 IF I THEN                        !Begin outer construct
     110   IF J THEN                      !Begin inner construct
     120     PRINT "I and J are not 0"
     130   ELSE                           !ELSE for inner construct
     140     PRINT "J is 0 but I is not"
     150   ENDIF                          !End inner construct
     160 ELSE                             !ELSE for outer construct
     170   PRINT "I is 0"
     180 ENDIF                            !End outer construct
```

The ELSE clause can be omitted.

```
     406 IF Number_left THEN
     407   PRINT "There are numbers left"
     408   STOP
     409 END IF
```

Control transfer into a THEN or ELSE clause, but this is not a recommended programming practice.

IMAGE

The IMAGE statement specifies the output format for the output items in the display list of a DISP USING or PRINT USING statement. If the image of a DISP USING or PRINT USING statement is a line identifier, the line identifier must identify an IMAGE statement. Because an IMAGE statement can end in an unquoted string literal, it cannot be followed by a comment. The IMAGE statement is not executable.

Syntax

IMAGE *format_string*

Parameters

format_string — *format_string* (if it belongs to an IMAGE statement) or its value (if *format_string* itself is the image of a PRINT or DISP statement) has the following syntax:

format_spec [, *format_spec*]...

[*num_expr*] (*format_spec* [, *format_spec*]...)

format_spec — One of the format specifiers described in "Format Specifiers" in chapter 6.

num_expr — Repeat factor. Rounded to a short integer, *n*. The *format_string n* (*format_spec_list*) is equivalent to *n* adjacent copies of *format_spec_list* (see examples).

Examples

```
     100 DISP USING 110; A,B,C
     200 PRINT USING 210; A,B,C
     300 DISP USING 310; P,Q
     400 PRINT USING 410;A,R
```

The IMAGE statements that they reference are:

```
     110 IMAGE DDD,XX,DDD,XX,DDD,XX
     210 IMAGE 3 (DDD,XX)
     310 IMAGE DDDDD,XX,ZZZ.DD
     410 IMAGE 5D,2X,3Z.DD
```

The format strings of lines 110 and 210 are equivalent, as are the format strings of lines 310 and 410. In line 210, three is the repeat factor represented by *num_expr*, above. In line 410, the numbers 5, 2, and 3 are also called repeat factors; "Digit Symbols" and "Space Specifications" in chapter 6 explain them.

IN DATASET

The IN DATASET statement specifies the record format of a particular data set. It is used to unpack data after the data is retrieved from a database by a SORT, SEARCH, or DBGET statement. It is also used to specify how data is packed for use by DBPUT and DBUPDATE statements.

The record format of a data set is required in order to accurately compute the location of the sort key and to evaluate the search condition. Therefore, a program must contain IN DATASET statements to SEARCH or SORT a database. When used, this statement must correspond to the record layout of the data set in the database.

If a string, string array, or numeric array is used as a formal parameter in an IN DATASET statement a compile time error will occur when that parameter is referenced before the sorted key in a SORT statement.

Syntax

IN DATASET *dataset* USE [REALV] *item_list*

Parameters

dataset A string expression with a maximum length of 16 characters. Its value is the name of a data set. The name must be left-justified and, if shorter than 16 characters, must be terminated by a semicolon or blank.

REALV The default real data type in a native mode program is in IEEE floating point real format. Therefore, REALV must be specified in the IN DATASET statement if the MPE V real data format is desired.

item_list A list of any of the following separated by commas:
- Scalar numeric variable
- Scalar string variable
- Substring
- String or numeric array
- String or numeric literal
- A numeric literal type converted with one of the following built-in functions:
  - SINTEGER
  - INTEGER
  - SREAL
  - REAL
  - SDECIMAL
  - DECIMAL

Space specifier: SKIP *number*, where *number* is a numeric constant.

Examples

The following examples show the use of the IN DATASET statement.

```
     300    IN DATASET Dset$ USE A, B, SKIP 4, D$
     400    IN DATASET Dset$ USE 3.019,"Super",SREAL(1)
```

The SKIP feature is used to bypass data in a dataset record that is not needed by the program. The numeric constant that immediately follows SKIP specifies the number of bytes to bypass. There must be an IN DATASET statement for each data set defined in the thread list. Refer to the THREAD IS statement description below for details about the thread list. The IN DATASET statement is a nonexecutable statement and is treated internally like a PACKFMT statement.

INPUT

The INPUT statement assigns data values obtained from the terminal or a file to one or more variables.

Syntax

```
         [           [{,}           ]   ]
INPUT[:] [input item  [{;} input_item ]...] [:]
```

Parameters

:   A colon specifies that either data currently in the input buffer should be assigned prior to prompting for input or extra input is saved in the input buffer.

- If a colon precedes the *input_items*, the INPUT statement assigns values in the input buffer to *input_elements*, from left to right, before prompting you for input or reading input from a file. If this colon is not specified, HP Business BASIC/XL empties the input buffer before accepting input values.

- If a colon follows the *input_item* s, the INPUT statement stores unassigned input values that are not required to satisfy assignments to the *input_items* in the input buffer.

```
                 {[prompt_option ] input_element }
input_item        {for_clause                    }
```

An INPUT statement without *input_items* puts the program in the input state until you press RETURN, but the values entered are not assigned to any *input_element* s.

```
                 {PROMPT str_expr }[,]
prompt_option     {str_lit         }[;]
```

If the prompt is not followed by a separator or a comma, a carriage return is generated and user input begins on the next line. Semicolons suppress the carriage return and input can be typed on the same line as the prompt. See "INPUT Prompt" in chapter 6 for more information.

input_element   One of the following variables that a value is assigned to:

num_var
str_var
array_name ([*[,*]...])

The last format above has either zero or one asterisk per dimension. The absence of asterisks specifies any number of dimensions. Either format is legal, but the format without asterisks is noncompilable.

for_clause   (FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *input_item* [, *input_item*]...)

A *for_clause* is useful for reading array elements (an array can also be input with the MAT INPUT statement). See "FOR Clause in Input List" for more information.

Examples

The following examples show several ways to use the INPUT statement.

```
     INPUT
     INPUT A,B$,C(*)
     INPUT A,B$,C(*):
     INPUT:
     INPUT: A,B$,C(*)
     INPUT: A,B$,C(*):
     INPUT A$ A,B$,C(*)
     INPUT PROMPT D$; X,Y, PROMPT D1$+D2$, Z
     INPUT "Input 2 numbers",X,Y,PROMPT D1$+"A";Z,"Input name",N$:
     INPUT: (FOR I=1 TO 10, W(I), WW(I,I))
     INPUT "Input A and elements of V"; A, (FOR J=1 TO 5, V[J]):
```

An INPUT statement that begins with a colon assigns the values in the input buffer before prompting you for input or reading it from a file.

An INPUT statement that ends in a colon stores unassigned input values in an input buffer.

An INPUT statement that does not begin or end with a colon empties the input buffer before prompting for input.

FOR Clause in Input List

An input list can contain a FOR clause. The FOR clause is similar to the FOR NEXT construct.

Syntax

(FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *input_item*

[,*input_item*]...)

Parameters

num_var Assigned the sequence of values: *num_expr1*, *num_expr1* +*num_expr3*, *num_expr1* +(2**num_expr3*), etc. The INPUT statement reads one input value for each value of num_var that is less than *num_expr2* (if *num_expr3* is positive) or greater than *num_expr2* (if *num_expr3* is negative).

num_expr1 First value assigned to *num_var*.

num_expr2 Value that *num_var* is compared to before the INPUT statement reads a value. If *num_expr3* is positive and *num_var* > *num_expr2*, loop execution is terminated. If *num_expr3* is negative and *num_var* < *num_expr2*, the loop execution is terminated.

num_expr3 Amount that *num_var* increases by at the end of the loop. The default = 1.

input_item Same as *input_item* in INPUT statement syntax.

Examples

```
     10 INPUT "Input 4 numbers: ", (FOR I=1 TO 4, A(I)), "Input X: ", X
```

If you input the underlined values during execution:

```
     Input 4 numbers:
```

```
     10, 20, 30, 40
     Input X:
     50
```

Following execution of line 10, the values assigned to each variable will be:

```
     A(1) = 10
     A(2) = 20
     A(3) = 30
     A(4) = 40
     X    = 50
```

Input list FOR clauses can be nested.

```
     20 INPUT (FOR I=1 TO 3, (FOR J=1 TO 2 (FOR K=1 TO 2, B(I,J,K))))
```

For each combination of values of I, J, and K, the following table shows the value that the above INPUT state assigns to each variable.

Value of I	**Value of J**	**Value of K**	**Variable Read**
1	1	1	B(1,1,1)
1	1	2	B(1,1,2)
1	2	1	B(1,2,1)
1	2	2	B(1,2,2)
2	1	1	B(2,1,1)
2	1	2	B(2,1,2)
2	2	1	B(2,2,1)
2	2	2	B(2,2,2)
3	1	1	B(3,1,1)
3	1	2	B(3,1,2)
3	2	1	B(3,2,1)
3	2	2	B(3,2,2)

INTEGER

This statement defines a variable of type INTEGER. If the SHORT option is included, the variable is of type SHORT INTEGER.

Syntax

```
                  {num_var } [ {num_var }]
[SHORT] INT[EGER] {arrayd  } [,{arrayd  }]...
```

Parameters

num_var Name of scalar numeric variable to be declared.

arrayd Numeric array description. The syntax for the array is described under the DIM statement.

Examples

```
     100 SHORT INTEGER I
     120 SHORT INTEGER A,B(6,9),Sum
     130 INTEGER Total
     140 INTEGER Var1,Var2,Var3(1,2,3),Var4(1:10,1:10)
```

INTRINSIC and GLOBAL INTRINSIC

The INTRINSIC or GLOBAL INTRINSIC statement defines procedures or functions that are not in the current program without requiring an explicit definition of the entire procedure or function heading. The external procedure or function either can be in an executable library or can be linked with the current program after the current program is compiled. A GLOBAL INTRINSIC statement must appear in the main program. These intrinsics can be called from the main or any procedure or function in the current program. An INTRINSIC statement defines intrinsics local to the program unit that the definition occurs in. Local definitions supersede global definitions.

Syntax

```
[GLOBAL] INTRINSIC [("fname ")] identifier  [ALIAS str_lit ]

[{,}                          ]
[{;} identifier  [ALIAS str_lit ]]...
```

Parameters

GLOBAL Allowed only if the statement is in the main program. If GLOBAL appears, the statement is a GLOBAL INTRINSIC statement. If GLOBAL is omitted, the statement is an INTRINSIC statement. A GLOBAL INTRINSIC statement affects every program unit in the program. An INTRINSIC statement affects only the program unit that contains it.

Information supplied in an INTRINSIC statement overrides information in a GLOBAL INTRINSIC statement, while the program unit that contains the INTRINSIC statement is executing.

fname Intrinsic file that contains the definitions of the intrinsics in the list of intrinsics that follow. The default is the default intrinsic file of the operating system (SYSINTR.PUB.SYS on MPE XL).

identifier Internal name; name that HP Business BASIC/XL program uses to call this intrinsic. If the intrinsic is a function and the program calls it without the FNCALL function, then *identifier* must be a legal function name;

that is, it must begin with FN, as in FNAdd. The actual name to use for the call is returned from the definition in the intrinsic file.

str_lit The alias is the name, if different from the name to be used in the HP Business BASIC/XL program, of the intrinsic in the *fname* file. The string provided is assumed to be the case-sensitive name of the intrinsic file entry. The actual name to use for the call is returned from the definition in the intrinsic file.

Examples

The following examples show the use of the INTRINSIC statement. Lines 10, 20, and 50 show the GLOBAL option. Lines 30, 40 and 80 specify file names, and the rest use the operating system default. Lines 50, 60, and 70 specify the actual procedure or function name with the ALIAS keyword when the actual name is different than the internal HP Business BASIC/XL name.

```
     10 GLOBAL INTRINSIC Findjcw     !Entry searched for is:
     15                              !FINDJCW in SYSINTR.PUB.SYS
     20 GLOBAL INTRINSIC Fmtcalendar,Command,Read_char
     30 INTRINSIC ("FILE1.LAB") Findjcw,Fmtcalendar,Command
     40 INTRINSIC ("FILE2.MKTG") Put_char;Put_block;Open_file
     50 GLOBAL INTRINSIC FNFind ALIAS "Find"
     60 INTRINSIC FNStore ALIAS "Store",FNRetrieve ALIAS "Retrieve"
     70 INTRINSIC FNAdd ALIAS "Add";FNSub ALIAS "Subtract"
     80 INTRINSIC ("File3") Print_file ALIAS "print_file_info"
```

LDISP

The LDISP statement provides an alternative form of output for the DISP statement. Under normal circumstances, the LDISP statement clears the current line before printing the output list. The screen line clears from the cursor to the end of the line. Note that only one line clears even if multiple lines prints. LDISP interacts differently with an active JOINFORM. If the cursor is within the form, LDISP will move the cursor to the first line after the form, clear the line and then print. For more information about how LDISP interacts with JOINFORM, refer to Appendix F.

Syntax

```
                         [,]
LDISP [output_item_list ] [;]
```

Parameters

output_item_ list [,]...*output_item* [{ [,]...} *output_item*]...]

output_item One of the following:

num_expr

str_expr

array_name (*) Array reference. See "Array References in Display List" in chapter 6.

```
                                {PAGE            }
                                {{CTL}           }
              output_function    {{LIN}           }
                                {{SPA} (num_expr )}
                                {{TAB}           }
```

See "Output Functions in Display

List" in chapter 6.

FOR_clause (FOR *num_var =num_expr1* TO *num_expr2* [STEP *num_expr3*], *d_list*)

See "FOR Clause in Display List" in the DISP and PRINT statements in this chapter.

Examples

```
     10 V$="Hi there."
     20 DISP V$
     30 LDISP V$
```

In the above example, if you type RUN and the screen already has characters on the next two lines:

```
        >RUN
        12345678901234567890
        12345678901234567890
```

then following program execution, the screen contains

```
        >RUN
        Hi there.01234567890
        Hi there.
```

LEFT MARGIN

The LEFT MARGIN statement is a Report Writer statement that defines the column in which a report line will start printing. This allows you to adjust the left margin on the output device. The MARGIN statement adjusts the right margin.

The LEFT MARGIN statement does not apply to terminal output. The output is adjusted if the standard output is redirected to a non-terminal device such as a printer. The COPY ALL OUTPUT statement, if applicable, reflects the left margin of the standard output.

There cannot be more than one LEFT MARGIN statement in a report description.

Syntax

LEFT [MARGIN] *column*

Parameters

column The column that the first character of a report line is in. That is, *column* - 1 spaces appear on the left of each line. The left margin column must have a value between 1 and 132.

Examples

The following examples show the LEFT MARGIN statement.

```
     100 LEFT MARGIN 10   !First column is 10, preceded by 9  blank spaces
     100 LEFT MARGIN 35   !First column is 35, preceded by 34  blank spaces
```

The LEFT MARGIN statement is evaluated only by BEGIN REPORT and is busy only during evaluation.

The default value is 1 if there is no left margin statement. The distance between the left and right margins must be at least 20 characters, or an error occurs. This is checked at BEGIN REPORT and whenever the right margin changes.

When report output is done, all output is preceded by *column* -1 spaces. However, the left margin only applies if the output device is not a terminal. For terminal devices, the left margin is always 1.

The left margin applies to both the standard output file and the COPY ALL OUTPUT file, if output is being copied. If the left margin is too large for the COPY ALL OUTPUT file or for the standard output file, there is an error in BEGIN REPORT.

LENTER

The LENTER statement assigns all or part of a line of display memory to a string variable. User input from the keyboard is not accepted.

The assigned string value begins at the cursor position and ends at the rightmost column for that line in display memory. Commas are read as characters, not as data item separators.

LENTER interacts in a special way with an active JOINFORM. This is described in detail in Appendix F.

Syntax

LENTER *str_var*

Parameters

str_var A string variable that will accept the data. An error occurs when the input string value exceeds its maximum length. If this value is a substring, and the input string value exceeds its length, the input string value is truncated on the right (no error occurs). See "Substring References" in chapter 3 for more information.

Examples

The following examples show the use of the LENTER statement.

```
320 LENTER Str_var$
330 LENTER Sub_str$ [2;3]
340 LENTER Sub_str$[1]
```

LET

The LET statement assigns a value to one or more variables.

Syntax

```
      {num_var  [, num_var ]...= num_expr }
[LET] {str_var  [, str_var ]...= str_expr }
```

Parameters

num_var Numeric variable(s) that the value of *num_expr* is/are assigned.

str_var String variable(s) that the value of *str_expr* is/are assigned.

num_expr Value that *num_var* will contain. This can be either a literal, or an expression.

str_expr Value that *str_var* will contain. This can be either a literal or an expression.

Examples

```
     10 LET Number=3                    !Assignment:  3, to Number
     20 Num1, Num2, Num3=4+6            !Assignment:  10, to Num1, Num2, and Num3
     30 String$="cat"                   !Assignment:  "cat" to String$
     40 LET Str1$,Str2$= "Ab" + "CdE" !Assignment:  "AbCdE", to Str1$, and Str2$
```

HP Business BASIC/XL accesses variables in LET statements from left to right. If variables have not been declared, and implicit declaration is illegal, an error occurs. If no error occurs, HP Business BASIC/XL evaluates the expression and assigns its value to the variables, from right to left. If the value is numeric, HP Business BASIC/XL converts it to the type of each of the variables prior to assigning it to the variables.

```
     10 OPTION NODECLARE    !Implicit declaration is legal
     20 OPTION REAL         !Default numeric type is real
     30 INTEGER A           !Integer A is explicitly declared
     40 DECIMAL B           !Decimal B is explicitly declared
     50 LET A,B,C=(5+4)*3   !Real C is implicitly declared
     99 END
```

In line 50, HP Business BASIC/XL does the following:

- Accesses A, B, and C in that order
- Evaluates (5+4)*3
- Assigns the following values in the following order:

To:	**The Value:**
C	real 27.0
B	decimal 27.0
A	integer 27

When HP Business BASIC/XL converts a numeric value to a numeric variable type that has fewer significant digits than the value does, it rounds the value first. An error occurs if the value is outside the range of the variable type. An error also occurs if an assigned string value is too long for it's string variable (that is, if the length of the string value exceeds the maximum length of the string variable).

If an assignment statement has more than one variable to the left of the equal sign, for example; A,B,C=5, and an error occurs in the middle of the assignment statement, the variables after (or to the right of) the error contain the new value. The variables before (or to the left of) the error do not. The variable in which the error occurred does not contain a new value.

In the example below, an error occurs when 80,000 is assigned to C in line 30 (C, a short integer can have a maximum value of 32767). D and E are assigned the value 80,000, but A, B, and C still have the value zero following the error.

```
     10 SHORT INTEGER C
     20 A,B,C,D,E=0
     30 A,B,C,D,E=80000
     99 END
```

Multiple Assignment Statement

The multiple assignment statement is a series of LET statements, separated by semicolons. The LET keyword can only appear in the first LET statement.

Syntax

LET_stmt [; *LET_stmt*]...

Parameters

LET_stmt A LET statement

Example

```
     10 LET A,B=5; C$="HI";D=4+2
```

LINPUT

LINPUT statement execution places the program in the input state and assigns a string value obtained from the terminal or input file to a single string variable. The string value accepted is an unquoted string literal. Double quotes are characters. Unlike the INPUT statement, the LINPUT statement includes the leading and trailing blanks as part of the string value. Commas and semicolons are not recognized as item separators or terminators, but are characters. LINPUT also reads one record of an ASCII file into a string variable.

Syntax

LINPUT [*prompt_option*] *str_var* LINPUT *#fnum* [, *rnum*]; *str_var*

Parameters

prompt_option The LINPUT statement displays its prompt the same way and under the same conditions as the INPUT statement. See the *prompt_option* parameter under the INPUT statement for more information.

str_var A string variable. An error occurs if the variable is a string rather than a substring and the input string value exceeds the string variable's maximum length. If the variable specified is a substring, and the input string value exceeds its length, the input string value is truncated on the right (no error occurs). See "Substring References" in chapter 3 for more information.

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer.

rnum Record number, a numeric expression. A file I/O statement that specifies *rnum* is direct; otherwise, it is sequential.

Examples

```
     05 B$= "Please enter A$ "
    10 LINPUT A$                  !Prints a question mark (?) and a carriage return.
     20 LINPUT PROMPT B$; A$         !Prints "Please enter A$"
    30 LINPUT PROMPT B$+": ", A$    !Print "Please enter A$ :" and a carriage return
     40 LINPUT "Enter A$: "; A$[1,3] !Prints "Enter A$:"
```

If the data from the record exceeds the maximum length of the string variable, an error occurs if *str_var* is a string (rather than a substring). For example, an error occurs at line 140 of the following sequence:

```
     120 DIM C$[8]
     130 PRINT #1,1; "more than eight"
     140 LINPUT #1,1; C$
```

If *str_var* is a substring, then the record data is truncated on the right. For example, there is no error in the above sequence if line 140 is replaced with:

```
     140 LINPUT #1,1; C$[1;8]
```

LOCK

The LOCK statement requests exclusive access to a file, for the program that executes the lock statement. If the file cannot be accessed at the time the LOCK statement is executed, an option can be specified to delay execution of the LOCK statement until the program has exclusive access.

Syntax

LOCK *#fnum* [; WAIT *num_var*]

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer.

num_var A numeric variable that contains a file locking flag. Two conditions occur dependent on the value assigned to *num_var* prior to the LOCK statement:

* Zero: File unlocking occurs unconditionally. If the file is being accessed by another program, execution of the LOCK statement is suspended until the file can be locked.

* Non- Zero: File locking occurs only if the file is not currently locked. If the file is locked, program execution resumes without locking the file.

If the file is successfully locked, the value one is assigned to *num_var*. If the value of num_var prior to the call is nonzero, then an unsuccessful attempt to lock the file results in zero being assigned to *num_var*.

Examples

```
     100 CREATE "File1",FILESIZE=1200  !Creates File1
     200 ASSIGN "File1" TO #10         !Assigns File1
     300 LOCK #10                      !Locks File1
     400 PRINT #10; A,B,C
     500 UNLOCK #10                    !Unlocks File1 after printing
     999 END
```

LOOP

The LOOP, EXIT IF, and ENDLOOP statements define a loop that repeats until the numeric expression in the EXIT IF statement is TRUE (nonzero).

Syntax

```
              [EXIT IF num_expr ]
              [[stmt ]          ]   {ENDLOOP }
LOOP [stmt ]...[.               ]...{END LOOP}
              [.               ]
              [.               ]
```

Parameters

stmt A program line that can be another LOOP statement. These statements constitute the loop body.

num_expr A numeric expression that determines program control. Considered FALSE if the value following evaluation is zero, TRUE if it evaluates to nonzero. If FALSE, control is transferred to the line following the EXIT IF statement; if TRUE, control is transferred to the line following the ENDLOOP statement. If the loop does not contain an EXIT IF statement, and control is not transferred out of the loop by some other means (for example, a GOTO statement) the loop never ends.

Examples

```
100 LET I=0                  !Initialize I
110 LOOP                     !Begin loop
120   PRINT I                !Print I (at this line, I=0,1,2,...,99)
130   LET I=I+1              !Increment I (at this line, I=1,2,3,...,100)
140   EXIT IF I=100          !If I=100, go to line 160; else go to line 120
150 ENDLOOP                  !End loop
160 PRINT I                  !Print I (at this line, I=100)
999 END
```

```
100 READ I                   !Read number to be guessed, I
101 Low=1                    !Lowest possible guess
102 High=100                 !Highest possible guess
103 Tries=0                  !Number of tries to guess I
110 LOOP
111   Tries=Tries+1          !Count one for guessing I=Low in 120
120   EXIT IF I=Low          !If Low=I, go to 230; else go to 121
121   Tries=Tries+1          !Count one for guessing I=High in 130
130   EXIT IF I=High         !If High=I, go to 230; else go to 140
140   Guess=(Low+High)/2     !Guess average of Low and High
145   Tries=Tries+1          !Count one for guessing I=Guess in 150
150   EXIT IF I=Guess        !If Guess=I, go to 230; else go to 160
160   SELECT Guess-I         !If I<>Guess, reset Low or High
170   CASE < 0               !If Guess < I,
180     Low=Guess            !then Guess is the new lowest guess.
190   CASE > 0               !If Guess > I,
200     High=Guess           !then Guess is the new highest guess.
210   END SELECT
220 END LOOP
230 PRINT Tries              !Print number of tries needed
250 DATA 47
999 END
```

Loops can be nested. An EXIT IF statement in a nested loop belongs to the innermost loop that contains it.

```
 1 Num_row=4
 2 Num_col=5
10 Row=1
11 LOOP                           !Begin outer loop
12   Column=1
13   LOOP                         !Begin inner loop
14     PRINT A(Row,Column)
15     Column=Column+1
16     EXIT IF Column=(Num_col+1) !Exit inner loop (go to line 18)
17   ENDLOOP                      !End inner loop
18   PRINT
19   Row=Row+1
20   EXIT IF Row=(Num_row+1)      !Exit outer loop (go to line 99)
21 ENDLOOP                        !End outer loop
99 END
```

Entering a LOOP construct from a statement other than the LOOP statement is considered to be a bad programming practice, and is not recommended. However, calling a local subroutine using GOSUB or calling an external subroutine using CALL from within a loop construct can be very useful.

```
100 I=0
110 LOOP             !Begin loop
130   EXIT IF I=100
140   GOSUB 200      !Leave loop for subroutine
```

```
     145   I=I+1          !Reenter loop here
     150 END LOOP         !End loop
     160 STOP
     200 REM Subroutine   !Begin subroutine
     210   PRINT I
     220 RETURN           !Return to loop
     999 END
```

If a program unit contains an EXIT IF statement that is not in a loop, an error occurs.

MARGIN

The MARGIN statement sets the margin for the terminal screen or for an ASCII file. Also, see the MARGIN option described in the "Device Specification Syntax" section of chapter 6.

Syntax

MARGIN [*#fnum*;] *num_expr*

Parameters

fnum — A file number that HP Business BASIC/XL uses to reference the file for which the margin is to be set. This is a numeric expression that evaluates to a positive short integer greater than zero. If it is not an ASCII file, the MARGIN statement has no effect. The default is the terminal screen.

num_expr — Maximum length of an output line on the terminal screen or in the ASCII file, provided that *num_expr* does not exceed the maximum length of a screen line, usually 80 characters, or the record length of the file. If *num_expr* does exceed the maximum length of a screen line, the margin is the maximum length instead of *num_expr*. If *num_expr* exceeds the file's record length, the margin is the record length instead of *num_expr*.

An output line that is longer than the physical margin allows overflows onto the next physical line.

Examples

The following examples show the use of the MARGIN statement. Lines 10 and 40 set the margin for the default *fnum*, the terminal screen.

```
     10 MARGIN 40
     20 MARGIN #2; Num_char_to_right_hand_margin
     30 MARGIN #1; X-5
     40 MARGIN Terminal_line_length
```

MAT =

The MAT = statement assigns the value of an expression to an array. Some forms of the MAT statement can redimension the array before the assignment.

Syntax

The numbers preceding these syntax specifications are referenced in Table 4-5. They are not part of the MAT statement syntax.

(1) MAT *num_array1* = *num_array2*

(2) MAT *num_array1* = (*num_expr*)

(3a) MAT *num_array1* = *num_array2 op* (*num_expr*)

```
(3b) MAT num_array1  = (num_expr ) op num_array2

(4) MAT num_array1  = num_array2 op num_array3

                      {CON}
(5) MAT num_array1  = {ZER} [(dims )]
                      {IDN}

(6) MAT num_array1  = s_or_a_function  (num_array2 )

(7) MAT num_array1  = array_function  (num_array2 )

(8) MAT num_array1  = MUL (num_array2, num_array3 )
```

Parameters

op +, -, *, /, <, <=, =,>=, <>, or #

num_array1 In equation (4), *num_array1* must have the same number of dimensions as *num_array2* and *num_array3*. It must have at least as many elements as each of *num_array2* and *num_array3*.

num_array3 In equation (4), *num_array3* must have the same number of dimensions as *num_array2*. Each dimension of *num_array3* must have the same number of elements as the corresponding dimension of *num_array2*. However, corresponding dimensions of *num_array3* and *num_array2* can have different bounds (for example, *num_array2* can be declared "DIM A(1:2,1:4)" and *num_array3* can be declared "DIM B(2:3,2:5)").

In equation (8), *num_array2* and *num_array3* can both be matrices, or one can be a matrix and one can be a vector. The dimensions of *num_array2* and *num_array3* are subject to the restrictions in Table 4-6.

CON Sets each element of *array* to one.

ZER Sets each element of *array* to zero.

IDN Makes *array* an identity matrix. If *dims* is specified, it must specify a square matrix. If *dims* is not specified, *array* must be a square matrix.

dims If specified, the statement redimensions *num_array1* before assigning values to its elements.

s_or_a_function A scalar or array function; one of the following:

```
ABS       ACS        ASN       ATN     CEIL      LGT
DECIMAL   EXP        FRACT     INT     INTEGER   LOG
REAL      SDECIMAL   SGN       SIN     SINTEGER  SQR
SREAL     TAN        TRUNC     COS
```

See chapter 5 for more information about these functions.

array_function See chapter 5 and Table 4-6 for more information about these functions.

CSUM Stores column sums of matrix in vector.

RSUM Stores row sums of matrix in vector.

TRN Transposes rows and columns of

	matrix.
INV	Inverts square matrix.
MUL	Multiplies two matrices or a vector and a matrix.

Table 4-5 through Table 4-6 give more information about the MAT = statement.

Table 4-5 Gives the new dimensions of and value of *num_array1* for each form of the MAT = statement.

Table 4-6 Shows how the dimensions of *num_array2* and *num_array3* determine the new dimensions of *num_array1*.

Table 4-5. Forms of MAT = Statement

Form	**Redimensions *num_array1* to Dimension of:**	**Where *num_array1*(*i*) and *num_array2*(*i*) are Corresponding Elements: *num_array1*(*i*)=**
1	*num_array2*	*num_array2* (*i*)
2	Does not redimension *num_array1*	*num_expr*
3a	*num_array2*	*num_array2* (*i*) *op num_expr*
3b	*num_array2*	*num_expr op num_array2* (*i*)
4	*num_array2*, *num_array3* (same)	*num_array2* (*i*) *op num_array3* (*i*)
5	Specified dimensions, if any	ZER: 0 CON: 1 IDN: 1 if it is on the top-left-to-bottom-ri|ht diagonal; 0 otherwise
6	*num_array2*	*scalar_or_array_function* (*num_array2*)
7 8	See Table 4-6	See Table 4-6

Table 4-6. Dimensions of Array Function Arguments and Results

Array Function	Dimensions of *num_array2*	Dimensions of *num_array3*	Dimensions of *num_array1* (result)
CSUM	(*m,n*)	Not applicable	(*n*)
RSUM	(*m,n*)	Not applicable	(*m*)
TRN	(*m,n*)	Not applicable	(*n,m*)
INV	(*m,m*)	Not applicable	(*m,m*)
MUL	(*m,n*)	(*n,p*)	(*m,p*)
MUL	(*m,n*)	(*n*)	(*m*)
MUL	(*m*)	(*m,p*)	(*p*)

Examples

```
 10 DIM A(4),B(4),C(4),D(4),E(2,4),F(2)
 20 READ (FOR I=1 TO 4,A(I))
 30 READ (FOR I=1 TO 2,(FOR J=1 TO 4,E(I,J)))
 40 !
 50  !  Form 1:
 60 MAT B=A           !B has the same elements as A, B(1) = A(1), etc
 70 !
 80 !  Form 2:
 90 MAT C=(2+3)       !All elements of C have the value of 5
100 !
110 !  Form 3:
120 MAT D=(2)*B       ! All elements of D are worth 2 * B, D(1) =20 ,etc
130 MAT D=B*(2)       !Alternate form 3b, results are the same as line 61
140 !
150 !  Form 4:
160 MAT C=A+B         !Each element, I of C is the total of A(I) + B(I)
170 !
180 !  Form 5:
190 MAT B=CON         ! Each element of B is now 1
200 !
210 !  Form 6:
220 MAT D=SQR(A)      ! Each element, I of D is now the square root of A(I)
230 !
240 !  Form 7:
250 MAT C=CSUM(E)     ! Each element, I of C is now the sum of the entries
260                   ! in column I of E
270 !
280 !  Form 8:
290 MAT F=MUL(E,A)    ! Array F contains the result of the matrix
300                   ! multiplication of E and A
310 !
```

```
        320 !
        330 DATA 10,20,30,40
        340 DATA 1,2,3,4,5,6,7,8
        999 END
```

MAT INPUT

The MAT INPUT statement accepts values from the terminal keyboard to one or more arrays. If new dimensions are specified for the arrays, the MAT INPUT statement redimensions them before assigning values to them. It assigns values element by element, in row-major order.

Syntax

MAT INPUT *array* [*dims*][, *array* [*dims*]]...

Parameters

array Structured collection of variables of the same type. The structure is determined when the array is declared. String variables names are suffixed with a "$".

dims Array dimensions used in syntax specification statements. Its syntax is

(*dim1* [,*dim2* [,*dim3* [,*dim4* [,*dim5* [,*dim6*]]]]])

where *dim1* through *dim6* each have the syntax

[*num_expr1*:]*num_expr2*

and *num_expr1* and *num_expr2* are the lower and upper bounds (respectively) of the dimension. If *num_expr1* is not specified, it is the default lower bound.

Examples

```
     100 MAT INPUT A,B,C$
     120 MAT INPUT D$
```

If array A has four elements, the following statements are equivalent:

```
     100 MAT INPUT A
     100 INPUT A(*)
     100 INPUT A(1),A(2),A(3),A(4)
     100 INPUT (FOR I=1 TO 4, A(I))
```

When reading from the terminal keyboard, the MAT INPUT statement prompts for input with a question mark (?). Respond to the prompt by typing a list of values. Separate values with a comma. Press RETURN to store the values. The MAT INPUT statement prompts for input until it has assigned a value to every array element.

The behavior of the MAT INPUT statement follows the general behavior of the INPUT statement, described in chapter 6.

If A is

```
     0 0
     0 0
```

before the statement

```
     10 MAT INPUT A
```

executes, and the response to the statement is

```
     ?2,4  RETURN
     ? RETURN

     ?8  RETURN
```

then A is:

```
     2 4
     0 8
```

MAT PRINT

The MAT PRINT statement prints one or more arrays to the standard list device or a data file. It prints them element by element, varying the rightmost subscript fastest.

Syntax

For printing to a string variable or the standard list device:

```
                 [{,}      ]   [,]
MAT PRINT array  [{;} array]...[;]
```

For printing to a data file:

```
                                         [{,}       ]   [{,}    ]
MAT PRINT #fnum [,rnum [,wnum ]]; array  [{;} array ]...[{;} END]
```

Parameters

array Structured collection of variables of the same type. The structure is determined when the array is declared. String variables names are suffixed with a "$".

fnum File number of a data file. For more information, see "File Identification," in chapter 6.

rnum Record number. If specified, the statement performs a direct write on the data file specified by *fnum*. For more information on rnum and direct reads, see "File Input and Output," in chapter 6.

wnum Word number. If specified, the statement performs a direct word write on the file specified by fnum. That file must be a BASIC DATA file. For more information on *rnum* and direct word reads, see "File Input and Output," in chapter 6.

{,|;} Determines spacing between elements of preceding *array*, if *array* is a numeric array. If a comma follows *array*, each element is printed at the beginning of a 20-character field. If a semicolon follows *array*, elements are separated by two spaces. Each string array element is printed on a separate line.

END Statement prints EOF after last element of last *array*. File must be ASCII or binary.

The following statements can also print arrays:

```
     DISP, PRINT
     DISP USING, PRINT USING
```

If array A has four elements, the following statements are equivalent:

```
     100 MAT PRINT A
     100 PRINT A(*)
     100 PRINT A(1),A(2),A(3),A(4)
     100 PRINT (FOR I=1 TO 4, A(I))
```

The following shows an example of printing an array with MAT PRINT.

```
>list
 !  mprtexam
     5 OPTION BASE 1
    10 DIM A(2,2)
    20 A(1,1)=0
    21 A(1,2)=0
    22 A(2,1)=0
    23 A(2,2)=0
    30 MAT INPUT A
    40 MAT PRINT A
>run
?1,2,3,4
 1                   2

 3                   4
```

MAT READ

The MAT READ statement assigns values from one or more DATA statements or a data file to one or more arrays. If new dimensions are specified for the arrays, the MAT READ statement redimensions them before assigning values to them. It assigns values element by element, varying the rightmost subscript fastest. The MAT READ statement cannot take input from the terminal keyboard.

Syntax

MAT READ [*#fnum* [,*rnum* [,*wnum*]];] *array* [*dims*][, *array* [*dims*]]...

Parameters

fnum

File number of a data file. If this parameter is specified, the MAT READ statement reads from a data file. If it not specified, the MAT READ statement reads from DATA statements. A program line can read from a DATA statement or a file, and a command can only read from a file. For more information on *fnum*, see "File Identification," in chapter 6.

rnum

Record number. If this parameter is specified, the statement performs a direct read on the data file specified by *fnum*. For more information on *rnum* and direct reads, see "File Input and Output," in chapter 6.

wnum

Word number. If this parameter is specified, the statement performs a direct word read on the file specified by *fnum*. That file must be a BASIC DATA file. For more information on *rnum* and direct word reads, see "File Input and Output," in chapter 6.

array

Structured collection of variables of the same type. The structure is determined when the array is declared. String variables names are suffixed with a "$".

dims

Array dimensions used in syntax specification statements. Its syntax is

(*dim1* [,*dim2* [,*dim3* [,*dim4* [,*dim5* [,*dim6*]]]]])

where *dim1* through *dim6* each have the syntax

[*num_expr1*:]*num_expr2*

and *num_expr1* and *num_expr2* are the lower and upper bounds (respectively) of the dimension. If *num_expr1* is not specified, it is the default lower bound.

Examples

The following examples show the MAT READ statement. Each reads a group of arrays into array variables. In lines 100 and 120, the entire arrays are read, and in lines 110 and 130 selected elements are read.

```
     100 MAT READ #1; A,B,C$
     110 MAT READ #2; A(1:3),B(0:4,0:6),C$(3,4,5,6)
     120 MAT READ #1,7; D$
     130 MAT READ #4,6,2; Q,P(9,9),R
```

If array A has four elements, the following statements are equivalent:

```
     100 MAT READ #1; A
     100 READ #1; A(*)
     100 READ #1; A(1),A(2),A(3),A(4)
     100 READ #1; (FOR I=1 TO 4, A(I))
```

NEXT

The NEXT statement is part of the FOR construct. Refer to the FOR statement for more information.

OFF DBERROR

The OFF DBERROR statement deactivates any ON DBERROR statement that affects the program unit containing the OFF DBERROR statement.

Syntax

OFF DBERROR

If the program unit containing an OFF DBERROR statement calls another program unit, then the ON DBERROR statement is inactivated in the called program unit also.

If the OFF DBERROR statement is in a called subunit, the ON DBERROR statement is reactivated when control returns to the calling program unit.

OFF END

The OFF END statement disables the ON END statement.

Syntax

OFF END *#fnum*

Parameters

fnum	The file number that the OFF END affects. This is the same *fnum* specified in the ON END statement.

It disables the ON END statement that specifies the same *fnum*.

Examples

```
     100 ASSIGN #1 TO "File1"
     110 ASSIGN #2 TO "File2a"
     120 ASSIGN #3 TO "File3"
     130 ON END #1 GOTO 999
     140 ON END #2 GOSUB 200  !ON END statement for file #2
     150 ON END #3 CALL End3
     160 READ #1; A1,B1,C1
     170 READ #2; A2,B2,C2
     180 READ #2; D,E,F
     190 READ #3; A3,B3,C3
     195 STOP
```

```
197 !
200 ASSIGN #2 TO "File2b"
210 OFF END #2             !Disables the ON END statement in line 140
220 RETURN
225 !
230 SUB End3
240   PRINT "Reusing File3"
250   POSITION #3;BEGIN
260 SUBEND
```

OFF ERROR

Execution of the OFF ERROR statement deactivates any ON ERROR statement that affects the program unit containing the OFF ERROR statement.

Syntax

```
OFF ERROR
```

If a program unit executes an OFF ERROR statement and then calls another program unit, any previous ON ERROR statement is inactive in the called program unit also.

If the OFF ERROR statement is in a subunit, the last previous ON ERROR statement is reactivated when control returns to the calling program unit.

The following program segment illustrates OFF ERROR.

Examples

```
100 ON ERROR CALL Error
105 I=J/0                   !Trapped by line 100 ON ERROR
110 CALL Sub1
120 I=J/0                   !Trapped by line 100 ON ERROR
130 END
200 SUB Sub1
210   I=J/0                 !Trapped by line 100 ON ERROR
220   OFF ERROR
225   I=J/0                 !Not trapped
230   CALL Sub2
240   I=J/0                 !Not trapped
300 SUB Sub2
310   I=J/0                 !Not trapped
320   CALL Sub3
400 SUB Sub3
410   I=J/0                 !Not trapped
420   ON ERROR GOTO 430
425   I=J/0                 !Trapped by line 420 ON ERROR
430   PRINT "Error at 425"
440 SUBEND
500 SUB Error
510   PRINT "Error at 105, 120, or 210"
515   I=0
520 SUBEND
```

OFF HALT

The OFF HALT statement deactivates the currently active ON HALT statement.

Syntax

```
OFF HALT
```

If the OFF HALT statement is in a subunit, it deactivates the currently active ON HALT statement only while the subunit is executing. Any active ON HALT statement in the calling program unit is reactivated when control returns to the calling program unit.

OFF KEY

The OFF KEY statement restores the last typing aid key definition for a user-definable key or set of keys. If no typing aid key definitions are active then the default key definitions are restored.

Syntax

OFF KEY [*key_number_list*]

Parameters

key_number_list A list of integers or numeric expressions that evaluate to an integer in the range of [1, 8] separated by commas or semicolons. No more than eight values can be specified for each statement. If the integer is not in the specified range, an error occurs. If values are not specified, typing aid definitions for all keys are restored.

Examples

The first example shows the use of the OFF KEY statement as a command.

```
>OFF KEY 1

100 OFF KEY           ! Restores the typing aid definition of all keys
110 OFF KEY 1         ! Restores the typing aid definition of key 1
120 OFF KEY 1,2,3,8   !Restores the typing aid definition of keys 1,2,3 and 8
```

ON DBERROR

The ON DBERROR statement defines a database error-handling routine. The ON DBERROR statement is unnecessary if each database operation utilizes the STATUS option because the status array returns the error code, and the error does not stop the program.

The ON DBERROR statement is disabled by the OFF DBERROR statement.

Syntax

```
            {GOTO  }
           {{GO TO }        }
ON DBERROR {{GOSUB } line_id }
           {{GO SUB}        }
           {CALL sub_id      }
```

Parameters

line_id Line label or line number.

sub_id Subunit identifier.

Table 4-7 shows the similarities and differences between the three forms of the ON DBERROR statement.

Table 4-7. ON DBERROR Statements

Statement Executed If Run-Time Error Occurs After ON DBERROR Statement Executes	**Line to Which Error-Handling Code Transfers Control When it Ends**	**Scope of ON DBERROR Statement**
GOTO *line_id*	None.	Program unit that contains it.
GOSUB *line_id*	Line following the line where the error occurred.	Program unit that contains it.
CALL *sub_id*	Line following the line where the error occurred.	Program unit that contains it and program unit that this program unit calls (until called program unit executes a local ON DBERROR statement or an OFF DBERROR statement).

Examples

```
     100 ON DBERROR GOTO 500    !Goes to line 500
     110 ON DBERROR GOTO Rtn5   !Goes to the line number in Rtn5
     120 ON DBERROR GOSUB 650   !Goes to the subroutine at line 650
     130 ON DBERROR GOSUB Rtn7  !Goes to the subroutine at the line in Rtn7
     140 ON DBERROR CALL Error  !Goes to the Error subroutine
```

ON END

The ON END statement traps the end-of-file condition for a specified file. That is, if an end-of-file is encountered during an I/O operation, the ON END statement causes an interrupt. When HP Business BASIC/XL responds to the interrupt, it transfers control to the line, subroutine, or subprogram specified by the ON END statement.

The OFF END statement disables the ON END statement. If an end-of-file is encountered during an I/O operation, and no ON END statement is associated with it (or its ON END statement is disabled), an error occurs. An active ON ERROR statement can trap this error. See the ON ERROR statement for more information.

Syntax

```
             {GOTO  }
            {{GO TO }        }
ON END #fnum {{GOSUB } line_id }
            {{GO SUB}        }
            {CALL sub_id      }
```

Parameters

fnum The file number of the file that the ON END statement applies to.

line_id Line label or line number. Control will transfer to this *line_id* when the ON END statement executes.

sub_id Subunit identifier. Control transfers to this subunit when the ON END statement executes.

Examples

The following example uses the ON END statement to trap an end-of-file error. Lines 20-90 set up the file. Line 200 contains the ON END statement. Lines 210-240 read the file, and an end-of-file occurs. The ON END statement prints line 300, and execution continues.

```
          10 DIM A(15),B(15)
          20 CREATE "Test1",FILESIZE=15,RECSIZE=10
          30 ASSIGN #1 TO "Test1"
          40 FOR I=1 TO 3
          50    A(I)=I
          60    B(I)=I*2
          70    PRINT #1;A(I),B(I)
          80 NEXT I
          90 POSITION #1;BEGIN
         200 ON END #1 GOTO 300
         210 FOR I=1 TO 15
         220    READ #1;A1,B1
         230    PRINT A1,B1
         240 NEXT I
         250 END
         300 PRINT " End of data file reached !!  "
         310 STOP
     >run
      1                   2
      2                   4
      3                   6
      End of data file reached !!
```

ON ERROR

The ON ERROR statement defines an error-handling routine to handle all run-time errors that are not trapped by an ON DBERROR or ON END statement in the same program.

Syntax

```
          {GOTO  }
         [{GO TO }        ]
ON ERROR [{GOSUB } line_id ]
         [{GO SUB}        ]
         [CALL sub-id      ]
```

Parameters

line_id Line label or line number. Control will transfer to this *line_id* when the ON ERROR statement executes.

sub_id Subunit identifier. Control will transfer to this subunit when the ON ERROR statement executes.

Table 4-8 shows the similarities and differences between the three forms of the ON ERROR statement.

Table 4-8. ON ERROR STATEMENTS

Statement Selected	**Line to Which Control is Transferred following ON ERROR processing**	**Scope of ON ERROR Statement**
GOTO *line_id*	None.	Program unit that contains it.
GOSUB *line_id*	Line following the line where the error occurred.	Program unit that contains it.
CALL *sub_id*	Line following the line where the error occurred.	Program unit that contains it and program unit that it calls, until called unit executes a local ON ERROR statement or an OFF ERROR statement.

HP Business BASIC/XL provides predefined functions that can be used in error recovery routines. They are ERRL, ERRN, ERRM$, and ERRMSHORT$. They are defined in chapter 5.

Examples

```
100 ON ERROR CALL Default
110 READ A,B
120 C=B/A                 !Error can occur here
130 DISP A,B,C
135 END
140 SUB Default
150   C=0
160 SUBEND
```

The next three examples show how the three forms of the ON ERROR statement transfer control when errors occur.

```
100 ON ERROR GOTO 140
110 I=J/0              !Error occurs; go to line 140.
120 PRINT "DONE"       !This statement is never executed.
130 GOTO 999
140 PRINT "ERROR"      !Execute line 999 next.
999 END

100 ON ERROR GOSUB 140
110 I=J/0              !Error occurs; gosub line 140.
120 PRINT "DONE"
130 GOTO 999
140 PRINT "ERROR"
150 RETURN             !Return to line 120.
999 END
100 ON ERROR CALL Error
110 I=J/0              !Error occurs; call to line 140.
120 PRINT "DONE"
130 END
135 SUB Error
140   PRINT "ERROR"
150 SUBEND             !Return to line 120.
```

The next three examples show the scope of the three forms of the ON ERROR

statement.

```
     100 ON ERROR GOTO 115
     105 A=B/0                !Error occurs; go to line 115
     115 PRINT "ERROR"
     116 CALL Sub1
     120 END
     130 SUB Sub1             !ON ERROR at line 100 inactive within Sub1
     140   I=J/0              !Error aborts program
     150 SUBEND

     100 ON ERROR GOSUB 115
     105 A=B/0                !Error occurs; gosub line 115.
     110 CALL Sub1
     115 PRINT "ERROR"
     116 RETURN               !Return to line 110.
     120 END
     130 SUB Sub1             !ON ERROR at line 100 inactive in Sub1
     140   I=J/0              !Error aborts program.
     150 SUBEND

     100 ON ERROR CALL Error
     110 A=B/0                !Error occurs; call Error; return will be to
     115                      !line 115
     120 CALL Sub1
     130 END
     141 SUB Error
     150   PRINT "ERROR"
     160 SUBEND
     170 SUB Sub1             !ON ERROR still active within Sub1
     180   I=J/0              !Error occurs; call Error
     190 SUBEND
```

The next example shows how a local ON ERROR statement overrides an active ON ERROR statement in the calling program unit.

```
     100 ON ERROR CALL Error
     105 P=Q/0                        !Error occurs; call Error
     110 CALL Sub1
     115 R=S/0                        !Error occurs; call Error
     120 CALL Sub2
     125 T=U/0                        !Error occurs; call Error
     130 END
     140 SUB Sub1
     150   A=B/0                      !Error occurs; call Error
     160 SUBEND
     170 SUB Sub2
     175   M=N/0                      !Error occurs; call Error
     180   ON ERROR GOSUB 240         !Overrides line 100
     190   I=J/0                      !Error occurs; GOSUB 210
     200   GOTO 230
     210   PRINT "Error at line 190"
     220   RETURN
     230 SUBEND
     240 SUB Error
     250   PRINT "Error at line 105,115,125,150, or 175"
     260 SUBEND
```

ON GOSUB

The ON GOSUB statement is one of the GOSUB corollaries of the ON GOTO and GOTO OF statements. Control is transferred to the selected line, L, by "GOSUB L" rather than "GOTO L." A RETURN statement returns control to the statement that follows the ON GOSUB statement. Although the ON GOSUB statement can be input as ON GOSUB or ON GO SUB, HP Business BASIC/XL will always list it as ON GOSUB.

Syntax

```
            {GOSUB }
ON num_expr  {GO SUB} line_id  [, line_id ]...[ELSE else_line_id ]
```

Parameters

num_expr A numeric expression that is evaluated and converted to an integer, *n*. The integer *n* must be between one and the number of *line_id* s, or an error occurs if no ELSE clause is present. Control is transferred to the *n* th *line_id*.

line_id Line number or line label of the line to which control is transferred. The line must be in the same program unit as the ON GOSUB statement.

else_line_id Line number or line label of the line to which control is transferred if the value of *num_expr* is not between one and the number of *line_ids* specified.

Examples

```
       1 READ I, J
       2 ON I GOSUB 10,20,30         !Go to subroutine at line 10, 20, or 30
       3 ON J GOSUB 40,50,60 ELSE 99 !Go to subroutine at line 40,50, or 60 or to
                                     !line 99
       4 STOP                        !if J < 1 or J > 3
      10 REM Subroutine for I=1
      11   PRINT "I is one"
      12   RETURN                    !Return to line 3
      20 REM Subroutine for I=2
      21   PRINT "I is two"
      22   RETURN                    !Return to line 3
      30 REM Subroutine for I=3
      31   PRINT "I is three"
      32   RETURN                    !Return to line 3
      40 REM Subroutine for J=1
      41   PRINT "J is one"
      42   RETURN                    !Return to line 4
      50 REM Subroutine for J=2
      51   PRINT "J is two"
      52   RETURN                    !Return to line 4
      60 REM Subroutine for J=3
      61   PRINT "J is three"
      62   RETURN                    !Return to line 4
      90   DATA 3,2
      99 END
```

The GOSUB OF statement works exactly the same as the ON GOSUB statement. The following statements are equivalent:

```
     150 ON I GOSUB 10, 20, 30, Quit
     150 GOSUB I OF 10, 20, 30, Quit
```

ON GOTO

The ON GOTO statement transfers control to one of several lines, depending on the value of a numeric expression. Although the ON GOTO statement can be input as ON GOTO or ON GO TO, HP Business BASIC/XL always lists it as ON GOTO.

Syntax

```
            {GOTO }
ON num_expr  {GO TO} line_id  [, line_id ]...[ELSE else_line_id ]
```

Parameters

num_expr A numeric expression that is evaluated and converted to an integer, *n*. The integer *n* must be between one and the number of *line_id* s, or an error occurs if no ELSE clause is present. Control is transferred to the *n* th *line_id*.

line_id A line number or line label of a line to which control can be transferred. The line specified must be in the same program unit as the ON GOTO statement.

else_line_id The ELSE clause allows the specification of a line to which control is transferred if the value of *num_expr* is NOT between one and the number of *line_id* s specified.

Examples

```
      10 I=2
      20 ON I GOTO 30,40,50
      30 PRINT "I IS 1"
      35 GOTO 99
      40 PRINT "I IS 2"      !Line 20 transfers control here
      45 GOTO 99
      50 PRINT "I IS 3"
      99 END
```

The GOTO OF statement works exactly the same as the ON GOTO statement. The following statements are equivalent:

```
     150 ON I GOTO 10,200,ReInit,Quit
     150 GOTO I OF 10,200,ReInit,Quit
```

ON HALT

The ON HALT statement specifies an action that the program executes when it traps pressing of the halt key.

If an ON HALT statement is active when the halt key is pressed, the ON HALT Statement traps the halt key, and the ON HALT directive (GOTO, GOSUB or CALL) is executed. The program is not suspended as it is when no ON HALT statement is present in the program.

Syntax

```
         {GOTO  }
        [{GO TO }         ]
ON HALT [{GOSUB } line_id ]
        [{GO SUB}         ]
        [CALL sub_id      ]
```

Parameters

line_id Line label or line number. Control will transfer to this *line_id* when the ON HALT statement executes.

sub_id Subunit identifier. Control will transfer to this subunit when the ON HALT statement executes.

Table 4-9 shows the similarities and differences between the three forms of the ON HALT statement.

Table 4-9. ON HALT Statements

Statement Selected	Line to Which Control is Transferred following ON HALT processing	Scope of ON HALT Statement
GOTO *line_id*	None.	Program unit that contains it.
GOSUB *line_id*	Line following the line that was executing when the halt key was pressed.	Program unit that contains it.
CALL *sub_id*	Line following the line that was executing when the halt key was pressed.	Program unit that contains it and program unit that it calls, until called unit executes a local ON HALT statement or an OFF HALT statement.

If you use the CALL option, it cannot have parameters. To achieve the effect of a CALL with parameters, use the GOSUB form and put the desired CALL statement at the GOSUB destination.

Examples

```
     10 ON HALT GOSUB 20
     20 CALL Sub3 (A,B)     !Control goes here if the halt key is pressed.
```

An ON HALT statement is deactivated by execution of another ON HALT statement or by an OFF HALT statement.

ON KEY

The ON KEY statement defines a branch that is to be executed when a specific branch-during-input key is pressed during execution of an HP Business BASIC/XL input or READ FORM statement.

Syntax

```
                        {CALL subprogram }
ON KEY key_number_list  {GOTO line_id    }
                        {GOSUB line_id   }

[{;}                    ]
[{,} LABEL [=] key_label ]

[{;} {PRI     }                   ]
[{,} {PRIORITY} [=] priority_level ]
```

Parameters

key_number_list A list of integers selected from the set of [1..8] or numeric expressions that evaluate to integers in the range of [1..8] separated by commas or semicolons. This set indicates which branch-during-input key is to be trapped. If the integer is not in the specified range, an error occurs. No more than eight values can be specified for each statement.

subprogram A valid subprogram name.

line_id A line number or line label.

key_label A quoted string of characters used to fill in the label field of the user-definable key. The string that you use is specific for your terminal. If the label is <= fifteen characters, it is centered in the key label. If the key label is missing, "f1" through "f8" are used.

priority_level A *num_expr* between 1 and 15, inclusive, used to determine the order in which multiple branches specified by interrupts and branch-during-input statements are handled. If this option is not selected, the branch is placed on the interrupt queue with a priority of 1.

Examples

```
          100 ON KEY 1 GOTO 120
          110 ON KEY 1 GOTO Help_label
          120 ON KEY 3 CALL Help
          130 ON KEY 1 CALL Help_routine,LABEL="  HELP  "
          140 ON KEY 1,2 GOSUB 10,LABEL=Label$
          150 ON KEY 4,5 CALL Assist,PRI=4
          160 ON KEY 2 CALL Help_entry,LABEL="  HELP    ENTRY",PRI=10
          170 ON KEY 7,8 CALL Exit_routine,LABEL="  exit",PRIORITY=15
```

The following example is designed to illustrate the behavior of the ON KEY statement with regard to labels and priorities.

```
      10  ON KEY 1 CALL Suba; LABEL = "Main"
      20  PRESS KEY 1
      30  PRINT "Done"
      40  SUB Suba
      50  ON KEY 1 GOTO Myline;PRIORITY = 4;LABEL = "Suba"
      60  PRINT "In Suba"
      70  PRESS KEY 1
      80  SUBEXIT
      90  Myline: PRINT "Myline"
     100  SUBEND
```

Running this program produces the following output:

```
     In Suba
     Myline
     Done
```

In line 10, KEY 1 is defined with a call action. Main is the label of function key 1.

In line 20, KEY 1 is pressed. This causes the Suba routine in lines 40 through 100 to be executed. On line 50 KEY 1 is defined with a GOTO action. Label "Suba" appears on function key 1. Next, the PRINT statement in line 60 is executed. Following that, in line 70, KEY 1 is pressed. Since the priority of KEY 1 is defined to be 4, the action associated with the ON KEY statement is taken. Remember that KEY 1 was given a priority of 1 on line 10 and a call action was taken. The call action remains active unless KEY 1 is redefined with a higher priority in the subunit. Since this is the case in the above example, line 90 is executed. Line 100 causes a branch to line 30. The PRINT statement in line 30 is executed and the program ends.

OPEN FORM

This statement opens a form and displays it on the terminal. There are several options provided to control or preserve information already on the screen.

If there is no form currently active and a form file is specified in *form_file_name*, the file type of the form file is examined to determine whether the file to be activated is an VPLUS form file or a JOINFORM File. If there is no form file specified as part of the form name, the most recently opened form file is used. An error occurs if no form file has been opened and the *form_file_name* is omitted.

Any function key that is defined using HP Business BASIC/XL's ON KEY statement takes precedence over the definition of the key defined by the VPLUS form. Therefore, you can define user-defined branch-during-input keys and the associated key labels that are to be active during the form processing prior to opening the form.

Syntax

```
                    [{;}     ]
                    [{,} HOME]
OPEN FORM form_name  [OVERLAY ]
                    [FREEZE  ]
                    [APPEND  ]
```

If the comma is used as the separator above, HP Business BASIC/XL will accept it, but will replace it with a semicolon.

Parameters

form_name The *form_name* is a string expression with the following format:

form_member_name [:*form_file_name*]

The *form_member_name* is the name of the form that you are opening. The *form_file_name* is the name of the file that contains the form.

HOME
OVERLAY
FREEZE
APPEND The HOME, OVERLAY, FREEZE, and APPEND options are ignored if the form to be opened is a JOINFORM.

Only one of these options can be used in an OPEN statement. When the HOME keyword is specified, any existing form is cleared and the form being opened is placed at the top of display memory. When the APPEND keyword is specified, the form being opened is positioned following the currently active form; if none is active, HOME is substituted. OVERLAY is the keyword to use when you wish to replace the currently active form without otherwise disturbing display memory. The FREEZE keyword causes *memory locking* of any currently active form followed by an APPEND. If none of these options is specified, HOME is the default.

Examples

The following examples show the OPEN FORM statement.

```
100 OPEN FORM "FORM1:ABC"        !Opens FORM1 in ABC
110 OPEN FORM A$;OVERLAY         !Opens the form in A$
120 OPEN FORM Form$+":FORMFILE"  !Opens the form in Form$
130 OPEN FORM "XYZ";FREEZE       !Opens form XYZ
140 OPEN FORM "XYZ";HOME         !Opens form XYZ
```

OPTION

The OPTION and GLOBAL OPTION statements can change the program unit characteristics shown in Table 4-10. The value of each program unit characteristic is initially set to the value in the HP Business BASIC/XL configuration file, HPBBCNFG.PUB.SYS supplied with HP Business BASIC/XL. The current values of each characteristic are displayed by the INFO command.

Table 4-10. Changeable Program Unit Characteristics

Program Unit Characteristic	Option (Default First)	Effect
Default numeric type (type assigned to implicitly declared numeric variables).	REAL DECIMAL	Implicitly declared numeric variables are type REAL. Implicitly declared numeric variables are type DECIMAL.
Initialization of numeric variables to zero.	INIT NOINIT	Numeric variables are initialized to zero. Numeric variables are not initialized to zero.
Implicit variable declaration. *	NODECLARE DECLARE	Implicit variable declaration is legal. Implicit variable declaration is illegal.
Default of lower bound of arrays.	BASE 0 BASE 1	Default lower bound is zero. Default lower bound is one.
Trace statement output control.	TRACE NOTRACE	Enables trace statements. Disables trace statements.

Table 4-10 Note

* If an implicit variable declaration is illegal, using a variable that is not explicitly declared causes an error. If the program is interpreted, the error occurs at run time; if the program is compiled, it occurs at compile time.

Table 4-11. Global Program Subunit Options

Program Unit Characteristics	Option (Default First)	Effect
Declare current program status in a multi-program application. Control creation and deletion of COM areas.	[NONEWCOM] MAIN [NEWCOM]	Current program starts multi-program application. NEWCOM allows new commons when program loaded. NONEWCOM prevents new commons.
**The MAIN and SUBPROGRAM global options control both program execution and the creation and deletion of common areas.	SUBPROGRAM [NONEWCOM] [NEWCOM]	Identifies current program as a module of multi-program application, but not the initial main. This program can not be RUN; only a programmatic GET can run this program. NEWCOM allows creation and deletion of COM blocks when the GET occurs. NONEWCOM prevents this.

```
|                     |                    |                                       |
----------------------------------------------------------------------------------
```

Table 4-11 Note

** The default value if neither is specified is MAIN NONEWCOM. The value of this option is not set in the configuration utility nor is the value displayed in the INFO command's display.

Syntax

[GLOBAL] OPTION *option_list*

Parameters

GLOBAL — Allowed only if the statement is in the main program. If GLOBAL appears, the statement is a GLOBAL OPTION statement; if GLOBAL is omitted, the statement is an OPTION statement. A GLOBAL OPTION statement affects every program unit in the program. An OPTION statement affects only the program unit that contains it.

An OPTION statement overrides a GLOBAL OPTION statement, but only while the program unit that contains it is running.

option-list — A list of one to five unique options separated by commas. Each option can be one of each of the following pairs:

```
  DECIMAL or REAL
  INIT or NOINIT
  DECLARE or NODECLARE
  BASE 0 or BASE 1
  TRACE or NOTRACE
```

With GLOBAL specified, one of the following additional options can also be specified:

```
  MAIN
  SUBPROGRAM
  MAIN NONEWCOM
  MAIN NEWCOM
  SUBPROGRAM NONEWCOM
  SUBPROGRAM NEWCOM
```

The term "OPTION *x* statement" where *x* is DECIMAL, REAL, INIT, NOINIT, DECLARE, NODECLARE, TRACE, NOTRACE MAIN, or SUBPROGRAM means an OPTION or GLOBAL OPTION statement that contains the OPTION *x*. For example, the line

```
     120 GLOBAL OPTION BASE 1, REAL, NOINIT, NODECLARE
```

can be called a GLOBAL OPTION statement, an OPTION BASE statement, an OPTION REAL statement, an OPTION NOINIT statement, or an OPTION NODECLARE statement.

If a program unit contains conflicting OPTION *x* statements, then the option active in the subunit is determined by the OPTION or GLOBAL OPTION statement with the highest line number. An OPTION statement can reset the same option that a GLOBAL OPTION statement has set in the main program.

The following are the default options for a program unit without an OPTION statement:

```
  REAL
  INIT
  NODECLARE
  BASE 0
  TRACE
  MAIN NONEWCOM
```

OPTION and GLOBAL OPTION statements are processed immediately before the program units containing them are run. Neither statement can be used as a command.

The MAIN and SUBPROGRAM global options are used chiefly for compiling multi-program applications. A program that uses the SUBPROGRAM option can only be run by execution of a GET program line from within an executing program. Trying to RUN a program in the interpreter that has a GLOBAL OPTION SUBPROGRAM statement results in an error. Programs that contain the GLOBAL OPTION MAIN can be executed by using the RUN command in the interpreter as well as by executing a GET statement for that program in an executing program.

A suboption of the MAIN/SUBPROGRAM option is NONEWCOM or NEWCOM. The suboption relevant at the execution of the GET statement is that in the called program unit, not that suboption present in the caller. The NONEWCOM suboption prevents the deletion and addition of COM areas regardless of whether the called program uses the COM area. COM areas named in both the calling and called programs are checked to ensure that the declarations in each match. NONEWCOM is the active suboption if neither suboption is specified. The NONEWCOM suboption causes every programmatic GET to compare COM area names. Any COM areas not named in both programs are deleted and any COM areas named only in the new program are created.

Examples

The comments in the following example explain the extent that local OPTION statements override the GLOBAL OPTION statement in the main program.

```
     100 GLOBAL OPTION DECIMAL, INIT, DECLARE, BASE 0
          .
          .
     125 CALL Sub1
          .
          .
     150 CALL Sub2
          .
          .
     175 CALL Sub3
          .
          .
     200 SUB Sub1
     210 OPTION REAL, NOINIT   !Options: REAL,NOINIT,DECLARE,BASE 0
          .
          .
     250 SUBEND
     300 SUB Sub2
     310 OPTION NODECLARE
     320 OPTION BASE 1         !Options: DECIMAL, INIT, NODECLARE, BASE 1
          .
          .
     350 SUBEND
     400 SUB Sub3
     410 OPTION DECIMAL        !Options: Same as global options
          .
          .
     450 SUBEND
     999 END
```

Each of the following three programs declares the variable A implicitly. In the first and third programs, A is real. In the second program, A is decimal.

```
     10 OPTION REAL          10 OPTION DECIMAL        10 REM No option specified
     20 A = PI               20 A = PI                20 A = PI
     99 END                  99 END                   99 END
```

In the first and second programs below, the variables are initialized to zero when the program is run; in the third, they are not.

```
     10 OPTION INIT          10 REM No option specified   10 OPTION NOINIT
     20 INTEGER X,Y,Z        20 REAL A,B,C,D,E            20 DECIMAL P,Q
     99 END                  99 END                       99 END
```

In the first and second programs below, implicit variable declaration is legal. Numeric variable X and string variable A$ are implicitly declared, and no error occurs. In the third program, implicit variable declaration is illegal. A run-time error occurs at line 20.

```
     10 OPTION NODECLARE      10 REM No option specified  10 OPTION DECLARE
     20 X = 4535              20 X = 4535                 20 X = 4535
     30 A$ = "Hi"             30 A$ = "Hi"                30 A$ = "Hi"
     99 END                   99 END                      99 END
```

Each of the following programs declares numeric array A and string array B$. In the first and second programs, the arrays have lower bounds zero. Therefore, A has six elements and B$ has 15 (three rows and five columns). In the third program, the arrays have lower bound one. Array A has five elements and B$ has eight (two rows and four columns). For more information, see "Array Variables" in chapter 3.

```
     10 OPTION BASE 0         10 REM No option specified   10 OPTION BASE 1
     20 DIM A(5)              20 DIM A(5)                  20 DIM A(5)
     30 DIM B$(2,4)           30 DIM B$(2,4)               30 DIM B$(2,4)
     99 END                   99 END                       99 END
```

An example of the MAIN/SUBPROGRAM global option requires the definition of a program file in addition to the program currently in the interpreter. The example demonstrates the allocation of new and deallocation of old common areas.

```
     >LIST
          10 ! current program in the interpreter
          20 GLOBAL OPTION MAIN NONEWCOM
          30 COM /Com1/ Main1,Main2
          40 COM /Com2/Main3,Main4
          50 Main1=1;Main2=2;Main3=3;Main4=4

          .

          .

          .
     >RUN
      1  2  0  0

     >LIST
      ! SUBFILE
         10 ! program in SUBFILE
         20 GLOBAL OPTION SUBPROGRAM NEWCOM
         30 COM /Com1/ Sub1,Sub2
         40 COM /Com3/ Sub3,Sub4
         50 PRINT Sub1;Sub2;Sub3;Sub4
```

The NONEWCOM/NEWCOM option in the called program in SUBFILE states that new common areas can be allocated. This allows allocation of Com3. Since Com2 is not used in the program in SUBFILE, Com2 is deallocated. As can be seen by program execution, the values assigned to the Com1 common area variables in the callee are those referenced by the variables in Com1 in the called program.

PACK

The PACK statement assigns the values of data from one or more variables to one scalar string variable, in the order specified by the names of the variables in the referenced PACKFMT statement.

Syntax

PACK USING *line_id*; *str_var*

Parameters

line_id Identifies the program line of the appropriate PACKFMT statement that specifies the variables to be packed and the format in which to pack them within *str_var*.

str_var Scalar string variable into which variables are to be packed.

When packing a string variable into *str_var*, if the length of the value of the string variable in the PACKFMT is less than that string variable's maximum length, the PACK statement blank-fills the unused portion and packs the entire string variable.

When packing a substring into *str_var*, if the length of the value of the substring in the PACKFMT is less than the length of the substring, the PACK statement blank-fills the unused portion and packs the substring length.

Examples

See the UNPACK Statement.

PACKFMT

The PACKFMT statement is a list of variables that are to be packed or unpacked by the PACK and UNPACK statements. You can also specify the number of characters to be skipped between data values.

Syntax

PACKFMT [REALV] *pack_item* [, *pack_item*]...

Parameters

REALV The default real data type in a native mode program is an IEEE floating point real format. Therefore, you must specify "REALV" in the PACKFMT statement if the MPE V real data format is desired. The keyword, "REALV" will be ignored on MPE V.

pack_item One of the following:

* Scalar numeric variable.

* Scalar string variable.

* Substring.

* String or numeric array.

* Space specifier: SKIP *number*.

* String or numeric literal.

* Numeric literal type converted with one of the following built in functions:
  SINTEGER

```
INTEGER
SREAL
REAL
SDECIMAL
DECIMAL
```

where *number* is a positive short integer numeric constant that specifies the number of characters skipped. The skip feature is used to bypass unneeded data in a data set. For the PACK USING and UNPACK USING statements, that number of characters (bytes) are skipped in the specified *str_var*. For the DBGET USING, DBPUT USING and DBUPDATE USING statements that number of characters is skipped in the implicit *str_var*. In both cases, use of this option can save time when accessing a subset of the variables in a data set.

Examples

The following example shows the PACKFMT statement. It declares the variables, and then specifies three PACKFMT statements.

```
     100 INTEGER Number,Times(4)
     110 DIM String$[10],A$[10]
     120 PACKFMT Number,String$,A$[6],Times(*)
     130 PACKFMT Times(*),SKIP 2,String$,SKIP 5,Number,SKIP 1,A$[1;5]
     140 PACKFMT 2, INTEGER(7.2),"wow"
```

PAGE HEADER

The PAGE HEADER statement prints at the top of every page of a report unless suppressed. The first page header follows the report header. This section is activated by any automatic page break or the TRIGGER PAGE BREAK statement. The PAGE HEADER section is optional.

Syntax

```
            [                [LINES]]
PAGE HEADER [WITH num_lines  [LINE ]]

[USING image  [; output_list ]]
```

Parameters

num_lines The maximum number of lines expected to be needed by the section statement. This number reflects ALL output done by the section.

image An image string or a line reference to an IMAGE line.

output_list A list of output items, identical to the list described in the PRINT USING statement.

Examples

The following examples show the use of the PAGE HEADER statement.

```
     100 PAGE HEADER
     110 PAGE HEADER WITH 4 LINES
     120 PAGE HEADER WITH 2 LINES USING Hdr;Co_name$,Pg
     130 Hdr: IMAGE 30X,25A,4D,/           !Image statement for line 120.
```

The WITH clause of the PAGE HEADER section is evaluated only once, when BEGIN REPORT executes. This number of lines specified is used throughout the rest of the report, and helps define the "effective" page size. The page header section does not have to print as many lines as are reserved. If it does not, other lines may be printed in the rest of the space.

The USING clause is executed each time a page header is printed.

The PAGE HEADER statement generates an error if a report is not active.

If a report section is active; that is, executing, and encounters this statement, then that report section is ended.

The PAGE HEADER statement executes when an automatic page break condition occurs, or when the TRIGGER PAGE BREAK statement is executed. Under these circumstances, the PAGE TRAILER prints, followed by the PAGE HEADER.

After the page eject, the Report Writer pauses if the PAUSE EVERY statement applies. The page function values; such as, page number, number of pages output, NUMLINE function, and number of lines left on the page are then reset. The PAGE HEADER prints after this. Thus, the PAGE HEADER lines do count as part of the NUMLINE value.

The PAGE HEADER does not print if the SUPPRESS PAGE HEADER ON statement has been executed. The TRIGGER PAGE BREAK statement can suppress the page header with its SUPPRESS option. Refer to TRIGGER PAGE BREAK for more information. If the page header is suppressed, none of the statements in the PAGE HEADER section are executed.

If the REPORT HEADER section executes a TRIGGER PAGE BREAK, so that a "cover" page is printed, the PAGE HEADER is printed only at the top of the new page. The PAGE HEADER is not printed twice as might be expected.

PAGE LENGTH

The PAGE LENGTH statement is used to set the size of a report page. You can specify the page length and the top and bottom margin size with this statement. There can be only one PAGE LENGTH statement in a report description.

Syntax

PAGE LENGTH *length* [, *blank_top* [, *blank_bottom*]]

Parameters

length Expression is a numeric expression in the range [0, 32767]. A value of zero indicates an infinite page length. This prevents error 260, "Insufficient space for printer output within the current page". The default value is 60.

blank_top A numeric expression indicating how many blank lines should be in the top margin. These lines are printed before the page header, and are not suppressed by the SUPPRESS HEADER ON statement. The value must be between zero and the length of the page. The default value is zero.

blank_bottom A numeric expression indicating how many blank lines are in the bottom margin. These lines are printed after the page trailer and are not subject to page trailer suppression. The value must be between zero and the length of the page. The default value is zero.

NOTE After the report definition is scanned by BEGIN REPORT, a final check is made on the page size. The following condition must hold or an error occurs:

```
Page_length - Blank_top - Blank_bottom -
Page_header_size - Page_Trailer_size >=3
```

Examples

The following examples show the use of the PAGE LENGTH statement.

```
     100 PAGE LENGTH 60,0,0
     100 PAGE LENGTH 66,2,2   ! HP 250/260 default
```

The PAGE LENGTH statement is evaluated only during BEGIN REPORT. The page size cannot change during the report. The statement is busy only while being evaluated.

PAGE TRAILER

The PAGE TRAILER statement in the PAGE TRAILER section is a Report Writer statement used to print lines at the bottom of every page of a report. This statement is executed when an automatic page break occurs or when the TRIGGER PAGE BREAK statement executes. A page trailer is printed after the REPORT TRAILER section and when a report ends. The page trailer does not execute when the report terminates abnormally; for example, when a STOP, or STOP REPORT executes in the program.

Syntax

```
             [                [LINES]]
PAGE TRAILER [WITH num_lines  [LINE ]]

[USING image  [; output_list ]]
```

Parameters

num_lines The maximum number of lines expected to be needed by the section statement. This number reflects ALL output done by the section.

image An image string, or a line reference to an IMAGE line.

output_list A list of output items, identical to PRINT USING.

Examples

The following examples show the use of the PAGE TRAILER statement.

```
     100 PAGE TRAILER
     100 PAGE TRAILER WITH N LINES
```

The WITH clause is evaluated only when BEGIN REPORT executes. This causes the indicated number of lines to be reserved for all page trailers. If the PAGE TRAILER section does not print on all the reserved lines, the remaining lines are printed as blank lines. The Report Writer cannot write extra lines in the page trailer.

The USING clause is evaluated each time a PAGE TRAILER is printed.

The PAGE TRAILER statement generates an error if no report is active.

If a report section is active; that is, executing, and encounters this statement, then that report section is ended.

The PAGE TRAILER statement and section executes when an automatic page break condition occurs, or when the TRIGGER PAGE BREAK statement is executed. In these circumstances, the PAGE TRAILER prints, followed by PAGE HEADER.

An error occurs if the program attempts to write a line in the page trailer area and the page trailer is not suppressed.

In order to perform a page break, the PAGE TRAILER section first prints enough blank lines to position the page trailer properly on the page. Then the PAGE TRAILER statement executes its USING clause, if present. The PAGE TRAILER section executes next, terminating when another REPORT WRITER section statement is encountered. Blank lines are then printed for the remaining lines reserved by the PAGE TRAILER and for the bottom margin.

The page function values; that is, number of lines printed on a page, number of lines left on a page, and number of lines output are then updated, followed by execution of a PAGE HEADER.

The PAGE TRAILER does not print if the SUPPRESS PAGE TRAILER ON statement has been executed. The TRIGGER PAGE BREAK statement can suppress the page trailer with its SUPPRESS option. Refer to TRIGGER PAGE BREAK for more information. If the page trailer is suppressed, none of the statements in the PAGE TRAILER section are executed.

PAUSE

The PAUSE statement suspends program execution. While the program is suspended, you can display and modify values of individual variables, modify program lines, and execute commands.

Syntax

PAUSE [*str_expr*]

Parameters

str_expr String expression that the PAUSE statement displays before suspending program execution.

A suspended program resumes execution when the CONTINUE command is executed.

PAUSE cannot be a command. The following are the equivalent to a PAUSE command:

- Control Y (if no ON HALT statement is active).
- Control Y twice in rapid succession (even if an ON HALT statement is active).

Examples

```
>LIST
 ! mat
    10 OPTION BASE 1
    20 DIM Matrix_read(3,3),Matrix_inverse(3,3)
    30 ASSIGN #1 TO "matrix"
    40 MAT READ #1; Matrix_read
    50 PAUSE
    60 MAT Matrix_inverse=INV(Matrix_read)
    70 PAUSE
>RUN
>MAT PRINT Matrix_read
1                     0                    3

1                     5                    2

6                     1                    1

>MAT PRINT Matrix_inverse
0                     0                    0

0                     0                    0

0                     0                    0
```

```
     >CONT
     >MAT PRINT Matrix_ inverse

     -.035714285714857      -.0357142857142857   .1785714285714286

     -.130952380952381       .2023809523809524  -.0119047619047619

      .3452380952380952      .0119047619047619  -.0595238095238095

     >80 CREATE "inverse"
     >90 ASSIGN #2 TO "inverse"
     >100 MAT PRINT #2;Matrix inverse
     >110 PRINT "Done with program"
     >120 END
     >CONT
     Done with program
     >
```

In the above program, the program is paused at line 50, and the first two MAT PRINT statements are executed. The program is then continued, and pauses again at line 70. At that time, the third MAT PRINT is executed, and lines 80 through 120 are added to the program. The program is then continued to completion. After this last CONT command, the new lines (80-120) are executed.

PAUSE EVERY

The PAUSE EVERY statement is a Report Writer statement that allows you to pause at the end of a report page. This statement is useful for looking at reports on the terminal as well as directing printers to stop for paper replacement at specified times.

Only one PAUSE EVERY statement can occur in a report description.

Syntax

```
      {AFTER EVERY}            [PAGE ]
PAUSE {AFTER      } num_pages  [PAGES]
      {EVERY      }
```

Parameters

num_pages A numeric expression indicating how often the Report Writer should pause. Output will be suspended every page that is a multiple of *num_pages*. The value of the expression must be a non-negative integer. A value of zero causes the statement to be ignored.

Examples

The following examples show the use of the PAUSE EVERY statement.

```
     100 PAUSE EVERY 1 PAGES
     100 PAUSE AFTER EVERY Pause_every PAGES
```

This statement is evaluated only by BEGIN REPORT. It is busy only during its evaluation. If the expression is zero, the statement is ignored and no pauses take place.

The PAUSE EVERY statement is active when report output occurs on the terminal. Reports redirected to non-terminal devices do not suspend output. The SUPPRESS PRINT FOR statement prevents the pause from taking place while output is suppressed. However, the pages are counted while output is suppressed, so the pause takes place on the first page that is a multiple of *num_pages* that gets printed.

When the report pauses, no prompt is given. This prevents extraneous characters from appearing on a printed report. The report writer waits until a carriage return is pressed before continuing. Any characters typed are not echoed. Essentially, the report writer executes the ACCEPT statement to accomplish the pause.

POSITION

The POSITION statement positions the record pointer of a specified file at a specified record. The RESET option can reset the file to an empty file.

Syntax

```
                {rnum  }
                {BEGIN}
POSITION #fnum; {END  }
                {RESET}
```

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer.

rnum A numeric expression. Positions record pointer at the record specified by *rnum*.

BEGIN Positions record pointer at first record in the file.

END Positions record pointer at the EOF mark, beyond the last record in the file.

RESET Positions record pointer at first record in the file and immediately writes an EOF marker. All previous contents of the file are lost following execution of the POSITION statement with this option.

The POSITION statement is used to position the record pointer before a sequential read or write to a file if the pointer is not already in the desired position. The POSITION statement is unnecessary before a direct read or write, because a direct read or write statement specifies a record.

Examples

The following examples show the use of the POSITION statement.

```
     10 POSITION #1; 10      ! Record pointer is at record 10.
     20 POSITION #2; Nextrec ! Record pointer is at record indicated in Nextrec.
     30 POSITION #3; BEGIN   ! Record pointer is at the first record.
     40 POSITION #4; END     ! Record pointer is at the EOF mark.
     50 POSITION #5; RESET   ! Deletes the contents of the file. #5
```

PREDICATE

The PREDICATE statement aids in locking database items. Without this statement, the PACK statement must be used to build a predicate string for the DBLOCK statement. The TurboIMAGE/3000 database requires a precise format for this string. The PREDICATE statement builds the string in the correct format and requires only the relevant information. An entire dataset, items within a dataset, or even a subset of an item can be locked using the PREDICATE statement. Note that more than one lock specification may be given at once. The string resulting from the PREDICATE statement is used in the DESCRIPTOR clause of the DBLOCK statement to lock the database.

Syntax

```
PREDICATE whole_str  FROM dataset
[               [{>=}     ]                  ]
[WITH item_name  [{<=} expr ]                  ]
[               [{= }     ]                  ]
[[{,}         [              [{>=}     ]]]   ]
[[{;} dataset  [WITH item_name [{<=} expr ]]]...]
[[            [              [{= }     ]]]   ]
```

Parameters

whole_str A string variable or string array element that is filled by the PREDICATE statement with the locking information required by TurboIMAGE. The string can then be used in the DBLOCK statement to perform the locking.

dataset The dataset name or number to be locked. If the WITH clause is not given the entire dataset is locked. Otherwise, items within the dataset are locked.

item_name A string expression containing "@" or the name of the database item to lock. The item must be in the dataset requested. If *item_name* is not "@", then the relational operators and the value of the data item to be locked must be included. If they are not, database error -123, "illegal relop in a descriptor" will result.

expr An expression used to limit which items are locked. Only the items from *item_name* that satisfy the relation are locked. If the WITH option is not selected, then the entire *dataset* is locked.

Examples

The following examples show the PREDICATE statement.

```
     100 PREDICATE Pred$ FROM Dset$ WITH Item$="xyz"; Dset2$ WITH Name$ >="TOYS"
     200 PREDICATE Pred$ FROM Dset1$; Dset2$; "parts"
     300 PREDICATE Pred$ FROM Dset1$; Dset2$ WITH Item$ = "skates"
     400 DBLOCK Base$, MODE=5, DESCRIPTOR= Pred$
```

PRESS KEY

The PRESS KEY statement simulates the pressing of a branch-during-input key from within a program.

Syntax

PRESS KEY *key_number*

Parameters

key_number An integer or a numeric expression that evaluates to an integer in the range [1, 8].

Examples

```
     100 PRESS KEY 8 ! Performs the branch associated with the currently
     110             ! defined ON KEY statement for  f8 in the current
     120             !subunit.
```

PRINT

The PRINT statement can output several values. It can use output functions to output control characters. The PRINT statement is similar to the DISP statement. The PRINT statement uses the output device specified by the most recently executed SEND OUTPUT TO statement, and the

DISP statement uses the standard list device. If the most recently executed SEND OUTPUT TO statement specifies the standard list device, or if the program has not executed a SEND OUTPUT TO statement, then the PRINT statement is equivalent to the DISP statement. The PRINT statement can also transfer the value of one or more variables to a data file.

Syntax

```
                         [,]
PRINT [output_item_list ] [;] PRINT #fnum  [, rnum  [, wnum ]];
```

output_item_list

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer.

rnum Record number, a numeric expression. If a file I/O statement specifies *rnum*, it is direct; otherwise, it is sequential.

wnum Word number, a numeric expression. If a file I/O statement specifies *wnum*, it is direct word. This is only allowed with BASIC DATA files.

```
                                    [{[,]...}            ]
output_item_     [,]...output_item  [{;     } output_item ]...
list
```

output_item One of the following:

num_expr

str_expr

, A separator that prints each new item in a separate output field.

; A separator that prints each new item right next to the previous item.

array_name (*) Array reference. See "Array References in Display List" in chapter 6 for more information.

```
                 {PAGE            }
                 {{CTL}           }
output_function   {{LIN}           }
                 {{SPA} (num_expr )}
                 {{TAB}           }
```

See "Output Functions in Display List" in chapter 6 for more information.

FOR_clause (FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *d_list*)

See the section that follows, "FOR Clause in Display List", for more information.

Examples

Below are several examples of the PRINT statement.

```
     200 PRINT
     210 PRINT,
     220 PRINT;
     230 PRINT X,X+Y;A$,LWC$(A$+B$);P(*),Q$(*);PAGE,TAB(10+X);
     240 PRINT Z(*), (FOR I=1 TO 10, Z(I); 2*Z(I); I*Z(I)), D$
     250 PRINT X,B$,C(*),D$(*),
     260 PRINT A,,B
     270 PRINT "THE ANSWER IS: "; Final_total
```

The PRINT statement evaluates the expressions in the display list from left to right, and displays their values on the appropriate output device. It displays numeric values in the current numeric output format (see "Numeric Format Statements").

A PRINT statement without a display list prints a carriage return and a line feed (a CRLF) on the output file or device.

The following examples show the PRINT statement used with data files.

```
     100 PRINT #1; A,B,C
     110 PRINT #2,5; D$,E
     120 PRINT #3,7,4; F(),G$(*,*)
     130 PRINT #4; N,M,(FOR I=1 TO 5, A(I,I), B$(I,I))
```

The PRINT statement writes BASIC DATA, binary, and ASCII files differently; see Table 4-12.

Table 4-12. Effect of File Type on PRINT Statement

	BASIC DATA	**Binary**	**ASCII**
Sequential Write Starts at	Record indicated by record pointer.	Record indicated by record pointer.	Record indicated by record pointer.
And Writes	As many records as needed for output list.	As many records as needed for output list.	As many records as needed for output list.
Direct Write Starts at	Record *rnum*.	Record *rnum*.	Record *rnum*.
And Writes	One record. Error occurs if record cannot accommodate output list.	One record. Error occurs if record cannot accommodate output list.	One record. Error occurs if record cannot accommodate output list.
Direct Word Write Starts at	Word *wnum* of record *rnum*.	Not allowed.	Not allowed.
And Writes.	As many records as needed for output list.	Not allowed.	Not allowed.

NOTE Data that is written to an ASCII file by a PRINT statement cannot be read accurately by a READ statement unless the PRINT statement writes commas between data items on the file. For example, the statement:

```
200 READ #1; A,B,C$,D$
```

can read the data written by the statement:

```
100 PRINT #1; 123, ",", 456, ",abc", ",def"
```

but not by the statement:

```
110 PRINT #1; 123,456,"abc","def"
```

FOR Clause in Display List

The display list of a PRINT statement can contain a FOR clause. The FOR clause is similar to the FOR NEXT construct.

Syntax

(FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *output_item_list*)

Parameters

num_var — A numeric variable assigned the sequence of values: *num_expr1*, *num_expr1* +num_expr3, *num_expr1* +(2* *num_expr3*), etc. The DISP or PRINT statement prints the values of the elements of *d_list* for each value that is less than *num_expr2* if *num_expr3* is positive or greater than *num_expr2* (if *num_expr3* is negative).

num_expr1 — First value assigned to *num_var*.

num_expr2 — Value to which *num_var* is compared before the DISP or PRINT statement prints a value. If *num_expr3* is positive and *num_var* > *num_expr2*, the loop execution is terminated. If *num_expr3* is negative and *num_var* < *num_expr2*, the loop execution is terminated.

num_expr3 — Amount by which *num_var* increases at the end of the loop. The default value is 1 if the step option is not specified.

output_item_list — Same as *d_list* in DISP or PRINT statement syntax.

Examples

```
     PRINT "Values for A are: ",(FOR I=1 TO 4, A(I);),,,"X Value: ",X
```

If each variable is assigned the following values prior to execution of line 20:

```
     A(1) = 10
     A(2) = 20
     A(3) = 30
     A(4) = 40
     X    = 50
```

The output generated by line 20 is:

```
     Values for A are:  10  20  30  40
     X Value:           50
```

Display list FOR clauses can be nested.

```
     20 PRINT (FOR I=1 TO 3, (FOR J=1 TO 2, (FOR K=1 TO 2, B(I,J,K))))
```

For each combination of values of I, J, and K, the following table shows the variable value that the above statement prints.

Value of I	**Value of J**	**Value of K**	**Variable Printed**
1	1	1	B(1,1,1)
1	1	2	B(1,1,2)
1	2	1	B(1,2,1)
1	2	2	B(1,2,2)

2	1	1	B(2,1,1)
2	1	2	B(2,1,2)
2	2	1	B(2,2,1)
2	2	2	B(2,2,2)
3	1	1	B(3,1,1)
3	1	2	B(3,1,2)
3	2	1	B(3,2,1)
3	2	2	B(3,2,2)

PRINT DETAIL IF

The PRINT DETAIL IF statement allows the Report Writer to suppress detail lines without affecting the rest of the report generation. This statement affects only the output associated with the DETAIL LINE statement. All PRINT statements as well as all output generated by report sections are unaffected. Additionally, all breaks and totaling are done normally.

There cannot be more than one PRINT DETAIL IF statement in a report description.

Syntax

[PRINT] DETAIL IF *boolean_expr*

Parameters

boolean_expr A numeric expression used to determine if printing should take place. Output is suppressed if the expression is false (zero).

Examples

```
     100 DETAIL IF Pdi            !Prints if Pdi is true.
     100 PRINT DETAIL IF FNX > 0  !Prints if FNX is > 0.
```

The PRINT DETAIL IF statement becomes busy when BEGIN REPORT executes and remains busy until an END REPORT or a STOP REPORT is executed. The statement is executed by the execution of a DETAIL LINE statement.

When DETAIL LINE executes, the PRINT DETAIL IF expression is evaluated just before detailed output takes place. That is, the statement is executed immediately before the WITH and USING clauses of DETAIL LINE. If the PRINT DETAIL IF expression is false (zero), the WITH and USING clauses are NOT evaluated. All HEADER and TRAILER output still takes place. Normal PRINT statements still produce output as well.
The following programs both suppress the output of DETAIL LINE. However,

controlling the detailed output with PRINT DETAIL IF is more automatic and centralized in one place:

Suppressed by program:

```
         100 REPORT HEADER
     ...
         200 END REPORT DESCRIPTION
     ...
         500 IF Pdi THEN
         510 DETAIL LINE 0 WITH N LINES USING A;X,Y
         515 ELSE
         520 DETAIL LINE 1 WITH N LINES USING A;X,Y
         530 ENDIF
```

Suppressed by Report Writer:

```
         100 REPORT HEADER
         110 PRINT DETAIL IF Pdi
     ...
         200 END REPORT DESCRIPTION
     ...
         500 DETAIL LINE 1 WITH N LINES USING A;X,Y
```

PRINT USING

The PRINT USING statement dictates the format of the values that it prints, by specifying either a format string or an IMAGE statement. The DISP USING statement is similar to the PRINT USING statement, and Table 4-13 compares them.

Table 4-13. DISP USING compared to PRINT USING

Statement	**Prints output to**
DISP USING	Standard list device.
PRINT USING	The device specified by the most recently executed SEND OUTPUT TO statement. If that device is the standard list device, or if the program has not executed a SEND OUTPUT TO statement, PRINT USING is equivalent to DISP USING.

Syntax

PRINT USING *image* [; *output_item* [, *output_item*]...]

Parameters

image Either a string expression or the line identifier of an IMAGE statement. See "Format String" or the IMAGE Statement for more information.

output_item Numeric or string expression.

Examples

```
     110 Image$="D,2D,4A,2X,6A"
     120 IMAGE 4A,AAA,3A
     130 Image1=120
     210 PRINT USING 300; Num, Str$, A+B
     220 PRINT USING Image1; S$(2,6), T$[1;3], S$(1,4)[5,7]
     230 PRINT USING Image$; A, B, C$, D$
     260 PRINT USING "DD2XZZ"; A, B
     300 IMAGE DDD,4A,DD
```

PROTECT

The PROTECT statement assigns a lockword to a file to protect the file against unauthorized copying, renaming, and purging. A COPYFILE, RENAME, or PURGE statement cannot access the file unless it specifies the associated lockword.

Syntax

PROTECT *fname* [, *lock_word*]

Parameters

fname The file name. A string expression or literal.

lock_word A string expression representing a valid file system lockword. If omitted, any existing lockword is removed.

Examples

```
     PROTECT "File1", "Lock1"   !Lock1 is assigned as the lockword for File1.

     PROTECT "File1/Lock1","Lock2"   !Changes the lockword for File1.
```

PURGE

The PURGE statement removes a file's name from the directory and releases its disk space, permanently.

Syntax

```
                          [{,}                    ]
PURGE fname  [, lock_word ] [{;} STATUS[=]num_var ]
```

Parameters

fname The PURGE statement removes *fname* from the directory and releases the disk space that was allocated to that file. The file data are irretrievably lost.

lock_word String expression that evaluates to the lockword for *fname*. It is required if the file has a lockword.

num_var The PURGE statement assigns a zero to *num_var* on successful completion of the file's removal from the system; otherwise, a nonzero value is assigned.

A nonzero value represents the file error code returned by the file subsystem of the MPE XL operating system. The error number can be translated to an MPE XL file system error message by looking up the table of file system error codes in the *MPE XL Intrinsics Reference Manual* under the FCHECK intrinsic.

Examples

```
     10 CREATE "File1", FILESIZE=1320
     20 CREATE "File2", FILESIZE=2950
     30 PROTECT "File1", "123ZINC"      !Lockword added to File1.
     40 PURGE "File1", "123ZINC"        !File1 is purged.
     50 PURGE "File2"                   !File2 is purged.
     99 END
```

RAD

The RAD statement indicates that angular units will be specified in Radians. This is the default.

A Radian is 1/(2*PI) of a circle. This statement is used with trigonometric functions.

Syntax

RAD

Example

```
     10 Radius=10
     20 RAD
     30 Area=PI*Radius**2
     40 PRINT Area
```

RANDOMIZE

The RANDOMIZE statement resets the value of a seed that the RND function uses for random number generation. The seed is set to one of 116 values that are available to it.

Syntax

RANDOMIZE [*n*]

Parameters

n An optional parameter specifying a value for the seed.

Examples

```
     10 RANDOMIZE             !A random seed value.
     20 RANDOMIZE 1.793       !The seed is 1.793.
```

READ

The READ statement assigns data from one or more DATA statements to specified variables. It also assigns the value of one or more data items in a file to one or more variables.

Syntax

```
     {variable      }{ [variable      ]}
READ {read_for_loop }{,[read_for_loop ]} READ #fnum  [, rnum  [, wnum ]]
```

[; *input_list*]

Parameters

variable Variable reference; that is, a variable name, array reference (one element or an entire array), or substring reference (see "Referencing Variables" in chapter 3 for syntax).

read_for_loop A FOR loop within a READ statement used to assign individual datum to variables. See below for syntax and an explanation.

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer.

rnum Record number, a numeric expression. If a file I/O statement specifies *rnum*, it is direct; otherwise, it is sequential *wnum* Word number, a numeric expression. If a file I/O statement specifies *wnum*, it is a direct word. Direct word reads are allowed only with BASIC DATA files.

```
                  [{,}    ]
input_list   item  [{;}item ]...
```

Each item is a numeric or string variable, an array reference, or a FOR clause. An array reference has the syntax

array_name ([*[,*]...])

with one asterisk per dimension or it does not have asterisks. Not using asterisks specifies any number of dimensions. Either format is legal, but the format without asterisks is noncompilable.

A FOR clause has the syntax

(FOR *num_var* =*num_expr3* TO *num_expr4*
[STEP *num_expr5*], *input_list*)

A sequential read must have an *input_list*.

If a direct read does not have an *input_list*, it is the same as the POSITION *#fnum;rnum* statement. That is, it positions the file at the beginning of record *rnum*.

When used with data files, the READ statement assigns one file datum to one input item. It accesses its input items from left to right. It reads BASIC DATA, binary, and ASCII files differently; see Table 4-14.

Table 4-14. Effect of File Type on READ Statement

	BASIC DATA	**Binary**	**ASCII**
Sequential Read Starts at	Datum indicated by datum pointer.	Record indicated by record pointer (possibly within unexhausted record).	Record indicated by record pointer.
And Reads	As many records as needed to satisfy input list.	As many records as needed to satisfy input list.	As many records as needed to satisfy input list.
Direct Read Starts at	Record *rnum*.	Record *rnum*.	Record *rnum*.
And Reads	As many records as needed to satisfy input list.	As many records as needed to satisfy input list.	As many records as needed to satisfy input list.
Direct Word Read Starts at	Word *wnum* of record *rnum*.	Not allowed.	Not allowed.
And Reads	As many records as needed to satisfy input list.	Not allowed.	Not allowed.

When reading from a binary file, HP Business BASIC/XL does not convert data to the types of the variables to which it assigns them. For example, if a program tries to read decimal data that is in a binary file into real variables, the numbers returned are incorrect.

Examples

```
     10 DATA 12,13,14
     20 DATA 15,16,17,18
     30 READ A,B          !A=12, B=13
     40 READ C            !C=14
     99 END
```

After line	**Data pointer is at:**	**In line:**
20	12	10
30	14	10
40	15	20

If possible, a datum from a DATA statement is interpreted as the type of data required by the variable into which it is read. If an underflow occurs, the value zero is assigned to the variable. Before a datum is assigned to a variable, it is converted to the type of the variable, if possible. A numeric variable requires a numeric literal, and a string variable requires a string literal or any unquoted string. Numeric literals are also unquoted string literals and can thus be assigned to a string variable.

```
     10 DATA 1234, "56", "seven", "eight", 12
     20 READ N,A$                           !N=1234, A$="56"
     30 READ B$, C$                         !B$="seven", C$="eight"
     40 READ D$                             !D$="12"
     99 END
```

Specification of a substring of a string variable does not always "use up" the value that is read into it. However, following the READ, the data pointer moves to the next datum anyway. The rules of substring assignment apply to READ.

```
     10 DIM Str$[3], Str_array$(1:5)[6]
     20 DATA Anteater, Bear, Cat, Dog
     30 READ Str$[1:3]                               !Str$="Ant"
     40 READ Str_array$(1)[1,2], Str_array$(2)[1;1]  ! Str_array$(1)="Be",
     99 END                                          ! Str_array$(2)="C"
```

The READ statement assigns values from left to right when multiple variables are specified. Thus, variable subscripts can be assigned just prior to assignment to an array element. For example, in the statement

```
     2450 READ X,Y,A(X,Y)
```

values are assigned to X and Y before the subscripts of the A array are evaluated.

An example of using READ statements with data files:

```
     100 READ #1; A,B,C
     110 READ #2,5; D$,E
     120 READ #3,7,4; F(),G$(*,*)
     130 READ #4; N,M,(FOR I=1 TO 5, A(I,I), B$(I,I))
```

FOR Loops in READ statements

The READ statements in the previous examples have contained only references to individual variables. A READ statement can contain a FOR loop designed to assign values to specific array elements or substrings of a string variable.

Syntax

(FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *input_list*)

Parameters

num_var The numeric loop control variable that assumes the values *num_expr1*, *num_expr1* +*num_expr3*, *num_expr1* +(2**num_expr3*) on successive executions of the loop body.

num_expr1 The initial value that *num_var* is assigned.

num_expr2 Value to which *num_var* is compared before the loop body is executed.

num_expr3 Amount by which *num_var* is incremented or decremented. The default is one.

input_list The list of items to be read. This is the same as for the READ statement without the FOR loop.

A READ statement executes the following steps each FOR loop specified:

1. *num_var* = *num_expr1*

2. If *num_expr3* is positive; go to step 3 else *num_expr3* is negative got to step 4.

3. If *num_var* <= *num_expr2*, then assign data to the *input_list* elements, and go to step 5; otherwise, stop.

4. If *num_var* >= *num_expr2*, then assign data to the *input_list* elements, and go to step 5; otherwise, stop.

5. *num_var* =*num_var* +*num_expr3*.

6. Return to step 3 or 4 if *num_expr3* is *positive* or *negative*, respectively.

Examples

A variable specified within a FOR loop in a READ statement must contain a reference to *num_expr1* as a subscript or substring if the data are not to be repeatedly assigned to the same variable or array element. When

```
     100 READ (FOR I=1 TO 4 STEP 1 A(I))
```

is executed, the index I assumes the values 1, 2, 3, and 4 and assigns the data to the array elements A(1), A(2), A(3), and A(4). The statement

```
     200 READ (FOR I=2 to 6 STEP 2, A$[I;1])
```

assigns the first character in each of the next three double-quoted string literal data items to positions 2, 4 and 6 in A$.

FOR loops within READ statements can be nested; for example, the following statement reads data into a 3-by-5 array.

```
     250 READ (FOR I=1 TO 3, (FOR J=1 TO 5, A(I,J))
```

READ FORM

The READ FORM statement assigns the values entered into the fields of a VPLUS form to HP Business BASIC/XL variables. A time limit for input can be specified by using the TIMEOUT clause.

Syntax

```
READ [FROM] FORM
[           [{,}             ]]
[form_item  [{;} form_item...]]

[{,}                      ]
[{;} TIMEOUT [=] time_expr ]

[{,}                             ]
[{;} NOEDIT [[=] key_number_list ]]
```

Parameters

form_item One of the following:

form_element
for_clause
skip_clause

form_element One of the following:

num_var
str_var $
array_name ([*[,*]...])
str_array_name $([*[,*]...])

The last two formats above have one asterisk per dimension or does not use asterisks. Not using asterisks specifies any number of dimensions. Either format is legal, but the format without asterisks is not compilable. Substrings are also allowed.

for_clause (FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *form_item* [, *form_item*]...)

A *for_clause* is useful for reading array elements. Refer to the INPUT statement for more information.

skip_clause SKIP *skip_expr*

A *skip_clause* is used to skip one or more fields in the form to avoid the necessity of assigning them. The *skip_expr* is a numeric expression that evaluates to the number of fields to skip.

time_expr *Time_expr* is a numeric expression that evaluates to the number of seconds that you have to fill in any input fields on the form. You must depress the ENTER key or a user-defined branch-during-input key before this time or an HP Business BASIC/XL error occurs. Under the latter conditions, no input is assigned to the form variable(s).

key_number_list A list of integers or numeric expressions that evaluate to an integer in the range of [1..8] separated by commas or semicolons. No more than 8 values can be specified for each statement. If the integer is not in the specified range, an error occurs. If you do not specify values, all keys do not have editing completed.

The READ FORM statement is terminated by pressing the ENTER key or a user-defined branch-during-input key. Fields with matching data items are converted and assigned to the corresponding HP Business BASIC/XL variables.

The READ FORM statement is designed to assign the information in all the fields on an entire screen at once. Each field is assigned to a single variable or array element. The first *form_item* is assigned the value of the first field on the form, the second *form_item* is assigned the next value, etc. Each variable specified in a *for_clause* is assigned the value from a single field. Each element of the array specified by the *array_name(*)* notation is also assigned from a single field.

Skip_clause is used; for example, if you wish to only assign the value of the fourteenth field to a variable without reading and converting fields one through thirteen. Simply include the option, SKIP 13.

The following is an example of a READ FORM statement that assigns values from a form with at least 13 fields assuming the A array has five elements and the B$ array has two elements.

```
          READ FORM Surname$, Firstname$, Initials, SKIP 3, &
                (FOR I=1 to 5 STEP 2,A(I)), B$(*), Choice1$, Choice2$
```

The first three fields are read into Surname$, Firstname$ and Initial$. The next three fields are ignored. The *for_clause* reads values into A(1), A(3), and A(5). B$(*) reads values into B$(1) and B$(2). The twelfth and thirteenth fields are read into Choice1$ and Choice2$ respectively. This same statement causes a run-time error if the active form has fewer than thirteen fields.

The TIMEOUT clause requires that you respond within a set amount of time. If input is not complete within this time, an error condition occurs. The built-in RESPONSE function returns a value of two.

If no VPLUS form is active, or a JOINFORM is active, executing a READ FORM statement causes a run-time error. See Appendix F for information on reading values from a JOINFORM form.

Examples

The following examples show the READ FORM statement.

```
     300 READ FORM
     310 READ FORM A;TIMEOUT=100
     320 READ FROM FORM A;TIMEOUT 100;NOEDIT=8
     330 READ FORM A,B;C$;NOEDIT
     340 READ FORM A,SKIP 2;C$
```

REAL

This statement defines a variable as a type REAL. If the SHORT option is used with it, the variable is type SHORT REAL.

Syntax

```
             {num_var } [  {num_var }]
[SHORT] REAL {arrayd  } [, {arrayd  }]...
```

Parameters

num_var Name of scalar numeric variable to be declared.

arrayd Numeric array description. The syntax for the array is described under the DIM statement.

Examples

The following examples show the REAL and SHORT REAL statements.

```
     100 SHORT REAL Fraction
     120 SHORT REAL Reading1, Reading2(36,36)
     130 REAL Distance
     140 REAL Time1(0:35,1:36,3),Time2
```

REDIM

An array can be redimensioned explicitly or implicitly. The REDIM statement explicitly redimensions one or more arrays. Unlike the DIM statement, it is executable.

You can do the following by redimensioning an array:

 * Change the bounds of one or more dimensions.
 * Decrease the number of elements accessible.

Redimensioning cannot do the following:

 * Change the number of dimensions.
 * Change the element values.
 * Change the storage order.
 * Increase the original number of elements (the number of elements assigned to it by the most recent call to the program unit that declared it).

Syntax

REDIM *array dims* [, *array dims*]...

Parameters

array Structured collection of variables of the same type. The structure is determined when the array is declared. String variables names are suffixed with a "$".

dims Array dimensions used in syntax specification statements. Its syntax is

(*dim1* [,*dim2* [,*dim3* [,*dim4* [,*dim5* [,*dim6*]]]]])

where *dim1* through *dim6* each have the syntax

[*num_expr1*:]*num_expr2*

and *num_expr1* and *num_expr2* are the lower and upper bounds (respectively) of the dimension. If *num_expr1* is not specified, it is the default lower bound.

Examples

```
     100 OPTION BASE 0
     105 DIM A(1:4,3)        !A is 4x4, with 16 elements.
     110 DIM B(1,2,1:3)      !B is 2x3x3, with 18 elements.
     120 REDIM A(1:3,0:1)    !A is now 3x2, with 6 elements.
     130 REDIM B(1,1,2)      !B is now 2x2x3, with 12 elements.
     999 END
```

If A and B look like this before redimensioning:

```
     A: 1 2 3 4     B: 1 2 3
        5 6 7 8        4 5 6
        9 0 1 2        7 8 9
        3 4 5 6
                       1 2 3
                       4 5 6
                       7 8 9
```

then they look like this after redimensioning:

```
     A: 1 2         B: 1 2 3
        3 4            4 5 6
        5 6
                       7 8 9
                       1 2 3
```

Arrays can also be explicitly redimensioned by the MAT READ and MAT INPUT statements, or implicitly redimensioned by the MAT = statement.

REM

The REM statement specifies a remark. It is the first keyword on a comment line. HP Business BASIC/XL ignores the rest of that line.

Syntax

REM

Examples

```
     10  REM  The rest of this line is ignored
```

RENAME

The RENAME statement changes the name of a file.

Syntax

RENAME *fname1* TO *fname2* [, *lock_word*]

Parameters

fname1 Old *fname* of file.

fname2 New *fname* of file.

lock_word String expression that evaluates to the lockword for *fname1*. It is required if *fname1* has a lockword. The lockword is not added to *fname2*.

Examples

```
     10 CREATE "File1/secret", FILESIZE=1320    !File1 has a lockword "secret"
     20 CREATE "File2", FILESIZE=2950           !File2 has no lockword
     30 RENAME "File1" TO "First", "secret"     !Lockword must be specified
     40 RENAME "First" TO "Number1",            !No lockword required
     50 RENAME "File2" TO "Number2"             !No lockword required
     99 END
```

REPEAT

The REPEAT and UNTIL statements define a loop that repeats until the boolean expression in the UNTIL statement is TRUE (nonzero).

Syntax

REPEAT [*stmt*] . . . UNTIL *boolean_expr*

Parameters

stmt Program line that is executed until *boolean_expr* evaluates to TRUE. These statements constitute the loop body. The loop body is always executed once prior to the evaluation of *boolean_expr*.

boolean_expr Considered FALSE if it evaluates to zero; TRUE otherwise.

Examples

```
     10 Nums_read=0
     20 REPEAT                          !Begin loop
     30   READ Number
     40   Nums_read=Nums_read+1
     50 UNTIL Number                    !End loop when Number<>0
     60 PRINT Nums_read," numbers read"
     99 END
```

REPEAT constructs can be nested.

```
     100 REPEAT                   !Begin outer loop
     110   READ Number1
     120   REPEAT                 !Begin inner loop
     130     READ Number2
     140   UNTIL Number1-Number2  !End inner loop
     150 UNTIL Number1+Number2    !End outer loop
     160 PRINT Number1,Number2
     999 END
```

Entering a REPEAT loop in the middle of the loop is considered to be a bad programming practice, and is not recommended. However, calling a local subroutine using GOSUB or calling an external subroutine using CALL from within a REPEAT construct can be useful.

```
     100 REPEAT                         !Begin loop
     110   READ N1,N2
     120   IF (N1 MOD 2) THEN GOSUB 200 !If N1 is odd, exit the loop
     125   PRINT N1
     130 UNTIL N2                       !End loop
     140 STOP
     200 N1=2*N1
     210 RETURN                         !Return to inside of loop
     999 END
```

REPORT EXIT

The REPORT EXIT statement defines a Report Writer section that executes when the STOP REPORT statement executes in a program. This condition typically indicates than an error has been detected and that the report must be stopped abnormally. The REPORT EXIT section allows the program to produce a meaningful message and finish the report gracefully. If no WITH or USING clause is present, the statement produces no output.

Syntax

```
                         [                [LINES]]
REPORT EXIT boolean_expr [WITH num_lines  [LINE ]]

[USING image  [; output_list ]]
```

Parameters

boolean_expr If this expression is zero (FALSE), the REPORT EXIT section does not execute. For nonzero values (TRUE), the REPORT EXIT statement and section will execute. This provides you with more control over early report termination.

num_lines The maximum number of lines expected to be needed by the section statement. This number reflects ALL output done by the section.

image An image string or a line reference to an IMAGE line.

output-list A list of output items, identical to the list used by the PRINT USING statement.

Examples

The following examples show the use of the REPORT EXIT statement.

```
     100 REPORT EXIT TRUE
     100 REPORT EXIT Implementor > 0 WITH 3 LINES USING Rpt_image
```

The REPORT EXIT statement generates an error if no report is active.

If a report section is active (executing) and encounters this statement, then that report section is ended.

The REPORT EXIT section executes ONLY when STOP REPORT is executed in a program. A STOP REPORT command stops the report immediately. When STOP REPORT is executed, the REPORT EXIT section evaluates the Boolean condition first. If the result is FALSE (zero), control returns to the STOP REPORT statement. If the result is TRUE (nonzero), the REPORT EXIT statement and section are executed. A page eject is always done, whether or not this statement is executed.

The REPORT EXIT section is executed even if report output has not started.

NOTE It is recommended that you include a TRIGGER PAGE BREAK at the beginning of the REPORT EXIT section. This ensures that there are enough lines left on the page to print the message, and provides a last page of the report that is dedicated completely to the REPORT EXIT output.

REPORT HEADER

The REPORT HEADER statement marks the beginning of a report description. This statement is required to define a report and is executed when the report output is started. If neither a WITH nor a USING clause is present, the REPORT HEADER produces no output.

Syntax

```
              [                [LINES]]
REPORT HEADER [WITH num_lines   [LINE ]]

[USING image  [; output_list ]]
```

Parameters

num_lines The maximum number of lines expected to be needed by the section statement. This number reflects ALL output done by the section.

image An image string, or a line reference to an IMAGE line.

output-list A list of output items, identical to the list used by the PRINT USING statement.

Examples

The following examples show the REPORT HEADER statement.

```
     100 REPORT HEADER
     100 REPORT HEADER WITH 3 LINES
     200 REPORT HEADER USING Rh_image;DATE$
```

If no report is active, that is, BEGIN REPORT has not been executed, program execution branches from the REPORT HEADER statement to the statement after the matching END REPORT DESCRIPTION statement.

If a report is active and the REPORT HEADER statement is executed, two possible actions can occur. If another report section is active, that section is ended. Otherwise, the statement is unexpected and an error occurs.

The REPORT HEADER section is executed when report output begins. The section only executes once. Report output starts when any of the following statements executes after a BEGIN REPORT statement has been executed:

```
                DETAIL LINE
                TRIGGER BREAK
                TRIGGER PAGE BREAK
                END REPORT
```

When the REPORT HEADER section executes, the REPORT HEADER statement output, if any, is done first. For further information about the REPORT HEADER refer to the WITH and USING clauses. Execution continues with the line following the REPORT HEADER until another Report Writer section statement is encountered.

REPORT TRAILER

The REPORT TRAILER section defines a block of code to be executed at the end of a report only if the END REPORT statement is executed. The report trailer is printed after the break-level trailers. This section is optional. The REPORT TRAILER statement must occur within a report description; that is, between the REPORT HEADER statement and the END REPORT DESCRIPTION statement. If neither the WITH or USING clause is present, the statement produces no output, but, there must be at least one line of space left on the page.

Syntax

```
               [               [LINES]]
REPORT TRAILER [WITH num_lines  [LINE ]]

[USING image  [; output_list ]]
```

Parameters

num_lines The maximum number of lines expected to be needed by the section statement. This number reflects ALL output done by the section.

image An image string, or a line reference to an IMAGE line.

output-list A list of output items, identical to the list used by the PRINT USING statement.

Examples

The following examples show the use of the REPORT TRAILER statement.

```
     100 REPORT TRAILER
     100 REPORT TRAILER WITH 3 LINES USING Rt;DATE$
```

If a report is not active, the REPORT TRAILER statement generates an error.

If the statement is encountered when a report section is not executing, an error occurs. If a report section is active; for example, a TRAILER section, that section ends.

The REPORT TRAILER section becomes active when END REPORT executes. All of the break level trailers are printed before the report trailer. A page trailer is printed after the report trailer.

RESAVE KEY

The RESAVE KEY statement's action is dependent on whether a filename parameter is included in the statement. If a filename parameter is included and the file does not previously exist, the RESAVE KEY statement stores the typing aid definitions in a BKEY file. The file to which the information is saved has a special format and a BKEY file code. If no filename parameter is specified, the RESAVE KEY statement causes HP Business BASIC/XL to store the current typing aid key definitions internally as the current definition. The RESAVE KEY statement does not save information for keys defined as branch-during-input keys, it saves only the key definition information for keys defined as typing aid definitions.

The file referenced by *fname* must exist and have a BKEY file format. An error occurs if the format is not correct. If any user-definable keys have been defined as branch-during-input keys when a RESAVE or SAVE statement is executed in the interpreter or in a compiled program, the last typing aid key definition for that key is the information written to the BKEY file.

Typing aid keys are discussed in detail in chapter 8, User-definable Keys.

Syntax

RESAVE KEY [*fname*]

Parameters

fname A file name represented by a quoted string literal, an unquoted string literal or a string expression as described in chapter 6.

Examples

```
     RESAVE KEY typeaid

     200 RESAVE KEY typeaid1             !File is typeaid1
     210 RESAVE KEY Filename$ + "." + Groupname$   !Uses the data in Filename$
     211                                           !and Groupname$
```

RESTORE

The RESTORE statement resets the data pointer to the beginning of a DATA statement so that the data can be reused.

Syntax

RESTORE [*line_id*]

Parameters

line_id Line identifier of a DATA statement in the same program unit as the RESTORE statement. The RESTORE statement positions the data pointer at the first datum in the specified DATA statement. If no *line_id* is specified, then the data pointer is positioned to the first datum in the first DATA statement in the program unit.

Example

```
     100 DATA 1,2,3
     110 DATA 4,5,6
     120 READ A,B,C     !A=1, B=2, C=3
     130 READ D,E,F     !D=4, E=5, F=6
     140 RESTORE 110    !Applies to line 110
     150 READ G,H,I     !G=4, H=5, I=6  (from line 110)
     160 RESTORE        !Applies to line 100 (by default)
     170 READ J,K,L     !J=1, K=2, L=3  (from line 100)
     999 END
```

RETURN

The RETURN statement returns control to the program unit that called a subroutine or multi-line function. When used in a subroutine, control is returned to the statement following the GOSUB statement. When used in a multi-line function, the value of the expression immediately following RETURN is returned to the statement or expression where the call was made.

Syntax

RETURN [*expr*]

Parameters

expr — A numeric expression if the RETURN statement is in a numeric function, and a string expression if the RETURN statement is in a string function.

HP Business BASIC/XL evaluates the expression and returns it value to the program unit that called the function. If the function is numeric, HP Business BASIC/XL converts the value to the function type before return it (the function type is either declared in the DEF FN statement that defines the function, or if not declared is the default numeric type).

This parameter is not used when the RETURN is used in conjunction with a subroutine.

If a multi-line function does not contain a RETURN statement, an error occurs when execution reaches the FNEND statement. The multi-line function RETURN statement that returns a value is legal only within a multi-line function. It is illegal in the main program or a subprogram.

Examples

```
 10 READ A              !Example of RETURN within a multi-line function
 20 C= FNAdd(A)
 30 PRINT C
 99 END
100 DEF FNAdd(X)
120   Y= X+2
130   RETURN Y          !FNAdd returns value to line 20
140 FNEND
```

GOSUB statements can be nested; that is, calls to more than one GOSUB statement can be executed before a RETURN statement is executed. The first RETURN statement executed matches the most recently executed GOSUB statement, the second RETURN statement executed matches the second most recently executed GOSUB statement, and so on.

```
 10 REM Main Program Unit
 20 PRINT "B"
 25 GOSUB First                  !Go to line 100; First prints "ASI"
 30 PRINT "C"
 35 STOP
100 First:  REM First subroutine
110   PRINT "A";
120   GOSUB second               !Go to line 200; Second prints "S"
130   PRINT "I"
140   RETURN                     !Return to line 25 to print "C"
200 Second: REM Second subroutine
210   PRINT "S";
220   RETURN                     !Return to line 130 to print "I"
999 END
```

The output from the above program is BASIC.

After a GOSUB statement is executed, the subroutine to which it transfers control is "open". When a matching RETURN statement is executed, the subroutine is "closed." An error occurs if a RETURN statement is executed when no subroutine is open.

```
 10 REM Main Program Unit
 20 GOSUB 100              !Open subroutine at line 100
 25 RETURN                 !No open subroutine;error occurs
 30 STOP
100 REM Subroutine
110 PRINT "In sub"
120 RETURN                 !Close subroutine; return to line 30
999 END
```

SAVE KEY

The SAVE KEY statement's action is dependent on whether a filename parameter is included in the statement. If a filename parameter is included and the file does not previously exist, the SAVE KEY statement stores the typing aid definitions in a BKEY file. The file to which the information is saved has a special format and a BKEY file code. If no filename parameter is specified, the SAVE KEY statement causes HP Business BASIC/XL to store the current typing aid key definitions internally as the current definition. The SAVE KEY statement does not save information for keys defined as branch-during-input keys, it saves only the key definition information for keys defined as typing aid definitions.

The SAVE KEY statement saves key labels, it does not save any actions that a program has set up when it traps those labeled keys. If a key is pressed, it will paint the screen. Any actions associated with that key have not been saved so they will not be performed.

NOTE It is important to do a SAVE KEY without the *fname* parameter following the initial setting of the fields of the user-definable keys for use as typing aid keys.

If this is not done, exiting from a program containing an OFF KEY statement restores the user-definable keys to the values present before you set those displayed on entry to the program. If you had just entered the interpreter, the values of the typing aid keys are restored to the terminal's default typing aid key definitions rather than your user-defined typing aid keys. In other words setting the typing aid key definitions and then executing, a program containing the OFF KEY statement, restores the terminal's default typing aid definitions.

HP Business BASIC/XL stores the values of typing aid keys internally. SAVE KEY without an *fname* parameter can be used in conjunction with GET KEY without an *fname* parameter to access HP Business BASIC/XL's internally stored values. The GET KEY statement without an *fname* parameter restores the definitions of the keys present at the last previous SAVE KEY statement without an *fname* parameter if the following condition is met: no other SAVE KEY, RESAVE KEY, GET KEY or SCRATCH KEY statement precedes the GET KEY statement without an *fname* parameter. Thus, GET KEY can be used without an *fname* parameter to restore definitions of any of the fields changed by the method outlined in the terminal's reference manual.

Syntax

SAVE KEY [*fname*]

Parameters

fname   A file name represented by a quoted string literal, an unquoted string literal or a string expression as described in chapter 6.

Examples

The following examples show the use of the SAVE KEY statement, and also show that SAVE key is also available as a command.

```
     SAVE KEY
     SAVE KEY typeaid

     100 SAVE KEY
     110 SAVE KEY typeaid1
```

SCRATCH KEY

The SCRATCH KEY statement resets the current typing aid contents of the attribute, label, and key definition fields of an individual or group of user-definable keys. The values of each field for the specified keys are assigned the default values, blank labels, local, and BEL. HP Business BASIC/XL also stores the default values of the keys as those retrieved by a GET KEY statement without a filename parameter.

Syntax

SCRATCH KEY [*key_number_list*]

Parameters

key_number_list A list of integers selected from the set of [1..8] or numeric expressions that evaluate to an integer in the range of [1..8] separated by commas or semicolons. If the integer is not in the specified range, an error occurs. No more than eight values can be specified for each statement. If no values are specified, all of the keys are scratched.

Examples

```
     SCRATCH KEY        ! Resets typing aid definition of all user-definable
                        ! keys
     100 SCRATCH KEY    ! Resets typing aid definition of all user-definable keys
     110 SCRATCH KEY 1  ! Resets typing aid definition of user-definable key
                        ! number one
     120 SCRATCH KEY 1,2,6
     130 SCRATCH KEY Typing_aid_key_number
```

SEARCH

The SEARCH statement starts the database retrieval process for HP Business BASIC/XL's Database Sort Feature. Functions, built-in as well as user-defined, can be used in the search condition. When the SEARCH statement is executed, the data sets contained in the thread list are accessed in the order and hierarchy specified by the THREAD IS statement. The data retrieved from each data set are unpacked into the local variables as defined in the respective IN DATASET statement. For each type of data sets from the thread list, the search condition is evaluated. If the search condition is true, the record pointers to the data set records that have been read are written out to the workfile; otherwise, they are ignored and the next data set record is searched. The workfile is a file created and used by the program to store the record number of the data set items that satisfy the search condition. You can access this file.

If no search condition is needed, the keyword ALL can be used and all records are retrieved.

When the SEARCH statement is executed, the workfile can be empty or nonempty. If the workfile is empty, all data records in the data sets, mentioned in the threadlist, are searched. If, however, the workfile is nonempty searching is done only on the records whose pointers are contained in the workfile. The pointers to those records whose data fail the search condition are dropped from the workfile.

Syntax

```
                      {search_condition }
SEARCH USING line_id; {ALL              }
```

Parameters

line_id Line label on line number that identifies the line on which a THREAD IS statement is defined.

search_ condition Any logical expression.

Examples

The following shows the use of the SEARCH statement.

```
         400  SEARCH USING 300; ALL
         410  SEARCH USING Thread_list; TRIM$(Name$)="HP" AND Price > 0
```

SELECT

The SELECT, CASE, CASE ELSE, and END SELECT statements define a construct that executes one set of statements, depending on the value of an expression.

Syntax

```
                   [CASE case_descriptor ]   [CASE ELSE]
                   [[stmt ]              ]   [[stmt ]   ]
SELECT select_expr  [.                   ]...[.         ] END SELECT
                   [.                   ]   [.         ]
                   [.                   ]   [.         ]
```

Parameters

select_expr An expression evaluated and compared to the *case descriptor* 's. If the value is numeric, it is converted to the default numeric type first.

case_descriptor One of the following:

* *num_item* [, *num_item*]...

* *str_item* [, *str_item*]...

num_item One of the following:

* *num_lit* [TO *num_lit* [EXCLUSIVE]

* {<,<=,>=,>} *num_lit*

EXCLUSIVE If specified, the range excludes the two specified *num_lits*. For example, CASE 10 TO 20 EXCLUSIVE excludes both 10 and 20.

str_item One of the following:

* *str_lit* [TO *str_lit* [EXCLUSIVE]

* {<,<=,>=,>} *str_lit*

Each *case_descriptor* must be a numeric literal if *select_expr* evaluates to a numeric value and a string literal if it evaluates to a string value.

If the *select_expr* value is equal to one of the specified *case_descriptor* literals or is within the range specified in the *case_descriptor*, then the case clause associated with the *case_descriptor* is executed.

stmt Program line. It is executed if *select_expr* fits the associated *case_descriptor*. Each sequence of program lines between a CASE and either the next CASE or an END SELECT constitutes a case clause.

Examples

```
     100 SELECT Number
     110 CASE < 0                !If Number is negative.
     120   READ Number
     130 CASE 0                   !If Number is zero
     140   LET Number=Default
     150 CASE 1 TO 10             !If Number is 1 - 10
     160   PRINT Number
     170   GOTO Routine1
     180 CASE 10 TO 20 EXCLUSIVE  !If Number is 11 - 19 (due to the EXCLUSIVE keyword)
     190   PRINT Number
     200   Number=2*Number
     210   GOSUB Routine2
     220 CASE 20,30,40            !If Number is 20, 30 or 40
     230   PRINT Number
     240   GOSUB 450
     250 CASE ELSE                !If Number is any other value
     260   LET Number=Default
     270 END SELECT
```

HP Business BASIC/XL evaluates the select expression and compares its value with each case descriptor's starting with the first and proceeding in line number order, until a case descriptor describes the value or every case descriptor has been checked.

When a case descriptor describes the value, the statements in its case clause are executed; then, control is transferred to the statement following the END SELECT statement. If more than one case descriptor describes the value, only the first one's case clause is executed.

If no case descriptor describes the value, the CASE ELSE clause is executed, if there is one. If there is no CASE ELSE clause, control is transferred to the statement following the END SELECT statement.

The string value of a *select_expr* is compared with the quoted string literals character by character. The following code segment outputs In the first half of the dictionary.

```
      100 Str_var$ = "dog"
      110 SELECT Str_var$
      120 CASE "a" To "m"
      130   PRINT "In the first half of the dictionary."
      140 CASE "dog"
      150   PRINT "my pet."
      160 END SELECT

      10 SELECT A+B
      20 CASE < 0                   !A+B < 0
      21   PRINT "Negative"
      22   STOP
      30 CASE 0                     !A+B = 0
      31   PRINT "Zero"
      32   LET X=Default
```

```
 40 CASE 1 TO 10               !1 <= A+B <= 10
 41   PRINT "1 thru 10"
 42   GOSUB Routine1
 50 CASE 10 TO 20 EXCLUSIVE    !10 < A+B < 20
 51   PRINT "Between 10 & 20"
 52   GOSUB Routine2
 60 CASE 20,22,24              !A+B = 20, 22, or 24
 61   PRINT "Special Case #1"
 62   GOSUB Spec_case1
 70 CASE 21,23,25              !A+B = 21, 23, or 25
 72   PRINT "Special Case #2"
 73   GOSUB Spec_case2
 80 CASE > 30                  !A+B > 30
 81   PRINT "Over 30 by:"
 82   PRINT (A+B)-30
 83   STOP
 90 CASE ELSE                  !26 <= A+B <= 30
 91   PRINT "26 thru 30"
 92   GOSUB Routine3
100 END SELECT
```

SELECT constructs can be nested.

```
100 SELECT Color1$                  !Start outer construct
110 CASE "red", "blue", "yellow"    !First case in outer construct
120   GOSUB Primary
130   SELECT Color1$                !Start first inner construct
140     CASE "red"                  !Case in first inner construct
150       PRINT "RED"
160       PRINT "ORANGE,PURPLE"
170     CASE "blue"                 !Case in first inner construct
180       PRINT "BLUE"
190       PRINT "PURPLE,GREEN"
200     CASE "yellow"               !Case in first inner construct
210       PRINT "YELLOW"
220       PRINT "ORANGE,GREEN"
230   END SELECT                    !End first inner construct
240 CASE "green","purple","orange"  !Second case in outer construct
250   GOSUB Secondary
260   SELECT Color2$                !Start second inner construct
270     CASE "green"                !Case in second inner construct
280       PRINT "YELLOW+BLUE"
290     CASE "purple"               !Case in second inner construct
300       PRINT "BLUE+RED"
310     CASE "orange"               !Case in second inner construct
320       PRINT "RED+YELLOW"
330   END SELECT                    !End second inner construct
340 END SELECT                      !End outer construct
```

Entering the middle of a SELECT construct from a statement other than the SELECT statement is considered to be a bad programming practice, and is not recommended. However, if control is transferred to a statement that is in the middle of a SELECT construct, execution proceeds in line number order starting with that statement. When a CASE, CASE ELSE, or END SELECT statement is reached, control is transferred to the statement following the END SELECT statement.

Control can be transferred out of a SELECT construct and then back by using a GOSUB or CALL statement.

```
100 SELECT T
110 REM Clause 1
120 CASE < 0
121   CALL Sub1        !Jump out of SELECT construct
122   PRINT T          !Return to construct from 520
130 REM Clause 2
131 CASE 0
132   GOSUB 300        !Jump out of construct
```

```
133   PRINT 2*T        !Return to construct from 310
134   PRINT T
140 REM Clause 3
141 CASE > 0
142   GOSUB 400        !Jump out of construct
150 END SELECT         !Return to construct from 410
160 STOP
300 REM Do anything    !Arrive from Clause 2, line 132
310 RETURN             !Return to Clause 2, line 133
400 REM Do anything    !Arrive from Clause 3, line 142
410 RETURN             !Return to Clause 3, line 122
500 SUB Sub1           !Called from Clause 1 line 121
510 REM In procedure
520 SUBEND             !Return to clause 1 line 122
999 END
```

SEND OUTPUT TO

The SEND OUTPUT TO statement specifies the output device for the PRINT statement, the PRINT USING statement, and the default target file for the COPYFILE statement. If the *dev_spec* is a disk file that already exists, additional information is appended to the file.

Syntax

[SEND] OUTPUT [TO] *dev_spec*

If a program does not contain a SEND OUTPUT TO statement, the output device for the PRINT statement is the terminal if HP Business BASIC/XL is running interactively or the standard list device if HP Business BASIC/XL is running in a job stream. The default target file for the COPYFILE statement is the standard list device.

The SEND OUTPUT TO statement overrides the COPY ALL OUTPUT TO statement; if a program contains both statements, then the PRINT statement output is displayed only on the device specified in the SEND OUTPUT TO statement. A SEND OUTPUT TO statement generates an error if it executes between the initiation of report writer output with the DETAIL LINE, TRIGGER BREAK, TRIGGER PAGE BREAK, or END REPORT statement and termination of the report.

Examples

```
>list
10 SYSTEM "FILE LP;DEV=LP"
20 SYSTEM "FILE LASER3;DEV=PP,3;ENV=LP602.HPENV.SYS;CCTL"
30 SEND OUTPUT TO "*LP"    !Output sent to LP
40 PRINT "Send a line to the printer."
50 SEND OUTPUT TO DISPLAY,MARGIN=40    !Output sent to DISPLAY
60 PRINT "Line to display on the terminal showing margin at 40."
70 SEND OUTPUT TO "*LASER3"    !Output sent to LASER3
80 PRINT "Send a line to the laser printer."
>run
Line to display on the terminal showing
margin at 40.
>
```

SEND SYSTEM OUTPUT TO

The SEND SYSTEM OUTPUT TO statement specifies the output device to which interpreter output is sent.

Syntax

[SEND] SYSTEM OUTPUT [TO] *dev_spec*

HP Business BASIC/XL interpreter output that is normally sent to the system's standard list device, usually, the terminal, can be redirected to other output devices specified by *dev_spec*. The interpreter statements and commands effected by SEND SYSTEM OUTPUT TO are CHANGE, COPY, FIND, INFO, LIST, LIST SUBS, MODIFY, MOVE, and REDO. If a program does not contain a SEND SYSTEM OUTPUT TO statement, output is sent to the system standard list device.

Examples

```
     100 SEND SYSTEM OUTPUT TO DISPLAY                  ! Terminal
     110 SEND SYSTEM OUTPUT TO PRINTER                  ! Spool file sent to
     115                                                ! system printer
     120 SEND SYSTEM OUTPUT TO "SYSOUT", FILESIZE 230   ! A user-defined file
```

SET PAGENUM

The SET PAGENUM statement allows you to change the page number maintained by the Report Writer. The page number available through the PAGENUM built-in function is affected. This statement can appear anywhere in the subunit containing the report description.

Syntax

```
            [TO]
SET PAGENUM [, ] page_expr
            [; ]
```

Parameters

page_expr A numeric expression. Its value must be a non-negative number in the INTEGER range.

Examples

```
     100 SET PAGENUM TO Last_page + 3  !Pagenumber is 3 greater than the
     101                               !number in Last_page
     100 SET PAGENUM TO 0              !Pagenumber is 0
```

The Report Writer maintains a page number for use by the user. The PAGENUM built-in function returns this page number. The page number can be changed at any time using the SET PAGENUM statement. This allows the addition of pages from other sources in a printed report.

The page number can be set to zero, which is particularly useful for reports with a report header on a page by itself. Negative page numbers are not allowed.

The page number does not affect the SUPPRESS FOR statement, which suppresses report output. The count of page breaks is distinct from the page number count kept by the report writer.

SETLEN

The SETLEN statement sets the current length of a HP Business BASIC/XL string variable to a specified length.

Syntax

```
               {TO}
SETLEN str_var  {, } num_expr
               {; }
```

Parameters

str_var The variable who length is to be set. A string variable or a string array element.

num_expr A numeric expression that evaluates to the length of the string.

An HP Business BASIC/XL program can pass an HP Business BASIC/XL string as an actual parameter to a Pascal PAC or C array formal parameter, but only the string characters are passed (the current string length is not). If the HP Business BASIC/XL program passes the string by reference, and the Pascal or C external routine changes its current length, then the HP Business BASIC/XL program must reset the current length when it resumes control.

Examples

```
     100 EXTERNAL PASCAL INTEGER FNAbbreviate(BYTE A$)
     110 INTEGER New_length
     120 READ String$
     130 New_length = FNAbbreviate(String$)   !Calls the function to set
     131                                      !the length
     140 SETLEN String$ TO New_length         !Sets string to length
     141                                      !returned by FNAbbreivate
     150 PRINT String$
     999 END
```

SHORT

This statement defines a variable as a type SHORT REAL.

The SHORT statement can also be used as an option with the REAL, DECIMAL or INTEGER statements to declare SHORT REAL, SHORT DECIMAL, or SHORT INTEGER types. See each of those statements for syntax and examples. SHORT and SHORT REAL are equivalent.

Syntax

```
      {num_var } [ {num_var }]
SHORT {arrayd  } [,{arrayd  }]...
```

Parameters

num_var Name of scalar numeric variable to be declared.

arrayd Numeric array description. The syntax for the array is described under the DIM statement.

Examples

```
     100 SHORT I
     120 SHORT L,M
     130 SHORT A(3)
```

SORT

The SORT statement sorts the record pointers contained in a workfile. The SORT statement must also specify the sort keys and their usage (ascending/descending). Since the record pointers must be sorted by the data to which they point, the database must be accessed once more. However, only the records whose pointers are in the workfile need to be read. The order in which the data sets are to be read is governed by the thread list. Sort keys specified must be defined in an IN DATASET statement. After sorting is done, the workfile contains the same record pointers but they are sorted.

The SEARCH and SORT statements are related, yet independent statements. SEARCH can be done before or after SORT. If no SEARCH has been done when SORT is executed, the workfile is empty, and a SEARCH ALL is issued first. In other words, the SORT statement does an implicit SEARCH ALL if it is executed before a SEARCH statement. On the other hand, if the SEARCH is done after the SORT, then all the record pointers contained in the workfile are searched. The workfile may then contain fewer records after a SEARCH because the record pointers to any data records that do not satisfy the SEARCH condition are deleted from the workfile.

Syntax

SORT USING *line_id; key_list*

Parameters

line_id Line label or line number that identifies the line on which the THREAD IS statement is defined.

key_list List of variables. The DES keyword can follow each variable in the list. Specifying DES means that the data is sorted in descending order. If not specified, data are sorted in ascending order. Whole arrays are not allowed.

Examples

```
     600  SORT USING 300; A DES, B   !Sorts using THREAD statement on line 300
```

SORT ONLY

As mentioned in the description of the SORT statement, SORT does an implicit SEARCH ALL if the workfile is empty. SORT ONLY does not do the SEARCH ALL. It sorts the database records whose record pointers are already in the workfile. An error is generated if the workfile is empty. SORT ONLY does only half of what SORT can do. Its main purpose is to save the amount of code generated by the compiler when only a SORT is required.

Syntax

SORT ONLY USING *line_id; key_list*

Parameters

line_id Line label or line number list that identifies the line on which THREAD IS statement is defined.

key_list List of variables. The DES keyword can follow each variable in the list. Specifying DES means that the data will be sorted in descending order. If not specified, data are sorted in ascending order. Whole arrays are not allowed.

Examples

```
     100 SORT ONLY USING 200;mp_ham$, Loc DES  !Sorts using THREAD in line 200
     200 THREAD IS 300,400
     300 IN DATASET Dset1$ USE SKIP 10, Emp_nam$
     400 IN DATASET Dset2$ USE Addr$, LOC
```

STANDARD

The STANDARD statement sets the default numeric output format to standard. The FLOAT and FIXED statements also set the default numeric output.

Syntax

STANDARD

Standard numeric format depends on the type of the number being formatted. Floating-point literals are of the default numeric type. Table 4-15 tells which digits are printed for each numeric type.

Table 4-15. Standard Numeric Output Formats

Type	Digits Printed
SHORT INTEGER INTEGER	All
SHORT DECIMAL DECIMAL	Most significant 6
DECIMAL	Most significant 12
REAL	Most significant 16

Examples

```
     10 STANDARD
     20 PRINT 123;.4567;-79810;-1.235E+47
     99 END
```

The above program prints:

```
     123   .4567   -78910   -1.235E+47
```

STOP

The STOP statement terminates program execution.

Syntax

STOP

The STOP statement can be in a main program or a subunit. A program can contain more than one STOP statement.

Examples

```
     100 READ I
     110 ON I GOSUB 200,300,400
     120 STOP                        !After return from the above ON GOSUB, the program
     130                             !stops.
     200 I=I+1
     210 RETURN
     300 I=I+I
     310 RETURN
     400 I=I*I
     410 RETURN
     999 END
```

The STOP, END, or SCRATCH statement can stop a program. When a program stops, the following occurs:

- Subroutine return pointers are lost.
- FOR NEXT loop "bookkeeping" is lost.
- Subunit call "bookkeeping" is lost.

- Files are closed (except those declared in COMMON).

- Data pointers are lost.

- ON END, ON ERROR, ON DBERROR, ON GOTO, and ON GOSUB statements are deactivated.

STOP REPORT

The STOP REPORT statement is a Report Writer statement that can be used to terminate a report prematurely. This statement can also be used when the user does not know if a report is active as no error is generated by this statement.

The STOP REPORT statement is implicitly used when a report ends abnormally for other circumstances, such as a STOP statement or END statement. Note that this statement can occur anywhere in the report subunit.

Syntax

```
STOP REPORT
```

Examples

```
     100 STOP REPORT   !Terminates a report
```

The STOP REPORT statement does not start report output if it has not already begun.

This statement performs different actions for active reports, depending on its usage. As a program statement, the STOP REPORT statement looks for a REPORT EXIT section in the report description. If present, the REPORT EXIT condition is evaluated. This section is then executed if the expression is true (nonzero). It is not executed if the condition is false or if the REPORT EXIT section does not exist.

As a command, or when called implicitly, for example, during a STOP, STOP REPORT does not look for a REPORT EXIT statement. The report is simply terminated.

If report output has started, the STOP REPORT statement always prints a page eject as its last action. This is done to guarantee that HP Business BASIC/XL does not print any system output on the report. Thus, a TRIGGER PAGE BREAK is not needed in the REPORT EXIT section.

STOP REPORT automatically ends all GOSUBS that were executed by the report; for example, all GOSUBS done after the last executable Report Writer statement are prematurely ended. Execution resumes at the line following STOP REPORT. This includes ON ERROR GOSUB, ON HALT GOSUB and so on.

In all cases, STOP REPORT deactivates a report. No errors can prevent this from happening.

SUB

The SUB statement defines the beginning of a subprogram. It is not executable.

Syntax

```
{SUBPROGRAM}
{SUB       } sub_name  [(f_param  [, f_param ]...)]
```

Parameters

sub_name   Subprogram name. This is an identifier.

f_param A formal parameter. One of the following:

[*type*] *num_var*

num_var is a numeric variable and *type* is one of the following:

SHORT
SHORT REAL
SHORT INTEGER
SHORT DECIMAL
REAL
INTEGER
DECIMAL

If *type* is not specified, *num_var* is declared with the default numeric type. If *type* is specified, it determines the type of each *num_var* between it and the next *type* or the next nonnumeric *f_param*.

str_var $

String variable. Its maximum length is the same as that of the actual parameter.

[*type*] *num_var* ([*[,*]...])

Abbreviated numeric array declaration, with one asterisk per dimension or no asterisks. No asterisks specifies any number of dimensions. Either format is legal, but the format without asterisks is noncompilable when there is no reference in the subunit that allows the compiler to determine the number of dimensions for the array.

type is one of the following:

SHORT
SHORT REAL
SHORT INTEGER
SHORT DECIMAL
INTEGER
REAL
DECIMAL

If *type* is not specified, *num_var* is declared with the default numeric type. If *type* is specified, it determines the type of each *num_var* between it and the next *type* or the next nonnumeric *f_param*.

str_var $ ([*[,*]...])

Abbreviated string array declaration, with one asterisk per dimension or no asterisks. No asterisks specifies any number of dimensions. Either format is legal, but the format without asterisks is noncompilable. The maximum length of each element is the same as declared for the actual parameter by the

	calling program unit.
#fnum	A file designator. *fnum* is a positive short integer greater than zero. The file designated by the actual parameter *file designator* is referenced by *#fnum* within the subprogram.

Examples

```
     SUBPROGRAM Sub1 (A(*), B$(*,*), INTEGER X,Y, P$, C,D, #15)
     SUB Sub2 (A(), B$(*), INTEGER X,Y P$, C,D, #15)
```

The above statements define the beginning of subprograms named Sub1 and Sub2 that have the following formal parameters:

Parameter	**Type**
A	Array of default numeric type.
B$	A 2-dimensional string array variable in Sub1. A 1-dimensional string array variable in Sub2.
X and Y	Integer variables.
P$	String variable.
C and D	Variables of default numeric type.
#15	File designator.

NOTE If a program has more than one subprogram with the same name, all calls refer to the first one; that is, the one with the lowest-numbered SUBPROGRAM statement. The others cannot be called.

SUBEND

The SUBEND statement ends a subprogram. Like the SUBPROGRAM statement, which begins a subprogram, the SUBEND statement is not executable. It returns control to the program unit that called the subprogram. Specifically, the statement returns control to the line following the CALL statement that originally called the subprogram. Although the SUBEND statement can be input as SUBEND or SUB END, HP Business BASIC/XL will always list it as SUBEND.

Syntax

```
{SUBEND }
{SUB END}
```

If a subprogram does not end with a SUBEND statement; that is, if the SUBEND statement is accidentally omitted, SUBEND is implied. Control does not pass to the following subunit.

Example

```
      10 CALL Sub1(L,M,N)      !Call Sub1 from main program
      99 END                   !End main program
     100 SUB Sub1 (A,B,C)      !Begin Sub1
     110  A=B+C
     120  CALL Sub2(A,B,C)     !Call Sub2 from Sub1
     130 SUBEND                !End Sub1
     140 !
     200 SUB Sub2(X,Y,Z)       !Begin Sub2
     210  X=Y*Z
     220 SUBEND                !End Sub2
```

The SUBEND statement is legal only in a subprogram. It is illegal in the main program or a multi-line function.

It is good programming practice to end a subprogram with a SUBEND statement, and use SUBEXIT statements within the subprogram. The SUBEND statement can appear more than once within a subprogram, and it need not be the last line. One subprogram ends where the next subunit begins, or where the program ends.

SUBEXIT

The SUBEXIT statement returns control to the program unit that called the subprogram. SUBEXIT can be used to exit a subprogram prior to execution of the SUBEND statement. Like the SUBEND statement, the SUBEXIT statement returns control to the statement following the CALL statement that called the subprogram. Although the SUBEXIT statement can be entered as either SUBEXIT or SUB EXIT, HP Business BASIC/XL will always enter it as SUBEXIT.

Syntax

```
{SUBEXIT }
{SUB EXIT}
```

The SUBEXIT statement is optional. If a program does not contain one, execution of the SUBEND statement returns control to the calling program unit.

A program can contain more than one SUBEXIT statement. Usually, a SUBEXIT statement is executed conditionally.

Example

```
      10 READ A,B
      20 CALL Sub(A,B)             !Control transfers to line 100
      30 PRINT "DONE"
      80 DATA 1,2
      99 END
     100 SUB Sub(X,Y)              !Start of Subprogram
     105   INTEGER Z
     110   IF X<0 THEN SUBEXIT    !If X < 0, control returns to line 30
     120   LET Z=X+Y
     130   IF Z<0 THEN SUBEXIT    !If Z < 0, control returns to line 30
     140   PRINT Z
     999 SUB END
```

A SUBEXIT statement is legal only in a subprogram. SUBEXIT is illegal in a main program or multi-line function.

SUBPROGRAM

The SUBPROGRAM statement is the long form of the SUB statement. Refer to the SUB statement for information.

SUPPRESS AT

The SUPPRESS AT statement allows the Report Writer to produce a report at particular summary levels. All output from lower numbered levels are executed. Those sections with levels at or higher than the indicated level are not executed. Except for the printout reduction, a report is produced exactly as if all sections were being printed. That is, all breaks occur normally and all totals are accumulated.

There cannot be more than one SUPPRESS AT statement in a report description.

Syntax

```
                  {AT [LEVEL]}
SUPPRESS [PRINT] {LEVEL     } suppress_level
```

Parameters

suppress_level A numeric expression with a value from zero to nine. A level of zero causes the statement to be ignored. All output from the *suppress_level* and higher summary levels is suppressed.

Examples

The following examples show the use of the SUPPRESS AT statement.

```
     100 SUPPRESS PRINT AT N+M
     100 SUPPRESS AT 4
```

The SUPPRESS AT statement is evaluated by BEGIN REPORT. It is busy only during evaluation. If the indicated level is zero, the statement is ignored and all output takes place.

Once report output starts, output is only produced by the HEADER and TRAILER sections with summary levels lower than SUPPRESS AT. The report sections (REPORT HEADER, REPORT TRAILER, and REPORT EXIT) are at level zero and can never be suppressed with this statement. The PAGE HEADER and PAGE TRAILER sections are not affected by this statement.

Only the actual printing of the report is affected. BREAK IF and BREAK WHEN conditions are still checked, and totals are still accumulated. PRINT and DETAIL LINE output are not affected by SUPPRESS AT. Only the HEADER and TRAILER sections are suppressed.

SUPPRESS FOR

The SUPPRESS FOR statement provides a means of inhibiting print for a specified number of pages at the beginning of a report. The report is generated normally, but no output is actually produced until the correct number of pages have been processed.

Syntax

```
                               [PAGE ]
SUPPRESS [PRINT] FOR num_pages  [PAGES]
```

Parameters

num_pages A numeric expression indicating how many pages should be skipped before printing starts. This must be a non-negative valid INTEGER value.

Examples

The following show the use of the SUPPRESS FOR statement.

```
     100 SUPPRESS PRINT FOR Spf PAGES
     100 SUPPRESS FOR 1 PAGE
```

The SUPPRESS FOR statement is evaluated by BEGIN REPORT, and is busy during its evaluation.

The report is generated normally, but all output is prevented by this statement. The Report Writer counts the number of physical pages produced. When the correct number of pages has been produced, actual output starts up. All Report Writer errors, including the not enough lines on a page, may occur while output is suppressed.

As an example of using this statement, assume that the first nine pages of a report have been printed, and an error occurs on the tenth page. After fixing the error, the user may wish to re-run the program. Since the first nine pages are correct, the following statement prevents the reprinting of those pages:

```
                    SUPPRESS PRINT FOR 9 PAGES
```

Suppressing print for zero pages allows all output to take place.

SUPPRESS HEADER

This statement enables and disables the execution of the PAGE HEADER section. Unlike the TRIGGER PAGE BREAK options, the suppression is in force across multiple pages.

This statement can occur anywhere in the same subunit as the report description.

Syntax

```
                {ON }
SUPPRESS HEADER {OFF}
```

Examples

```
     100 SUPPRESS HEADER ON
```

The SUPPRESS HEADER ON statement prevents the execution of all PAGE HEADER sections until a SUPPRESS HEADER OFF statement is encountered. The statement takes effect beginning with the next PAGE HEADER to be printed. Thus, it cannot affect the current page once the page has started.

If the SUPPRESS HEADER ON statement appears in the REPORT HEADER section, the PAGE HEADER does not appear after the report header.

The SUPPRESS HEADER OFF statement re-enables the output of PAGE HEADER. It takes effect on the next page.

The SUPPRESS options of the TRIGGER PAGE BREAK do not reset the settings of the SUPPRESS HEADER statement. For example, if SUPPRESS HEADER ON has been executed, then both of the following statements suppress the page header:

```
     TRIGGER PAGE BREAK
     TRIGGER PAGE BREAK, SUPPRESS HEADER
```

Because the page header is suppressed anyway, no output is expected. The *temporary* suppression in the second statement does not cause the page header to resume printing on the next page. Only a SUPPRESS HEADER OFF statement can do that.

SUPPRESS TRAILER

This statement enables and disables the execution of the PAGE TRAILER section. Unlike the TRIGGER PAGE BREAK options, the suppression is in force across multiple pages.

This statement can occur anywhere in the same subunit as the report description.

Syntax

```
                 {ON }
SUPPRESS TRAILER {OFF}
```

Examples

```
     100 SUPPRESS TRAILER OFF
```

The SUPPRESS TRAILER ON statement prevents the execution of the PAGE TRAILER section. If this statement occurs during the execution of a PAGE TRAILER, it does not take effect until the next page. If executed before the PAGE TRAILER for the current page, the PAGE TRAILER does not appear on the current page.

When the PAGE TRAILER is suppressed, the lines normally reserved for the page trailer are available to you. Therefore, this statement can increase the size of a page. The bottom margin reserved in the PAGE LENGTH statement are not suppressed.

The SUPPRESS TRAILER OFF statement re-enables the execution of the PAGE TRAILER section. Normally, this is all that this statement does. However, if you have already printed in the area reserved for the page trailer, the SUPPRESS TRAILER OFF statement causes an automatic page break.

The SUPPRESS options of the TRIGGER PAGE BREAK do not reset the settings of the SUPPRESS TRAILER statement. For example, if SUPPRESS TRAILER ON has been executed, then both of the following statements suppress the page trailer:

```
     TRIGGER PAGE BREAK
     TRIGGER PAGE BREAK, SUPPRESS TRAILER
```

Because the page trailer is suppressed anyway, no output is expected. The *temporary* suppression in the second statement does not cause the page trailer to resume printing on the next page. Only a SUPPRESS TRAILER OFF statement can do that.

SYSTEM

The SYSTEM statement executes an operating system command from HP Business BASIC/XL.

Syntax

As a statement or command:

```
       [         [{,}                    ]]
SYSTEM [str_expr  [{;} STATUS [=] num_var ]]
```

As a command only:

```
 {str_lit          }
:{unquoted_str_lit }
```

Parameters

str_expr An operating system command, a UDC, a program file name, or a commandfile. For information on operating system commands, see the operating system reference manual and the *Console Operator's Guide*. If this parameter is not specified, HP Business BASIC/XL returns control to the operating system. You can then return to HP Business BASIC/XL by typing RESUME at the operating system prompt.

See the *MPE XL Intrinsics Manual* for information on what will be selected if commands, UDCs, programs or commandfiles exist with the same names.

If HP Business BASIC/XL is running from a batch job, *str_expr* must be specified.

If *str_expr* is specified, the SYSTEM command executes the CICOMMAND intrinsic, accessing the operating system only to execute the specified command, and return to HP Business BASIC/XL.

If any error or warning results from the command the JCW CIERROR will be changed to reflect the error or warning.

num_var If *str_expr* is specified, *num_var* returns the operating system error number. If *str_expr* is not specified, *num_var* returns the interpreter command error number, which is:

0 No error (if HP Business BASIC/XL is running interactively).

1 Error (if HP Business BASIC/XL is running from a batch job).

NOTE Just as on-line information on HP Business BASIC/XL is available by typing HELP in response to the ">" prompt, information on operation system commands is available by typing :HELP.

Examples

```
     10 SYSTEM                        !Returns to operating system.
     20 SYSTEM "LISTF"                !Issues the LISTF command.
     30 SYSTEM "SETMSG OFF"0           !Issues the SETMSG command.
     40 SYSTEM "LISTF"; STATUS=S      !Issues the LISTF command and returns status.
     50 SYSTEM "LISTF", STATUS S      !Same as line 40.
```

SYSTEMRUN

The SYSTEMRUN statement runs another program from HP Business BASIC/XL. The new program can be any program that the operating system can run or. HP Business BASIC/XL is suspended until the new program finishes, unless otherwise specified. This statement is primarily available for MPE/V compatibility; on MPE XL, the SYSTEM statement also can run other programs.

Syntax

As a statement or command:

```
                    [{,}                   ]
SYSTEMRUN str_expr  [{;} STATUS [=] num_var ]
```

As a command only:

```
     {str_lit          }
:RUN {unquoted_str_lit }
```

Parameters

str_expr The run string that the operating system recognizes for any program (HP Business BASIC/XL or not). For the syntax of this run string, see the appropriate operating system manual or type

```
:HELP RUN
```

.

The following parameters can be added to the operating system run string (run string parameters are separated by semicolons):

NOSUSP HP Business BASIC/XL is not suspended.

PRI= Priority of new program. This is one of: BS, CS, DS, or ES. BS is highest; ES is lowest. If the specified priority exceeds the highest priority that the system permits for the log-on account, then the priority is the highest possible below BS. The default priority is HP Business BASIC/XL's priority.

num_var Returns job control word (JCW) of called process.

For more information about these parameters, see the *MPE XL INTRINSICS Reference Manual* or type

```
:HELP RUN
```

Examples

```
100 SYSTEMRUN "Prog1"
200 SYSTEMRUN "Prog2;MAXDATA=31000"
300 SYSTEMRUN "Prog3;MAXDATA=20000;INFO=""Text3"""
400 SYSTEMRUN "Prog4;NOSUSP"; STATUS=S
500 SYSTEMRUN "Prog5;NOSUSP;PRI=DS", STATUS=S
600 SYSTEMRUN "Prog6;MAXDATA=10000;NOSUSP;PRI=DS",STATUS S
700 SYSTEMRUN "Prog7;NMSTACK=395000;XL=""XL.pub.tools,lib7.diag.sys"""
800 SYSTEMRUN Progname$+Run_options$+";unsat=debug"
```

You can execute a program on another terminal if that terminal is not in use (that is, no one is logged on). HP Business BASIC/XL requires additional settings beyond those which MPE XL requires in order for such programs to execute correctly.

To use a remote terminal, the specific device must be given a name with a file equation. To use HP Business BASIC/XL programs (or the interpreter) on that terminal, use a file equation similar to the following:

```
FILE <name>,NEW;DEV=<ldev>,ACC=INOUT;REC=-500;CCTL
```

You must specify the logical device number of the terminal, and the name must be a legal file name. If the access used is not INOUT, the system console will ask if the device can be used; this will then require an operator response.

The record size and CCTL specifications are for HP Business BASIC/XL. If these are not specified, neither input nor output can be guaranteed to be correct. If a record size of less than 500 bytes is used, any type of input, or a READ statement may cause the program to abort.

In order to run a program on another terminal, redirect $STDLIST and $STDIN to the equated file name.

```
10 SYSTEM "file term,new;dev=55;acc=inout;rec=-500;cctl"
20 SYSTEMRUN "myprog;stdlist=*term;stdin=*term"
30 SYSTEMRUN "myprog;stdlist=*term;stdin=*term;nosusp"
```

Line 20 will run the program MYPROG on the terminal whose logical device number is 55. This program will run correctly, allowing forms and keys to be used. The program above will wait for MYPROG to finish before executing line 30.

Line 30 also runs the MYPROG program. In this case, the NOSUSP in the systemrun command will allow the current program to continue without waiting for MYPROG to finish. Both programs will continue to run at the same time. However, NOSUSP can affect the handling of the HALT key (CONTROL Y).

THEN

The THEN statement is part of the IF THEN ELSE and IF THEN statements and constructs. Refer to the IF THEN statement for more information.

THREAD IS

The THREAD IS statement defines the thread list that is used by the SEARCH/SORT process. A thread list is a list of data sets in a database being searched. The thread list defines the hierarchy as well as the relationship between the data sets. In a THREAD IS statement, each data set is represented by a line label that refers to an IN DATASET statement of the corresponding data set.

Syntax

```
             [        [PATH num_expr   ]       ]
THREAD [IS] [line_id  [LINK identifier ] {,1;}] ...line_id
```

Parameters

line_id Line number or line label that identifies the line on which the IN DATASET statement of the dataset is defined.

num_expr A numeric expression that evaluates to an integer that represents the path to use when accessing a detail data set that is connected to it's master by multiple paths.

identifier A variable that holds a link value used when trying to access data in a detail data set that is not linked to any other data sets in the current thread list.

Examples

```
     100  Set1 :  IN DATASET "parts" USE A, B
     200  Set2 :  IN DATASET "customer" USE Comp$
     300          THREAD IS Set1, Set2
```

The THREAD IS statement on line 100 indicates that during a SEARCH or SORT, the data set "parts" is accessed first. The data for "parts" is retrieved and unpacked into variables A and B. Then the data set "customer" is read, its data retrieved and unpacked into the variable Comp$.

In going from one data set to the other while walking the thread list, you can optionally specify the path to be used in case there is more than one or the key value to be used in case it is from a detail to a master.

The THREAD IS statement must satisfy the following conditions:

- The thread list can be one to ten data sets long.

- The first data set can be either a master set or a detail set. However, the thread must not have two consecutive data sets of the same type. That is, a master set cannot follow a master set and a detail data set cannot follow a detail data set.

- You can optionally specify which path (PATH) to use when connecting two sets. If PATH is not specified, path 1 is assumed.

  Example :

  ```
            400    THREAD IS Set1, Set2 PATH 2, Set3, Set4
  ```

- In case there are no paths defined in the database between a detail set and a master set, THREAD allows you to define a temporary link by specifying a link variable (LINK) in the detail set. The link variable, if used, must be defined in the HP Business BASIC/XL program and must appear in the IN DATASET statement of the detail set. It must also be of the same data type as the key in the master.

Example :

```
500    THREAD IS Set1, Set2 LINK Var, Set3, Set4
```

- An error results if the specified path between the data sets does not exist and (for detail sets) no LINK is specified. Link cannot be used to connect a master to a detail.

- The THREAD statement is nonexecutable. Its validity will be checked at run time by the SEARCH statement or the SORT statement.

TINPUT

The TINPUT statement obtains a string of characters from an input device. The characters are echoed to the display as they are entered. If a string or numeric variable is included in the TINPUT statement, then the value of the string of characters entered is assigned to the variable. TINPUT options control the maximum amount of time allowed for input, the time required for input, the maximum number of characters that can be input, and the line feed generated subsequent to the statement execution. At least one option must be selected when using the TINPUT statement.

Syntax

TINPUT [*var*] [*separator*] *option_clause* [*separator option_clause*]...

```
                 {TIMEOUT [=] timeout_num_expr }
                 {ELAPSED [=] elapsed_num_var  }
option_clause  -> {CHARS [=] chars_num_expr     }
                 {NOLF                        }

             {WITH}
separator  -> {,   }
             {;   }
```

EACH individual *option_clause* can occur only once in a TINPUT statement.

Parameters

var — The numeric or string variable to which the input is assigned. A TINPUT statement without a *var* discards the input. Characters are assigned to the variable when you type RETURN. For string variables, note that no character, such as a comma or a double quote, is considered to be a data item separator or terminator within the input string. Leading and trailing blanks are also included in the string of characters assigned to a string variable.

For numeric variables, the input character string is interpreted as a numeric literal and is assigned to the numeric variable. In this case, a comma is a valid item separator or terminator. If using the European format, set by the Native Language Number, then a semicolon replaces the comma as a separator or terminator. Any leading, embedded and trailing blanks are suppressed. If an invalid character is entered, then an HP Business BASIC/XL error occurs.

timeout_num_ expr — Numeric expression for the maximum amount of time, in seconds, allowed by the user to enter input. The input time limit is determined as follows:

Value of *timeout_num_ expr*	Input Time Limit
Zero or less	Unlimited
In the range (0,255)	That number of seconds

	rounded to nearest second
Greater than 255	Set to 255 seconds

If the TIMEOUT option is not selected, then the input time limit is unlimited.

If input time is limited through the use of the TIMEOUT option, HP Business BASIC/XL transfers control to the next program statement when the time limit is exceeded without assigning a new value to the specified *var*.

elapsed_num_var A numeric variable to which the time, in seconds, taken to enter the input is returned. If the ELAPSED option is not selected, the elapsed time is not measured. If TIMEOUT is also specified, and a timeout occurs, *elapsed_num_var* is set to -256.

chars_num_expr A numeric expression that evaluates to the maximum number of characters that can be input. Typing this number of characters will cause the generation of an automatic carriage return and assignment of the value to the specified *str_var*. The program will then begin execution of the next statement in the program.

NOLF Suppresses the automatic line feed normally generated after pressing RETURN, subsequent to reaching the TIMEOUT limit specified, or after typing in that number of characters specified in the CHARS option.

Examples

The following examples show the TINPUT statement.

```
     10  TINPUT String_var1$, TIMEOUT Time_limit
     20  TINPUT String_var2$, ELAPSED Elapsed_time
     30  TINPUT String_var3$, CHARS Num_chars
     40  TINPUT String_var4$, NOLF
     50  TINPUT String_var5$, WITH TIMEOUT=10, ELAPSED=Elapsed_time
     60  TINPUT String_var6$, ELAPSED Elapsed time, CHARS 1, NOLF
     70  TINPUT Num_var WITH ELAPSED=Elapsed_time
     80  TINPUT Num_var, TIMEOUT Time_limit
     90  TINPUT CHARS 2
     100 TINPUT TIMEOUT 5
     110 TINPUT ELAPSED Elapsed_time
     120 TINPUT CHARS=1,NOLF
```

TOTALS

The TOTALS statement is a Report Writer statement that provides an easy means for automatic accumulation of numeric data. It provides totaling at the individual summary levels in a report.

A TOTALS statement can appear in a HEADER or TRAILER section only. There cannot be more than one TOTALS statement for each summary level. The TOTALS statement is not used if it is contained in a section with a level of zero, as the section is unused.

Syntax

```
                     [{,}          ]
TOTALS [ON] num_expr  [{;} num_expr ]...
```

Parameters

num_expr Any numeric expression can be totaled. There can be as many expressions as desired. When referring to a particular total, a *sequence* number is used. The first expression is sequence number 1, the second is number 2, and so on.

Examples

```
     100 TOTALS ON My_var, TRUNC(Sales), Quantity*100
```

The BEGIN REPORT statement makes the TOTALS statement busy and it remains busy until an END REPORT or STOP REPORT statement is executed. The TOTALS statement is used ONLY if contained in a HEADER or TRAILER section with a nonzero level number. There can only be one TOTALS statement per summary level. All accumulated totals are set to zero by BEGIN REPORT.

The TOTALS calculation occurs when a DETAIL LINE statement executes, but only when the *totals flag* of the DETAIL LINE is nonzero. The accumulated values are reset to zero for any summary level where a break occurs. This is done after the TRAILER sections are printed. After all break conditions are processed, the totals are accumulated.

The TOTALS statements are evaluated starting with GRAND TOTALS and working to level nine. For each statement, the expressions are evaluated from left to right. The value of each expression is added to previous totals.

All totals are stored in either REAL or DECIMAL data type, depending on the data type option in effect when the report started. However, the expressions themselves are evaluated as any other expression in HP Business BASIC/XL. This means that an individual expression may cause an overflow error without causing an overflow in the total.

TRAILER

The TRAILER statement allows you to define logical levels for separating and summarizing data printed in a report. The TRAILER section is used to print trailing data for a particular level in the report of which there are nine levels available.

In order to define a report level, there must be a TRAILER or HEADER statement in the report description. However, there can not be more than one TRAILER section for a single level within the report description. If no WITH or USING clause is present, the statement produces no output. However, other statements in this section might produce output.

Syntax

```
                     [               [LINES]]
TRAILER level_number [WITH num_lines [LINE ]]

[USING image  [; output_list ]]
```

Parameters

level_number A numeric expression with a value from 0 to 9. This defines the *summary* or *break* level for this trailer section. This number is used to create different summary levels for data, and to cause breaks in the report at appropriate times. A level of zero causes the entire section to be ignored.

num_lines The maximum number of lines expected to be needed by the section statement. This number reflects ALL output done by the section.

image An image string or a line reference to an IMAGE line.

output-list A list of output items, identical to the list used by the PRINT USING statement.

Examples

```
     100 TRAILER 1 WITH 3 LINES
     100 TRAILER Order(1) USING Hd_image;Who
```

If a report section is active when this statement is seen, the section is ended. An error occurs if this statement is executed directly when a report section is not active.

When BEGIN REPORT executes, the *level _number* of each TRAILER statement is evaluated and the statement is made BUSY. TRAILER sections with level numbers equal to zero are ignored. All of the level numbers are therefore fixed by BEGIN REPORT and the statements are made busy. All nonzero TRAILER levels must be distinct and within the range of one to nine. The levels do not have to be contiguous. A TRAILER statement can define a section without a corresponding HEADER section and vice versa.

TRAILER sections are executed when an automatic break occurs from BREAK IF or BREAK WHEN, or when the TRIGGER BREAK statement. TRAILER sections are printed in descending sequence by level number. See DETAIL LINE and EXECUTION FLOW for more details on automatic breaks.

The TRAILER sections are automatically executed when the report output stops normally. The trailers precede the printing of the report trailer and page trailer, printing in descending order.

A particular TRAILER section executes the TRAILER statement first. This causes the evaluation of the WITH clause first that may cause a page break, followed by the execution of the USING clause. Any additional statements in the TRAILER section execute after the TRAILER statement.

TRIGGER BREAK

The TRIGGER BREAK statement allows you to manually cause a Report Writer break to take place. This results in the printing of the TRAILER and HEADER sections.

The TRIGGER BREAK statement can not occur within a report description.

Syntax

TRIGGER BREAK *break_level*

Parameters

break_level A numeric expression in the range zero to nine. A level of zero has no effect. Other values cause a break to take place at the given level.

Examples

```
     100 TRIGGER BREAK 5
     100 IF Old_data <> New_data THEN TRIGGER BREAK N
```

The TRIGGER BREAK statement generates an error if a report is not active. If report output has not started, this statement starts the report, followed immediately by the break.

The *break_level* is evaluated after starting the report if this is necessary. Then all BREAK statements are evaluated in order to determine the new values for OLDCV and OLDCV$. Then the break actually occurs.

Executing a Report Writer Break

The execution of a summary level break involves several steps. Each step can execute several different sections of the report. The processing of the break is described below, in the order in which actions are taken.

A break can be caused either by DETAIL LINE, when a BREAK IF or BREAK WHEN condition is satisfied, or by the TRIGGER BREAK statement. In either case, the Report Writer function LASTBREAK is set to the lowest break that occurred.

1. Execute TRAILER sections from level nine down to the level contained by LASTBREAK. Each section first executes the TRAILER statement. The WITH clause is evaluated, and if the number of lines left on the page is less than the WITH value, a page break is automatically triggered. If the USING clause is present, it is then executed. Then the lines in the section are executed. The WITH clause accounts for all PRINT output generated by the section.

2. Update the OLDCV and OLDCV$ values. These values are not recalculated; the values found during the DETAIL LINE or TRIGGER BREAK are stored until this point, at which time the values are put into the OLDCV area. All OLDCV values from levels LASTBREAK to nine are updated.

3. Zero all TOTALS expressions from level LASTBREAK to nine.

4. Set NUMDETAIL to zero for levels LASTBREAK to nine.

5. Update NUMBREAK for levels LASTBREAK to nine. Also, the total number of breaks [NUMBREAK(0)] is incremented.

6. Finally, all HEADER sections are executed from LASTBREAK to nine. Each section first executes the HEADER statement. The WITH clause is evaluated, and if the number of lines left on the page is less than the WITH value, a page break is automatically triggered. If the USING clause is present, it is then executed. Then the lines in the section are executed. The WITH clause accounts for all PRINT output generated by the section.

Errors during a section can cause the break to stop early. However, most errors do not cause this to happen. Having fewer lines left on the page than the WITH value automatically triggers a page break.

TRIGGER PAGE BREAK

The TRIGGER PAGE BREAK statement allows you to do page breaks manually. This statement can occur anywhere except in the PAGE HEADER and PAGE TRAILER sections of a report description. When this statement is encountered, a page break executes immediately.

The suppress options of the TRIGGER PAGE BREAK statements allow for more flexibility than automatic page breaks. The use of these options may affect the number of lines available for printing on the page.

Syntax

```
                   [             {       [{,}        ]}]
                   [{,}          {HEADER [{;} TRAILER]}]
TRIGGER PAGE BREAK [{;} SUPPRESS {        [{,}       ]}]
                   [             {TRAILER [{;} HEADER]}]
```

Examples

The following examples show the TRIGGER PAGE BREAK statement.

```
100 TRIGGER PAGE BREAK
100 TRIGGER PAGE BREAK, SUPPRESS TRAILER
100 TRIGGER PAGE BREAK, SUPPRESS HEADER, TRAILER
```

This statement causes an error if no report is active. If report output has not begun, this statement starts the report.

When no suppress options are specified, this statement acts identically to an automatic page break; for example, one caused by a WITH clause on any Report Writer statement. The following actions are taken:

* Print blank lines up to the location where the PAGE TRAILER should begin.

* Execute the PAGE TRAILER section, if present. During this process, the number of lines left on the page is reset to the page trailer size.

* Print the blank lines at the top of the page.

* Execute the PAGE HEADER section, if present.

The SUPPRESS options of the TRIGGER PAGE BREAK statement alter the actions listed above. With these options, you can suppress the PAGE TRAILER on the current page and the PAGE HEADER at the top of the next page. These options apply only to the current page break. More permanent suppression can be done with the SUPPRESS HEADER and SUPPRESS TRAILER statements.

If the TRIGGER PAGE BREAK statements specify that TRAILER is to be suppressed, then the PAGE TRAILER section is not executed. Instead, blank lines are printed for the PAGE TRAILER. All other steps apply as stated above.

When the SUPPRESS HEADER option is encountered, all steps take place as indicated above, except for the execution of the PAGE HEADER section. The top margin specified in the PAGE LENGTH statement is not suppressed. Since the PAGE HEADER is not printed, there are more lines available on the page.

As an example, consider a report description such as the following, where a TRIGGER PAGE BREAK occurs at the end of the REPORT HEADER:

```
100 REPORT HEADER
110   PAGE LENGTH 60,0,0
.
.
.
200   TRIGGER PAGE BREAK, SUPPRESS TRAILER
210 PAGE HEADER WITH 3 LINES USING Ph_1
.
.
.
500 END REPORT DESCRIPTION
```

When this report starts printing, the REPORT HEADER section executes first. After the desired title is printed (lines 110 to 199), the report executes a page break. Suppressing the page trailer on this first page, causes a title page to print at the start of the report.

NOTE Normally the PAGE HEADER is printed immediately after the REPORT HEADER. However, when the TRIGGER PAGE BREAK executes in the report header section, the PAGE HEADER executes at the top of the second page. The Report Writer does not put out a second page header.

UNLOCK

The UNLOCK statement relinquishes the exclusive access that the LOCK statement requested for a file.

Syntax

UNLOCK *#fnum*

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer.

Examples

```
     100 CREATE "File1",FILESIZE=1200
     200 ASSIGN "File1" TO #10         !Assigns file to #10.
     300 LOCK #10                      !File is locked.
     400 PRINT #10; A,B,C
     500 UNLOCK #10                    !File is unlocked after printing.
     999 END
```

For more information, see the LOCK statement.

UNPACK

The UNPACK statement assigns the values of individual data items contained in one scalar string variable to one or more HP Business BASIC/XL variables. The correspondence of the values in the scalar string variable is determined by the order of the variables listed in the referenced PACKFMT statement.

Syntax

UNPACK USING *line_id*; *str_var*

Parameters

line_id Specifies the program line of the appropriate PACKFMT statement that specifies the variables to be unpacked and the current format in which they are packed within *str_var*.

str_var Scalar string variable from which variables are to be unpacked.

Examples

The following example shows the use of the UNPACK statement. Lines 120 and 130 contain the PACKFMT statements that the UNPACK statements use. Lines 210 and 220 PACK the data using those PACKFMT statements, and lines 235 and 236 UNPACK the data, and assigns them to the referenced variables.

```
     100 INTEGER Number, Times(4)
     105 INTEGER Num, N1(4)
     110 DIM String$[10], S1$[10], S2$[10]
     115 DIM A$[10], P1$[60], P2$[60]
     120 Pack1: PACKFMT Number,String$,A$,Times(*)
     130 Pack2: PACKFMT Times,SKIP 2,String$,SKIP 1,Number,SKIP 1,A$[3;5]
     140 Number=1234
     150 Times(1)=65
     160 Times(2)=73
     170 Times(3)=42
     180 Times(4)=90
     190 String$="abcd"
     200 A$="efghi"
     210 PACK USING Pack1; P1$
     220 PACK USING Pack2; P2$
     230 Pack3: PACKFMT Num,S1$,S2$,N1(*)
     235 UNPACK USING Pack3;P1$
     236 PRINT Num,S1$,S2$,(FOR I=1 TO 4, N1(I))
     240 Pack4: PACKFMT N1,SKIP 2,S1$,SKIP 1,Num,SKIP 1,S2$[3;5]
     245 UNPACK USING Pack4; P2$
     246 PRINT N1(1),N1(2),N1(3),N1(4),S1$,Num,S2$
     250 END
```

UNTIL

The UNTIL statement is part of the REPEAT UNTIL construct. Refer to the REPEAT statement for more information.

UPDATE

The UPDATE statement assigns a value to the current datum of a specified BASIC DATA file, if the assignment is legal. The UPDATE statement cannot change the type of the datum, and it cannot change the length of a string datum.

Syntax

UPDATE *#fnum*; *expr*

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the BASIC DATA file. It is a numeric expression that evaluates to a positive short integer.

expr Its value is assigned to the current datum or the datum indicated by the file's datum pointer if the assignment is legal.

If the new value is not of the same type as the old value, the UPDATE statement converts the new value to the old type. If this is impossible, an error occurs.

If *expr* is a string, and it is shorter than the string that it replaces, it is blank-filled on the right. If *expr* is a string that is longer than the string that it replaces, it is truncated on the right.

Examples

The following statements show the update statement.

```
     10 UPDATE #1; 1234       !Updates #1
     20 UPDATE #2; "CAT"      !Updates #2
     30 UPDATE #3; SIN(X+35)  !Updates #3 with the results of a function
     40 UPDATE #3; LWC$("JOHN " + "DOE")   !Updates #4 with "john doe"
```

WAIT

The WAIT statement delays program execution.

Syntax

WAIT [*num_expr*]

Parameters

num_expr Number of seconds that the program is delayed (can be a REAL value). If *num_expr* is not specified, the program waits for a user interrupt, a CONTROL Y.

Examples

```
     10 WAIT 120           !120 seconds (2 minutes)
     15 WAIT 0.5           !0.5 seconds
     20 WAIT Wait_time     !Number of seconds specified by Wait_time
     25 WAIT Wait_time+60  !One minute more than specified by Wait_time
     30 WAIT               !Until user interrupt (control-Y)
```

WARNINGS OFF

The WARNINGS OFF statement suppresses the warning messages that HP Business BASIC/XL normally displays. Use WARNINGS ON to return to the default.

Syntax

WARNINGS OFF

WARNINGS ON

The WARNINGS ON statement allows HP Business BASIC/XL to display warning messages, as it does by default. This statement is used to deactivate a WARNINGS OFF statement.

WARNINGS ON

WHILE

The WHILE and END WHILE statements define a loop that repeats until the numeric expression in the WHILE statement evaluates to FALSE (zero).

Syntax

WHILE *num_expr* [DO] [*stmts*]...END WHILE

Parameters

num_expr Considered FALSE if it evaluates to zero; TRUE otherwise. If it is TRUE, the statement or statements in the loop are executed. If it is FALSE, the statement following ENDWHILE is executed. Following execution of the statements in the loop body, *num_expr* is again evaluated to determine whether the loop body is executed again.

stmts Program lines that are executed if *num_expr* is TRUE. These statement constitute the loop body.

Examples

```
     10 I=50       !Let I be the first number to be printed, 50
     20 WHILE I<>0 !If I<>0, execute loop (lines 30 and 40)
     30   PRINT I  !Print current number, I
     40   I=I-1    !Let I be the next number to be printed
     50 END WHILE  !Return to line 20
     99 END
```

WHILE constructs can be nested.

```
     100 Num_rows=3             !Number of rows in matrix M
     110 Num_cols=4             !Number of columns in matrix M
     120 Row=1                  !Let Row be first row to be printed
     130 WHILE Num_rows-Row     !BEGIN OUTER WHILE LOOP
     140   Col=1                !Let Col be first column to be printed
     150   WHILE Num_cols-Col   !BEGIN INNER WHILE LOOP
     160     PRINT M(Row,Col)   !Print one matrix element
     165     Col=Col+1          !Let Col be next column to be printed
     170   END WHILE            !END INNER WHILE LOOP
     180   Row=Row+1            !Let Row be next row to be printed
     190 END WHILE              !END OUTER WHILE LOOP
     999 END
```

Entering a WHILE loop from a statement other than the WHILE statement is considered to be a bad programming practice, and is not recommended. Use of a GOSUB or CALL statement from within a WHILE loop can be useful.

```
     100 PRINT "Sum of the odd numbers 1 to 150 is: "
     110 N=1
     120 Sum=0
     130 WHILE 150-N                  !Begin loop
     140   IF N MOD 2 THEN CALL Odd(Sum,N)
     150   N=N+1
     160 ENDWHILE                     !End loop
     170 PRINT Sum
     180 END
     190 !
     200 SUB Odd(Sum,N)               !Return to loop
     210   Sum=Sum+N
     220 SUBEND                       !Return to next line following CALL
```

WORKFILE IS

The WORKFILE IS statement identifies the file, called a workfile, that holds the record pointers of the selected records in the database. This statement is global in nature and deactivates any previously defined workfile. The file designated as the workfile must be open. The workfile itself must be a binary file with record size (in words) equal to twice the number of data sets in the THREAD IS statement. It must be defined before the SORT or SEARCH statement is executed. Since the workfile is, by definition, a user-defined file, it is subject to the same rules and restrictions that apply to HP Business BASIC/XL files. In addition, open it with both read and write capability.

Syntax

WORKFILE IS *#fnum*

Parameters

fnum A numeric expression that evaluates to a positive short integer greater than zero. The value is the same as that used to open the workfile.

Examples

```
     100    ASSIGN #1 TO "filex"   !File #1 is filex
     200    WORKFILE IS #1         !filex is the workfile
```

WRITE FORM

The WRITE FORM can be used to display the value of an HP Business BASIC/XL variable in a field of a VPLUS form, position the cursor, or write to the message window of the VPLUS form. It is possible to do any combination of these operations in a single statement.

Syntax

WRITE [TO] FORM

```
[          [{,}            ]]
[form_item  [{;} form_item...]]

[{,}                       ]
[{;} CURSOR [=] cursor_expr ]
[{,}                     ]
[{;} MSG [=] message_expr ] form_item ->

{form_element  | for_clause  | skip_clause }
```

Parameters

form_element One of the following:

num_var
str_var $
array_name ([*[,*]...])
str_array_name $([*[,*]...])

The last format above has one asterisk per dimension or no asterisks. No asterisks specifies any number of dimensions. Either format is legal, but the format with no asterisks is not compilable. Substrings are also allowed.

for_clause (FOR *num_var* =*num_expr1* TO *num_expr2* [STEP *num_expr3*], *form_item* [, *form_item*]...)

A *for_clause* is useful for reading array elements. Refer to the INPUT statement in this chapter for more information.

skip_clause SKIP *skip_expr*

A *skip_clause* is used to skip one or more fields in the form to avoid the necessity of displaying values for them. The *skip_expr* is a numeric expression that evaluates to the number of fields to skip.

cursor_expr Either a numeric or a string expression. If a string expression is used, it must be the name of a field in the form that is active.

If a numeric expression is used, it is a field number. The fields corresponding to the value of the numeric expressions are:

0:	for input field on the form.
positive value:	field number according to the form.
negative value:	field number according to the terminal.

message_expr A string expression. Its value is written to the message window located at the bottom of each VPLUS form.

The WRITE FORM statement writes an entire screen of information at once. The value of each field is obtained from a single variable or array element. The value of the first *form_item* is written to the first field on the form, the value of the second *form_item* is written to the second field on the form, etc. The value of each data item specified in a *for_clause* is written to a single field. The value of each element of the array specified by the *array_name* (*) notation is also written to a single field. Use of the option SKIP 3 allows you to write a value in the fourth field of the form without having to write information in the preceding three.

The clauses are evaluated in the following order:

1. Any message that is to be written to the message window.

2. Any specified final cursor positioning takes place.

3. Any data that is to be written to the fields.

It is important to understand that cursor positioning is of value only for a subsequent READ FORM; using a WRITE FORM to position the cursor for a subsequent WRITE FORM does not produce the expected results unless you position the cursor to the first field. Use the SKIP clause to begin writing to a field other than the first field.

If no VPLUS form is active, executing a WRITE FORM statement causes a run-time error.

Examples

The following examples show the use of the WRITE FORM statement.

```
     400 WRITE FORM Num_var
     410 WRITE FORM A,B;C$
     420 WRITE FORM A,B;C$
     430 WRITE FORM A,SKIP 3,B
     440 WRITE FORM ;MSG="ERROR: BAD NAME";CURSOR=5
     450 WRITE FORM ;CURSOR="Emp-name"
     460 WRITE FORM A;SKIP 3,B;MSG="ERROR: BAD NAME";CURSOR=5
```

Chapter 5   Functions

Introduction

HP Business BASIC/XL has a set of predefined standard functions. These functions do not need to be defined to be called, nor is a calling statement necessary. They can be treated like any expression. For example, in the program below, Bnum and Cnum are assigned the return value from the ABS function.

```
     10  Anum = -10
     20  Bnum = ABS(Anum)                !Absolute value function
     30  Cnum = ABS(3)                   !Absolute value function
     40  PRINT Anum,Bnum,Cnum,ABS(-24)   !Prints -10,10,3,24
```

The return value for each function has a specific data type. You can, however, assign the return value to a variable of a different type, and HP Business BASIC/XL will convert the return value to the type of the variable that the function is assigned to.

ABS

The ABS function returns the absolute value of a number.

Syntax

ABS(*n*)

Parameters

n   The number whose absolute value is to be returned. *n* can be of any numeric type.

The return variable is the same type as *n*, except for INTEGER and SHORT INTEGER types. INTEGER variables return a REAL number, and SHORT INTEGER variables return an INTEGER.

Examples

```
     10  Abs = ABS(-10)     !Abs is 10
     20  Abs = ABS(10)      !Abs is 10
```

ACS

The ACS function returns the principal value of the arc cosine of a number. The argument value will be in the range of [-1, 1]. The result can be expressed in angular units of degrees, grads, or radians.

Syntax

ACS(*n*)

Parameters

n   The number to be evaluated. *n* is a REAL number.

The ACS function returns a REAL number.

Examples

```
     10  A = ACS(0.5)     !A = 60.00 (degrees)
     20  B = ACS(0.5)     !B = 66.67 (grads)
     30  C = ACS(0.5)     !C = 1.05  (radians)
```

ASN

The ASN function returns the principal value of the arc sine of a number. The argument value is in the range [-1, 1]. The result can be expressed in angular units of degrees, grads or radians.

Syntax

ASN(*n*)

Parameters

n The number to be evaluated. *n* is a REAL number in the range of [-1, 1].

The ASN function returns a REAL number.

Examples

```
     10 A = ASN(0.6)   !A = 36.87(Degrees)
     20 B = ASN(0.6)   !B = 40.97(Grads)
     30 C = ASN(0.6)   !C =   .64(Radians)
```

ATN

The ATN function returns the principal value of the arc tangent of a number. The result can be expressed in angular units of degrees, grads, or radians.

Syntax

ATN(*n*)

Parameters

n The number to be evaluated. *n* is a REAL number.

The ATN function returns a REAL number.

Examples

```
     10 A = ATN(0.7)   !A = 34.99 (Degrees)
     20 B = ATN(0.7)   !B = 38.88 (Grads)
     30 C = ATN(0.7)   !C = .61 (Radians)
```

AVG

The AVG function is a Report Writer function that returns the average value of a Report Writer total. It returns the value of the TOTAL(Level, Sequence)/NUMDETAIL(Level) functions. See those functions for further detail.

Syntax

AVG(*level*,*sequence*)

Parameters

level The summary level number. It must be in the range [0, 9].

sequence Indicates which expression in the given TOTALS statement should be returned. The first expression is sequence number one. An error occurs if the sequence number is less than one or greater than the number of expressions in the totals statement.

Example

The following program segment calls the AVG function.

```
     100 Level1=3
     120 Sequence1=2
     130 Average=AVG(Level1,Sequence1)
```

BINAND

The BINAND function returns the binary AND for two numbers. The result of this function is a short integer that contains a one in each bit for which the same bit in both of the arguments is a one.

It returns a short integer *R* such that:

R (*n*) = N1(*n*) AND N2(*n*)

for all *n* in [0, 15] where N1(*n*) and N2(*n*) represent the value of bit *n* of each expression and *R* represents the short integer result of BINAND.

Syntax

BINAND(N1,N2)

Parameters

N1 Binary representation of a numeric expression. *N1* is a short integer.

N2 Binary representation of a numeric expression. *N2* is a short integer.

Examples

The example below shows a layout of each bit of the arguments, and the resulting bit layout of the result.

```
Bit Number:   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
N1=           0  1  1  0  0  1  0  1  0  0  0  1  1  1  0  1
N2=           0  1  0  0  1  1  1  0  0  1  0  0  1  1  1  0

BINAND(N1,N2)=0  1  0  0  0  1  0  0  0  0  0  0  1  1  0  0
```

BINCMP

The BINCMP function returns the binary complement for all *R* such that

R (*n*) = NOT N1(*n*)

for all *n* in [0, 15] where N1(*n*) represents the value of bit *n* in N1 and R represents the short integer result of the function. HP Business BASIC/XL stores a negative number as the two's complement of its absolute value. The two's complement of a number is its complement or the results of the BINCMP function, plus one.

Syntax

BINCMP(*N1*)

Parameters

N1 Binary representation of a numeric expression. This is a short integer.

Examples

The example below shows the bit layout for the argument, N1. It shows the bit layout for the result of the BINCOMP function.

```
Bit Number:   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
N1=           0  1  0  0  1  1  1  0  0  1  0  0  1  1  1  0

BINCMP(N1)=   1  0  1  1  0  0  0  1  1  0  1  1  0  0  0  1
```

BINOR

The BINOR function returns the Binary OR for all *R* such that

R(*n*)=N1(*n*) OR N2(*n*)

for all *n* in [0, 15] where N1(*n*) and N2(*n*) represent the value of bit *n* in each expression and *R* represents the short integer result of the function. That is, if a particular bit in either argument contains a one, the resulting bit will be one. If both arguments have a zero in a particular bit, the result will have a zero in that bit.

Syntax

BINOR(*N1*,*N2*)

Parameters

N1 Binary representation of the value of a numeric expression. This is a short integer.

N2 Binary representation of the value of a numeric expression. This is a short integer.

Examples

The example below shows the bit layout for the BINOR function. It shows each bit of both arguments, and the result of the BINOR function.

Bit Number:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
N1=	0	1	1	0	0	1	0	1	0	0	0	1	1	1	0	1
N2=	0	1	0	0	1	1	1	0	0	1	0	0	1	1	1	0
BINOR(N1,N2)=	0	1	1	0	1	1	1	1	0	1	0	1	1	1	1	1

BINXOR

The BINXOR function returns the Binary Exclusive OR for all *R* such that

R (*n*) = N1(*n*) XOR N2(*n*)

for all *n* in [0, 15] where N1(*n*) and N2(*n*) represent the value of bit *n* in each expression and *R* represents the short integer result of the function. That is, if a particular bit of both arguments have the same contents (either zero or one) the same bit in the result will contain a zero. If a particular bit in both arguments do not have the same contents, the same bit in the result will contain a one.

Syntax

BINXOR(*N1*,*N2*)

Parameters

N1 Binary representation of a numeric expression. This is a short integer.

N2 Binary representation of a numeric expression. This is a short integer.

Examples

The example below shows the bit layout for the BINXOR function. It shows the values in each bit of the arguments, and the values in each bit of the result.

Bit Number:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
N1=	0	1	1	0	0	1	0	1	0	0	0	1	1	1	0	1
N2=	0	1	0	0	1	1	1	0	0	1	0	0	1	1	1	0
BINXOR(N1,N2)=	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	1

BITLR

The BITLR function returns the value of a particular bit of an expression, where 0 is the Most Significant (or leftmost) bit. The result is a SHORT INTEGER.

Syntax

BITLR(*N1*,*N2*)

Parameters

N1 Binary representation of a numeric expression. This is a SHORT INTEGER. This is the number containing the bit to be extracted.

N2 Binary representation of a numeric expression. This is a SHORT INTEGER. This is the number of the bit to be extracted from *N1*.

Examples

The example below shows a bit layout for N1. It shows the results of the BITLR function for several values of the second parameter (*N2*).

```
     Bit Number:   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
     N1=           0  1  1  0  0  1  0  1  0  0  0  1  1  1  0  1

     BITLR(N1,15)=1, BITLR(N1,11)=1, BITLR(N1,8)=0, BITLR(N1,3)=0
```

BITRL

The BITRL function returns the value of a particular bit of an expression, where 15 is the Most Significant (or leftmost) bit. The result is a SHORT INTEGER.

Syntax

BITRL(*N1*,*N2*)

Parameters

N1 Binary representation of a numeric expression. This is a SHORT INTEGER. This is the number containing the bit to be extracted.

N2 Binary representation of a numeric expression. This is a SHORT INTEGER. This is the number of the bit to be extracted from *N1*.

Examples

The example below shows the bit layout for N1. It shows the result of the BITRL functions for several values of the second parameter (*N2*).

```
     Bit Number:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
     N1=           0  1  1  0  0  1  0  1  0  0  0  1  1  1  0  1

     BITRL(N1,15)=0, BITRL(N1,11)=0, BITRL(N1,8)=1, BITRL(N1,3)=1
```

BRK

The BRK function returns the status that BREAK and CONTROL Y (halt) had before the BRK function was called. It can also change the status of these, depending on the value of the argument passed. BRK is a Boolean function that returns the value TRUE (one) or FALSE (zero).

Syntax

BRK (*num_expr*)

Parameters

num_expr This value determines whether BRK changes the status of BREAK and CONTROL Y, as follows:

num_expr	Status of BREAK and CONTROL Y
Negative	Does not change status
Zero	Disables both
Positive	Enables both

The BRK function returns:

TRUE (one) If BREAK and CONTROL Y were enabled before the BRK function was called.

FALSE (zero) If BREAK and CONTROL Y were disabled before the BRK function was called.

When BREAK is enabled, pressing BREAK causes the operating system to suspend HP Business BASIC/XL. The operating system command :RESUME restarts HP Business BASIC/XL. If CONTROL Y is enabled and pressed and a program is being executed, a message is printed indicating that HALT was pressed and control is returned to the HP Business BASIC/XL interpreter. If CONTROL Y is pressed while in the HP Business BASIC/XL interpreter, only the message is printed.

When BREAK and CONTROLY are disabled, pressing either has no result.

Examples

```
     10 Was_enabled=BRK(-1)              !BRK does not change status
     11 Was_enabled=BRK((10+10)-(10*10))  !BRK does not change status
     20 Was_enabled=BRK(0)               !Disables BREAK and CONTROL Y
     21 Was_enabled=BRK((X-Y)-(X+(-Y)))   !Disables BREAK and CONTROL Y
     30 Was_enabled=BRK(1)               !Enables BREAK and CONTROL Y
     31 Was_enabled=BRK(ABS(X))          !Enables BREAK and CONTROL Y
                                         !(if X != 0)
     40 Was_enabled=BRK(X)               !Action depends on value of X
     50 IF BRK(X) THEN GOTO 100          !Action depends on value of X
```

BUFTYP Function

The BUFTYP function returns the number that represents the type of the next item in the input buffer. See the INPUT statement in chapter 4 for an explanation of the input buffer. The BUFTYP function returns the same numeric values representing HP Business BASIC/XL data types returned by the DATATYP and TYP functions (see Table 5-1).

Table 5-1. Numbers Representing Input Data Types

BUFTYPE Result	**Type of Next Item in DATA Statement or Input Buffer**
1	DECIMAL
2	Entire string
4	End-of-record (EOR) mark
5	SHORT INTEGER
11	INTEGER
13	REAL

Syntax

BUFTYP

The BUFTYP function determines the type of a numeric datum by its format, whether it contains a decimal point or is expressed in scientific notation, its value, and the default numeric type.

Table 5-2 explains how BUFTYP determines the type of a numeric datum.

Table 5-2. Type Assignment by BUFTYP Function

Range	Without Decimal Point and Not Expressed in Scientific Notation	With Decimal Point or Expressed in Scientific Notation
[-32768, 32767]	SHORT INTEGER	REAL
[-2147483648, 2147483647]	INTEGER	REAL
Outside integer ranges but inside real range	REAL	REAL
Outside real ranges	DECIMAL	DECIMAL

Examples

```
     10  A=BUFTYP   !After this call, A will contain the data type of the
     20             !next item in the input buffer.
```

CCODE

The CCODE function returns the condition code set by the last called MPE XL intrinsic. The results are:

Intrinsic condition code:	CCODE Returns:	Meaning:
CCG	0	A special condition occurred but the request may have been granted.
CCL	1	An error has occurred and the request was not granted.
CCE	2	Request has been granted.

Refer to the *MPE XL Intrinsics Reference Manual* for more information.

Syntax

CCODE

Examples

The example below calls an intrinsic (Findjcw), and then uses the CCODE function to make sure the intrinsic was executed successfully.

```
     10 INTRINSIC Findjcw
     20 CAll Findjcw
     30 IF CCODE < 2 THEN GOSUB 300
     .
     .
     .
```

CEIL

The CEIL function returns the smallest integral number that is greater than or equal to the specified number. This function returns a value that is the same type as the argument.

Syntax

CEIL(*n*)

Parameters

n The number to be evaluated. This can be of any numeric type.

Examples

```
     10  A = CEIL(3.7)     !A = 4
     20  B = CEIL(-3.7)    !B = -3
```

CHR$

The CHR$ function returns the single ASCII character associated with a number.

Syntax

CHR$(*N*)

Parameters

N The numeric expression to be evaluated. This must evaluate to a value within the range of an HP Business BASIC/XL integer. If *N* is greater than 256, then HP Business BASIC/XL performs (*N* MOD 256), and the CHR$ returns the ASCII character of that result.

Examples

```
     10  A$ = CHR$(65)   !A$ = A
     20  B$ = CHR$(321)  !B$ = A
```

CLOCK

The CLOCK function returns the current value of the system clock in seconds. This is an INTEGER. On the HP 3000, the value of the system clock is the number of seconds since the time 00:00:00 on January 1, 1980.

Syntax

CLOCK

Since the CLOCK function is precise to the nearest second, two calls to CLOCK within the same second may return equal values.

Examples

```
     100 Start=CLOCK
             .
             .
             .
     900 Stop=CLOCK
     910 PRINT "Elapsed time:  "; Stop - Start
     999 END
```

COL

The COL function returns the number of columns in an array as it is currently dimensioned. If it is a vector (a one dimensional array), the number of columns is one. Otherwise, the number of columns is the size of the rightmost dimension. The result is an integer value by default.

Syntax

COL(*array*)

Parameters

array Structured collection of variables of the same type. The structure is determined when the array is declared. String variables names are suffixed with a "$".

Examples

OPTION BASE 1 is assumed.

The following shows several examples of the result of the COL function on arrays A, B,C,D,E, and F.

```
     A(2,2): 1 2   B(2,4):  1 2 3 4   C(4,3,2): 1 2  0 4   0 0  1 2
             4 5            5 6 7 8             5 1  1 0   4 5  0 0
                                                2 0  3 2   1 2  0 1

     D(3,3): 1 0 1  E(2,2): 8 3       F(5): 5 4 3 2 1
             3 5 7          4 7
             9 0 9

     COL(A) = 2
     COL(B) = 4
     COL(C) = 2
     COL(D) = 3
     COL(E) = 2
     COL(F) = 1
```

COMPRESS$

The COMPRESS$ function returns a copy of string in which a single blank space replaces each run of blank spaces.

Syntax

COMPRESS$(*S* $)

Parameters

S$ A string expression to be compressed.

Examples

```
     10  A$ = COMPRESS$("c  a   t")    !A$ = "c a t"
```

COS

The COS function returns the cosine of a number. The result is a real number. The argument and result can be expressed in angular units of degrees, grads, or radians.

Syntax

COS(*n*)

Parameters

n The number that is to be evaluated. This is a REAL number.

Examples

```
     10  A = COS(45)     !A = .71 (Degrees)
     20  B = COS(45)     !B = .76 (Grads)
     30  C = COS(45)     !C = .53 (Radians)
```

CPOS

The CPOS function returns the column position of the cursor in display memory. For terminals that have a display 80 columns wide, a value in the range 1..80 is returned. A return value of 1 corresponds to the leftmost column and a return value of 80 corresponds to the rightmost column. The program fragment:

```
        100 CURSOR (,45)  ! Position cursor to column 45
        120 PRINT CPOS    ! Prints position of the cursor in display memory
```

prints the number 45. CPOS determines the cursor position by reading it from the terminal. Therefore, typing on the keyboard while a CPOS statement is executing may cause an error.

Syntax

CPOS

Examples

```
     200 PRINT CPOS
```

CPU

If called from within either an interpreted or compiled program, the CPU function returns the number of CPU seconds elapsed since the beginning of program execution.

If typed directly in response to the interpreter prompt, the CPU function returns the total number of CPU seconds required for the execution of the last previous program to execute in the interpreter.

The result of this function is a REAL number.

Syntax

CPU

Examples

```
     100 Cpu_time = CPU
     110 PRINT "CPU time is: " ; CPU
     >
```

The above example returns a REAL value that contains the elapsed CPU time.

CSUM

The CSUM function returns an array that contains the sum of the elements of each column of an array. Both arrays must be of the same type. The result has the format

MAT *num_array1* = CSUM(*num_array2*)

where element *i* of *num_array1* is the sum of the elements in column *i* in *num_array2*. *num_array2* is dimensioned (*m*,*n*) and *num_array1* is dimensioned (*n*). The data type of the resulting array is the same as that of the argument.

The CSUM function is used in the MAT = statement, with two dimensional arrays.

Syntax

CSUM(*array*)

Parameters

array Structured collection of variables of the same type. The structure is determined when the array is declared. This array can be of any type.

Examples

```
     10  DIM A(4)
     20  DIM B(3,4)
     .
     .
     .
     80 MAT A = CSUM(B)
```

IF B is

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

then A is

```
     12 13 16 9
```

CURKEY

The CURKEY function returns the integer value of the last branch-during-input key pressed. If the value returned is 0, then no branch-during-input keys have been pressed during the execution of the program. The value returned representing a key is in the range [1, 8].

Syntax

CURKEY

Example

```
     100 PRINT CURKEY               !Prints the value of the last branch-during-input
     105                             !key pressed.
     110 IF CURKEY > 0 THEN ENABLE  !If a branch-during-input key was pressed,
     115                             !then the branch occurs.
```

DAT3000$

The DAT3000$ function returns a substring of the date string returned by the HP 3000 DATELINE intrinsic. On the HP 3000 under MPE XL, the date string is a string of 27 characters with the following format:

```
     MON, MAR  3, 1986, 12:44 PM
```

Syntax

DAT3000$ (*num_expr1*, *num_expr2*)

Parameters

num_expr1 A numeric expression that evaluates to the position of the first character of the date string.

num_expr2 A numeric expression that evaluates to the position of the last desired character of the date string.

Both *num_expr1* and *num_expr2* must evaluate to a value in the range of [1, 27], inclusive. The value of *num_expr1* must be less than or equal to the value of *num_expr2*. If any of these conditions is violated an error occurs.

Examples

```
     10 A$=DAT3000$(1,17)
     15 PRINT "12345678901234567"
     20 PRINT A$
     99 END
```

The above program prints:

```
     12345678901234567
     MON, MAR  3, 1986
```

```
     10 A$=DAT3000$(1,10)
     15 PRINT "1234567890"
     20 PRINT A$
     99 END
```

The previous program prints:

```
     1234567890
     MON, MAR
```

DATATYP

The DATATYP function returns a number that represents the data type of the next value to be read from a DATA statement (see Table 5-3).

Table 5-3. Numbers Representing Input Data Types

DATATYP Result	**Type of Next Item in DATA Statement**
1	DECIMAL
2	Entire string
4	End-of-record (EOR) mark
5	SHORT INTEGER
11	INTEGER
13	REAL

Syntax

DATATYP

Examples

```
     10 READ A,B$
     20 PRINT DATATYP
     30 DATA 1.0,"hello",3
```

Line 20 above will print 5. Since the first two items have been read, the value 3 is the next item in the DATA statement.

DATE$

If the system date has been set, the DATE$ function returns an eight-character string that contains the current system date.

If the system date has not been set, the DATE$ function returns the null string.

Syntax

DATE$ [(*num_expr*)]

Parameters

num_expr   Determines date format as shown in Table 5-4.

Table 5-4. Effect of DATE$ Function Parameter

num_expr	**Date format**
Not specified	The default specified by HP Business BASIC/XL Configuration Utility (see appendix C). If HP Business BASIC/XL Configuration Utility has not set default, it is U. S. format. This is compatible with the HP250.

Integer zero	U. S. format: *mm/dd/yy*.
Integer one	European format: *dd.mm.yy*.
Other	Error.

Examples

```
     10 DIM Us_date$[8], Eur_date$[8], Default_date$[8]
     20 Us_date$=DATE$(0)
     30 Eur_date$=DATE$(1)
     40 Default_date$=DATE$
     50 PRINT Us_date$
     60 PRINT Eur_date$
     70 PRINT Default_date$
     99 END
```

If the system date is June 12, 1984, the above program prints:

```
     06/12/84
     12.06.84
     06/12/84
```

DEBLANK$

The DEBLANK$ function returns a copy of a string without blanks.

Syntax

DEBLANK$(*S* $)

Parameters

S $ The string expression to be deblanked.

Examples

```
     10 A$ = DEBLANK("c a t")    !A$ = "cat"
```

DECIMAL

The DECIMAL function converts a number to DECIMAL format.

Syntax

DECIMAL(*n*)

Parameters

n   The number that is to be converted to decimal. This can be any numeric data type.

Examples

```
     10 Dec_val = DECIMAL(3)    !Dec_val = 3.00
```

DET

The DET function returns the determinant of a square numeric matrix. If the matrix is DECIMAL or SHORT DECIMAL, HP Business BASIC/XL converts it to REAL before computing the determinant. The result is of the default numeric type.

Syntax

DET(*num_sq_matrix*)

Parameters

num_sq_matrix   A two dimensional numeric array with the same number of rows as columns.

Examples

OPTION BASE 1 is assumed.

```
     A(2,2): 1 2   D(3,3): 1 0 1  E(2,2): 8 3
             4 5           3 5 7          4 7
                           9 0 9

     DET(A) = -3
     DET(D) = 0
     DET(E) = 44
```

DOT

The DOT function returns the dot product, or inner product, of two vectors. The elements of the two vectors must be of the same type. If they are short integer arrays, the result is an integer; otherwise, the result is the same type. Intermediate calculations for computing the DOT product of two short decimal type vectors are performed after converting each of the appropriate elements to decimal type values. Therefore, in compiled programs, short decimal overflow is reported as decimal overflow. The result is of the default numeric type.

Syntax

DOT(*num_vector1*,*num_vector2*)

Parameters

num_vector1 A numeric one dimensional array.

num_vector2 A numeric one dimensional array.

Examples

OPTION BASE 1 is assumed.

```
     A(4) = 1 2 3 4
     B(4) = 2 2 2 2

     DOT (A,B) = 1*2+2*2+3*2+4*2= 21
```

DROUND

The DROUND function rounds a number to a specified number of digits. The result is of type DECIMAL.

Syntax

DROUND(*n1*,*n2*)

Parameters

n1 The number to be rounded. Although this can be of any numeric type, it is converted to DECIMAL.

n2 The number of digits that *n1* is to be rounded to.

Examples

```
     10 A = DROUND(.3214,3)   !A = .321
     20 B = DROUND(.3215,3)   !B = .322
     30 C = DROUND(5.07,2)    !C = 5.1
```

ERRL

The ERRL function returns information about the last error trapped by an ON ERROR statement. It returns the line number that the error occurred in.

Syntax

ERRL

Example

```
     100 ON ERROR CALL Fixit
     110 I=J/0                    !The error occurred here.
     120 END
     200 SUB Fixit
     210 PRINT ERRL
     250 SUBEND
```

The above program prints:

```
     110
```

ERRM$

The ERRM$ returns information about the last error trapped by an ON ERROR statement. It returns an error message associated with the error number, as listed in Appendix A.

Syntax

ERRM$

Example

```
     100 ON ERROR CALL Fixit
     110 I=J/0                   !The error occurred here.
     120 END
     200 SUB Fixit
     210 PRINT ERRM$
     250 SUBEND
```

The above program prints:

```
     Division by zero, or modulo 0.
```

ERRMSHORT$

The ERRMSHORT$ functions returns information about the last error trapped by the ON ERROR statement. It returns an error message of the form:

ERROR *n* IN LINE *m*

where *n* is the HP Business BASIC/XL error number, and *m* is the line number that the error occurred in.

Syntax

ERRMSHORT$

Example

```
     100 ON ERROR CALL Fixit
     110 I=J/0                  !The error occurred here.
     120 END
     200 SUB Fixit
     210 PRINT ERRMSHORT$
     250 SUBEND
```

The above program prints:

```
     ERROR 31 IN LINE 110
```

ERRN

The ERRN function returns information about the last error trapped by the ON ERROR statement. It returns the HP Business BASIC/XL error number.

Syntax

ERRN

Examples

```
     100 ON ERROR CALL Fixit
     110 I=J/0                       !The error occurred here.
     120 END
     200 SUB Fixit
     210 PRINT ERRN
     250 SUBEND
```

The above program prints:

```
     31
```

EXP

The EXP function returns the value of **e** ** *n*. The result is a REAL number.

Syntax

EXP(*n*)

Parameters

n The power that **e** is to be raised to. Although this can be of any numeric type, it is converted to REAL.

Examples

```
     10 A = EXP(0)    !A = 1
     20 B = EXP(1)    !B = 2.71828
     30 C = EXP(1.0)  !C = 2.718281828459
```

FNUM

The FNUM function returns the MPE XL file number of a file. This is used primarily when calling MPE XL file intrinsics.

Syntax

FNUM(*fnum*)

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer. An optional # can precede *fnum*.

Examples

```
     100 MPE_num = FNUM(1)  !MPE_num is the MPE file number of file 1.
     120 REM                !MPE_num can then be used to call intrinsics
```

FRACT

The FRACT function returns the fractional part of a number. The result can be of type REAL, SHORT REAL, DECIMAL, or SHORT DECIMAL.

Syntax

FRACT(*n*)

Parameters

n The number to be evaluated. This can be of any numeric type.

Examples

```
     10 A = FRACT(2.7)   !A = .7
     20 B = FRACT(45)    !B = 0
```

INFO$

The INFO$ function returns the value of a string that was assigned to INFO following the command RUN;INFO=S$.

Syntax

INFO$

Examples

```
     >RUN;INFO="Debug"
```

In the program:

```
     120  IF INFO$="Debug" THEN
     ...
     180  ENDIF
```

In this case, the above block would execute since the expression INFO$="Debug" is true.

The INFO$ function can also be used with an HP Business BASIC/XL program file.

```
     :RUN Prog1;INFO="Debug"
```

The INFO$ function can be used within Prog1.

INT

The INT function returns the largest integer that is less than or equal to a specified number. The result is of type INTEGER.

Syntax

INT(*n*)

Parameters

n The number to be evaluated. This argument can be of any numeric type.

Examples

```
     10 A = INT(4.5)   !A = 4
     20 B = INT(-0.3)  !B = -1
```

INTEGER

The INTEGER function converts a number to an integer. The result is of type INTEGER.

Syntax

INTEGER(*n*)

Parameters

n The number to be converted. This can be of any numeric type.

Examples

```
     10 A = INTEGER(3.0)    !A = 3
```

INTERPRETED

The INTERPRETED function returns a value that determines whether a program is being run in the interpreter or as a compiled program.

The return value is as follows:

Table 5-5. Result of INTERPRETED Function

Result	Meaning
0	Compiled program.
1	Interpreted program.

Syntax

INTERPRETED

Examples

```
     10 A=INTERPRETED
    20 IF A=1 THEN GOSUB 100 !Control transfers to 100 if this program is interpreted
     30   ELSE GOSUB 200      !Control transfers to 200 if this program is compiled
     .

     .

     .
```

INV

The INV function returns an array that is the inverse of a specified array. Both arrays must be of the same floating-point type. HP Business BASIC/XL converts a DECIMAL or SHORT DECIMAL array to REAL before computing the inverse.

This function has the form

```
       MAT num_array1  = INV(num_array2 )
```

where *num_array1* is the inverse of *num_array2*. *Num_array1* and *num_array2* are both dimensioned (*m,m*) MUL(*num_array1*,*num_array2*) is an identity matrix. An identity matrix is a square matrix in which each element on the upper-left to lower-right diagonal is one and all others are zero. For example:

```
        1 0 0
        0 1 0
        0 0 1
```

The function is used in the MAT = statement, with two dimensional arrays.

Syntax

INV(*array*)

Parameters

array	Structured collection of variables of the same type. The structure is determined when the array is declared.

Examples

```
     10 DIM A(3,3),B(3,3)
     .
     .
     .
     50 MAT A = INV(B)
```

If B is

```
     1 2
     3 4
```

then A is

```
     -2    1
     1.5 -0.5
```

ITM

The ITM function returns the number of data items between the beginning of a record and its current position in the same record. In other words, it returns the number of datum between the beginning of the current record and the current datum pointer (after a direct read, this number is one).

Syntax

ITM(*fnum*)

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer. For this function, *fnum* must specify a BASIC DATA file. An optional # can precede *fnum*.

Examples

```
     10 CREATE "File1", FILESIZE=10  !BASIC DATA file; each PRINT
     11                                    ! statement starts a new record.
     12 ASSIGN "File1" TO #1
     13 POSITION #1; BEGIN                 !Pointer at record 1.
     14 PRINT #1; 10                       !Print 10 on record 1;
     16 PRINT #1; 20,30                    !Print 20 and 30 on the same record ;
     18 DISP ITM(#1)                       !Pointer is at record 1.
     19                                    !Three datum are between
     20                                    ! the pointer and the beginning
     21                                    ! of the record; display value 3.
     99 END
```

LASTBREAK

The LASTBREAK function is a Report Writer function that returns the level number of the last BREAK statement satisfied. If more than one BREAK statement is satisfied, it returns the lowest level number. If no report is active, it returns -1. If no breaks have occurred, it returns zero.

Syntax

LASTBREAK

Examples

```
     100 Level = LASTBREAK   !Level contains the level of the last BREAK.
```

LEN

The LEN function returns the length of a string expression in number of characters.

Syntax

LEN(*S* $)

Parameters

S $ The string expression whose length is to be returned.

Examples

```
     10  A = LEN("Cat")   !A = 3
```

LEX

The LEX function is used to compare two strings in a Native Language dependent manner. For example:

```
     LEX(String1$, String2$,Nl_var)
```

returns:

```
        -1 if String1$ < String2$
         0 if String1$ = String2$
         1 if String1$ > String2$
```

Syntax

LEX(*str_expr1*, *str_expr2* [*nl_num_expr*])

Parameters

str_expr1, *str_expr2* String variables, quoted strings, the values returned from a string function, or any expressions using the appropriate string operators to construct an expression.

nl_num_expr A numeric expression that evaluates to a Native Language ID. If *nl_num_expr* is -1, the underlying native language number is used as the language specifier. If a non-negative number is used, that number is taken directly as the language specifier. If the native language option is not specified then the option defaults to zero (the underlying native language).

The underlying native language specifies NATIVE-3000. NATIVE-3000 is the system language that does not consider Native Language Support. For more information on Native Language Support, refer to "Native Language Support" in chapter 6.

The native language number used for the comparison is determined by the normal selection process. A native language number can be supplied as the third argument.

LGT

The LGT function returns the log to the base 10 of a number. The result is a REAL number.

Syntax

LGT(*n*)

Parameters

n The number that log to the base **l0** is evaluated to. This is a REAL number.

Examples

```
     10 A = LGT(100)    !A = 2
     20 B = LGT(0.01)   !B = -2
```

LOG

The LOG function returns the log of **e** to a number. **e** is a constant that has the value of 2.718281828. This function returns a REAL number.

Syntax

LOG(*n*)

Parameters

n The number that log **e** is evaluated to. This argument is a REAL number.

Examples

```
     10 A = LOG(1)              !A = 0
     20 B = LOG(2.718281828)    !B = 1
```

LTRIM$

The LTRIM$ returns a copy of a string expression without leading blanks.

Syntax

LTRIM$(*S* $)

Parameters

S $ The string expression that is to be trimmed.

Examples

```
     10 A$ = LTRIM("  Hi")   !A$ = "Hi"
```

LWC$

The lowercase function, LWC$, converts a string with any uppercase letters to a string containing only lowercase letters. An optional second parameter can be used to specify the native language number.

Syntax

LWC$ (*str_expr* [, *nl_num_expr*])

Parameters

str_expr A string variable, a quoted string, the value returned from a string function, or any expression using the appropriate string operators to construct a string expression.

nl_num_expr A numeric expression that evaluates to a Native Language ID. If *nl_num_expr* is set to -1, the underlying native language number is used as the language specifier. If a non-negative value is used, that number is taken directly as the language specifier. If this option is not specified then the option defaults to zero (the underlying native language).

The underlying native is NATIVE-3000. NATIVE-3000 is the language the system uses before considering Native Language Support. Refer to "Native Language Support" in chapter 6 for more information.

MAX

The MAX function returns the largest value in a group of numbers. The result of this function is of the same type as the argument.

Syntax

MAX(*n* [,*n*]...)

Parameters

n Each number that is to be evaluated. These can be of any numeric type.

Examples

```
     10  A = MAX(3,1,2)    !A = 3
```

MAXLEN

The MAXLEN function returns the maximum length of a string expression, in characters. The maximum length is determined by the DIM statement or the system default.

Syntax

MAXLEN(*S* $)

Parameters

S $ A string expression whose maximum length is to be returned.

Examples

```
     10  DIM A$[30]
     20  B = MAXLEN(A$)     !B = 30
```

MIN

The MIN function returns the smallest value in a series of numbers. The result of this function will be of the same type as the arguments.

Syntax

MIN(*n* [,*n*]...)

Parameters

n Each number that is to be evaluated. These can be of any numeric type.

Examples

```
     10 A = MIN(3,1,2)   !A = 1
```

MUL

The MUL function returns an array that is the result of multiplying two arrays. The arrays being multiplied must be of the same numeric type and the result array must be a different variable than either of the arrays being multiplied. This function has the form:

MAT *num_array1* = MUL(*num_array2*,*num_array3*)

where *num_array1* is *num_array2* multiplied by *num_array3*. Table 5-6 shows the dimensions of each array in different cases.

Table 5-6. Dimensions of MUL Function Arguments and Results

Dimensions of *num_array2*	**Dimensions of *num_array3***	**Dimensions of *num_array1* (result)**
(*m,n*)	(*n,p*)	(*m,p*)
(*m,n*)	(*n*)	(*m*)
(*m*)	(*m,p*)	(*p*)

This function is used with the MAT = statement, with two dimensional arrays.

Syntax

MUL(*array1*,*array2*)

Parameters

array1 Structured collection of variables of the same type. The structure is determined when the array is declared.

array2 Structured collection of variables of the same type. The structure is determined when the array is declared.

Examples

Example 1.

```
     10 DIM A(2,4), B(2,3),C(3,4)
     .
     .
     .
     110 MAT A = MUL(B,C)
```

If B is

```
     5 3 1
     2 7 8
```

and C is

```
     1 2 8 5
     7 1 3 7
     6 4 2 9
```

then A is

```
     32 17 51 55
     99 43 53 131
```

Example 2.

```
     10 DIM A(2,1),B(2,4),C(4,1)
     .
     .
     .
     80 MAT A = MUL(B,C)
```

If B is

```
     9 8 7 6
     1 2 3 4
```

and C is

```
     1
     2
     3
     4
```

then A is

```
     78
     42
```

Example 3.

```
     10 DIM A(1,4),B(1,3),C(3,4)
     .
     .
     .
     110 MAT A = MUL(B,C)
```

If B is

```
     6 9 2
```

and C is

```
     1 2 8 5
     7 1 3 7
     6 4 2 9
```

then A is

```
     81 29 79 111
```

NUM

The NUM function returns the ASCII code that corresponds to the first character of a string. This is an integer in the range [0, 255].

Syntax

NUM(*S* $)

Parameters

S $ A string expression whose first character will be evaluated.

Examples

```
     10  A = NUM("Angle")    !A = 65
```

NUMBREAK

The NUMBREAK function is a Report Writer function that returns the number of BREAK conditions satisfied for levels one through the given level. Lower numbered breaks are counted because they automatically trigger a break at the given level. If there is no active report, an error occurs.

Syntax

NUMBREAK(*level*)

Parameters

level The summary level number. This must be in the range [0, 9].

Example

```
     100 No_conds = NUMBREAK(level3)  !Returns the number of break
     101                              !BREAK conditions satisfied
```

NUMDETAIL

This Report Writer function returns the number of DETAIL LINES with a non-zero *totals_flag* executed for the given level. This value is reset to zero each time a break occurs at the indicated level. If a report is not active, an error occurs.

The level number can be zero. Zero returns the total number of DETAIL LINE statements that accumulate totals. This value is used by the AVG function.

Syntax

NUMDETAIL(*level*)

Parameters

level The summary level number. This must be in the range of [0, 9]

Example

```
     110 Numbers = NUMDETAIL(Level1)  !Numbers receives the number of
     111                              !DETAIL LINES executed.
```

NUMLINE

The NUMLINE function is a Report Writer function that returns the number of lines printed on the current page. This number includes the blank lines at the top of the page (from PAGE LENGTH) and the page header. Each line printed with DETAIL LINE, PRINT, or PRINT USING also increments this value. If no report is active, -1 is returned.

Syntax

NUMLINE

Examples

```
     100  Lines= NUMLINE  !Lines receives the number of lines on the current page
```

NUMREC

The NUMREC function returns the number of records in a file that contain data.

Syntax

NUMREC(*fnum*)

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer. An optional # can precede *fnum*.

Example

```
     110 Filesize= NUMREC(2)     !Filesize is the number of records in file 2
```

OLDCV

The OLDCV function is a Report Writer function that returns the value of a BREAK WHEN control expression. The value stored is the value the last time a break at the given level (or lower level) occurred. The BREAK WHEN statement at this level must have a numeric control expression.

The OLDCV value is not available until report output is started. References to this function before that time generate an error.

Syntax

OLDCV(*level*)

Parameters

level A summary level number. This must be in the range [1, 9].

Examples

```
     100  bwval= OLDCV(level2)  !bwval receives the value of BREAK WHEN condition
```

OLDCV$

The OLDCV$ is a Report Writer function that returns the value of a BREAK WHEN control expression. The value stored is the value the last time a break at the given level (or lower level) occurred. The BREAK WHEN statement at this level must have a string control expression.

The OLDCV$ value is not available until report output is started. References to this function before that time generate an error.

Syntax

OLDCV$(*level*)

Parameters

level A summary level number. This must be in the range [1, 9].

Examples

```
     100 bwcont$ = OLDCV$(Level4)  !bwcont$ receives the value of a BREAK
     101                           !WHEN string control expression.
```

PAGENUM

The PAGENUM function is a Report Writer function that returns the current page number. The page number is set to 1 when a report is activated. You can reset this value with the SET PAGENUM statement. If no report is active, -1 is returned.

Syntax

PAGENUM

Example

```
     100 Cpage = PAGENUM   !Cpage receives the current page number
```

POS

The POS function returns the starting character of a string embedded in another string. It will return 0 if the specified substring is not found.

Syntax

POS(*S1* $,*S2* $)

Parameters

S1 $ A string expression indicating the string that the substring is to occur in.

S2 $ A string expression indicating the substring. The function returns the position of the first character of *S2 $*.

Examples

```
     10 A = POS("abcde","fg")     !A = 0
     20 B = POS("abcde","cd")     !B = 3
```

REAL

The REAL function converts a number to a real number. The result is of type REAL.

Syntax

REAL(*n*)

Parameters

n The number to be converted. This can be of any numeric type.

Examples

```
     10  A = REAL(2)   !A = 2.0000
```

REC

The REC function returns the number of the record indicated by a file's record pointer.

Syntax

REC(*fnum*)

Parameters

fnum The file number by which HP Business BASIC/XL identifies the file. It is a numeric expression that evaluates to a positive short integer. An optional # can precede *fnum*.

Examples

```
     10 CREATE "File1",RECSIZE=5,FILESIZE=10 !BDATA file; 10 5-word records
     11 ASSIGN "File1" TO #1
     12 POSITION #1;BEGIN                    !Pointer at record 1.
     13 DISP REC(#1)                         !Display 1.
     14 POSITION #1;3                        !Pointer at record 3.
     15 DISP REC(#1)                         !Display 3.
     16 PRINT #1,7; 502                      !Print 502 on record 7
     18 DISP REC(#1)                         !Display 7.
     19 POSITION #1;END                      !Pointer at end of file.
     20 DISP REC(#1)
     99 END
```

RECSIZE

The RECSIZE function returns the number of bytes per record in a file. The result of this function is of type INTEGER.

Syntax

RECSIZE(*fnum*)

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer. An optional # can precede *fnum*.

Examples

```
     100  Rec=RECSIZE(2)   !Rec is number of bytes per record of file 2
```

RESPONSE

The RESPONSE function returns information about the method and type of input last entered from the keyboard. The statements listed below affect the value that is returned by this function:

* INPUT

* LINPUT

* TINPUT

* ACCEPT

* PRESS KEY

* FLUSH INPUT

Likewise, the actions that are listed below affect the value that is returned by the RESPONSE function:

* Pressing any branch-during-input key.

* Pressing the HALT key (control Y).

* Specifying a HARD HALT.

* Execution of the PRESS KEY statement.

NOTE Input using the ENTER and LENTER statements does not affect the value returned by this function.

A FLUSH INPUT statement sets the value returned by RESPONSE to zero.

This function can be used in conjunction with the input statements and softkeys to determine how the user has a responded to a program's input statement. Table 5-7 lists the possible values returned by this function with their corresponding meanings.

Table 5-7. RESPONSE Function Return Values and Their Meanings

Value	**Meaning**
-1 through -8:	One of the user-definable keys , f1 through f8, was pressed. The value corresponds to the negative number of the actual key pressed.
-255:	The HALT key was pressed.
0:	There has not been any input entered or a FLUSH INPUT statement preceded the function call.
1:	The HARD HALT key was pressed.
2:	A timeout has occurred. This occurs during the execution of a TINPUT or ACCEPT statement.

10:	The last previous input is valid.
11:	The input was accepted without a carriage return. The TINPUT and ACCEPT statements allow suppression of the carriage return by specification of the CHARS and NOLF options.

Syntax

RESPONSE

Examples

```
     10 ON KEY 1 GOSUB Help;LABEL="Help"
     100 LOOP
     200 INPUT "Your Name; ";Name$
     300 EXIT IF RESPONSE > 2
     400 ENDLOOP
     500 STOP
     600 HELP:!
     700 PRINT "In Help"
     800 RETURN
```

The program above continues to prompt for the user's name until it is entered on the keyboard. If the user presses f1, the program executes the specified HELP subroutine. When it returns from the HELP subroutine, since RESPONSE returns a value of -1, the program reprompts for the user's name.

REVISION

The REVISION function returns the revision number of HP Business BASIC/XL running on the system. The result is of type INTEGER.

Syntax

REVISION

Examples

```
     10 DISP REVISION
```

RND

The RND function returns a pseudo-random number in the range of [0.0, 1.0]. The result is of type REAL. You can supply a dummy parameter.

Syntax

RND[(*n*)]

Parameters

n A dummy parameter. This dummy parameter, called a seed, is used by the RND function to completely determine a pseudo-random number sequence. For each seed number, a different random number sequence is generated. In order for the sequence to be correctly followed for multiple random numbers, the seed value from the previous RND call must be used as input for the next RND call, as each call changes the seed value. This is of type REAL.

Examples

```
     10  A = RND      !A = 0.237298
     20  B = RND(4)   !B = 0.789717
```

ROTATE

The ROTATE function returns the result of moving each bit of a number a specified number of bits. If the number of bits to be moved is positive, the bits move toward the right and if negative, the bits move left. If a bit is rotated past the last bit in the number, it is placed at the other end of the number. That is, the bits wrap around.

Syntax

ROTATE(N1,N2)

Parameters

N1 Binary representation of the value of a numeric expression. This is of type short integer. This is the number whose bits are to be rotated.

N2 Binary representation of the value of a numeric expression. This is a short integer. This parameter specifies the number of places the bits of N1 are to be rotated. N2 must be in the range [-32767, 32767].

Examples

The following example shows the bit layout for N1 and N2. It shows the bit layouts for N1 after the ROTATE function.

Bit Number:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
N1	0	1	1	0	0	0	1	1	0	1	0	1	1	0	1	0
ROTATE(N1,-1)	1	1	0	0	0	1	1	0	1	0	1	1	0	1	0	0
ROTATE(N1,1)	0	0	1	1	0	0	0	1	1	0	1	0	1	1	0	1
ROTATE(N1,2)	1	0	0	1	1	0	0	0	1	1	0	1	0	1	1	0
ROTATE(N1,18)	1	0	0	1	1	0	0	0	1	1	0	1	0	1	1	0
ROTATE(N1,4)	1	0	1	0	0	1	1	0	0	0	1	1	0	1	0	1

In the above example, note that ROTATE(N1,2)=ROTATE(N1,18) because 2=18 MOD 16.

ROUND

The ROUND function rounds a number to a specified power of 10. The result is of type DECIMAL.

Syntax

ROUND(*n1* [,*n2*])

Parameters

n1 The number to be rounded. This is of type DECIMAL.

n2 The power of 10 that *n1* is to be rounded to. If *n2* is not specified, 0 is the default.

Examples

```
10 A = ROUND(32767,2)    !A = 32800
20 B = ROUND(32067,3)    !B = 32000
30 C = ROUND(5.07,0)     !C = 5
40 D = ROUND(5.07)       !D = 5
50 E = ROUND(5.07,-1)    !E = 5.1
```

ROW

The ROW function returns the number of rows in an array as it is currently dimensioned. If it is a vector (one dimensional array), the number of rows is the number of elements. Otherwise, the number of rows is the size of the dimension that is second from the right in the *dims* of the DIM statement. The result is an integer value by default.

Syntax

ROW(*array*)

Parameters

array Structured collection of variables of the same type. The structure is determined when the array is declared. String variable names are suffixed with a "$".

Examples

OPTION BASE 1 is assumed.

```
     A(2,2): 1 2   B(2,4):  1 2 3 4   C(4,3,2): 1 2  0 4   0 0  1 2
             4 5            5 6 7 8             5 1  1 0   4 5  0 0
                                                2 0  3 2   1 2  0 1

     D(3,3): 1 0 1  E(2,2): 8 3       F(5): 5 4 3 2 1
             3 5 7          4 7
             9 0 9

     ROW(A) = 2
     ROW(B) = 2
     ROW(C) = 3
     ROW(D) = 3
     ROW(E) = 2
     ROW(F) = 5
```

RPOS

RPOS is a numeric function that returns the number of the row where the cursor is currently located. An integer in the range of one through the maximum number of lines in your terminal's display memory is returned. If your terminal has two pages of display, the value 48 is returned if the cursor is located on the last line of display memory. The program fragment:

```
        100 CURSOR (999)  !Position the cursor
        120 T_rows = RPOS !Moment of truth; how much display memory?
```

can be used to find the number of lines of display memory in a terminal. However, if the value 999 is returned, your terminal may have exceeded the valid display memory range for these statements. HP Business BASIC/XL determines cursor position by reading the position from the terminal. So, typing on the keyboard during the execution of an RPOS statement may cause an error.

Syntax

RPOS

Examples

```
     300 PRINT RPOS
```

RPT$

The RPT$ function returns a string that results from concatenating a specified string a specified number of times.

Syntax

RPT$(*S* $,*N*)

Parameters

S $

A string expression that contains the string to be concatenated.
N The number of times that *S* $ is to be concatenated. This is of type INTEGER.

Examples 20 A$ = RPT$("xy",3) !A$="xyxyxy"

RSUM

The RSUM function returns an array that contains the sum of the elements of each row of an array. Both arrays must be of the same type. This has the format

MAT *num_array1* = RSUM(*num_array2*)

where element *i* of *num_array1* is the sum of the elements in row *i* in *num_array2*. *num_array2* is dimensioned (*m*,*n*) and *num_array1* is dimensioned (*m*).

This function is used in the MAT = statement, with two-dimensional arrays.

Syntax

RSUM(*array*)

Parameters

array Structured collection of variables of the same type. The structure is determined when the array is declared.

Examples

```
     10  DIM A(3)
     20  DIM B(3,4)
     .
     .
     .
     80 MAT A = RSUM(B)
```

IF B is

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

then A is

```
     23 12 15
```

RTRIM$

The RTRIM$ function returns a copy of a string without trailing blanks.

Syntax

RTRIM$(*S* $)

Parameters

S $ A string expression that is to be evaluated.

Examples

```
     10  A$ = RTRIM("Hi     ")    !A$ = "Hi"
```

RWINFO

The RWINFO function is a Report Writer function that returns various pieces of information that may be useful in controlling the Report Writer. Table 5-8 shows the values returned. If there is no active report, -1 is returned.

Table 5-8. RWINFO Return Values

Input Value	Description
1	Page Size. Zero indicates an infinite page size.
2	Effective Page Size. Defined as page_size - # blank lines at top - # blank lines at bottom - size of PAGE HEADER - size of PAGE TRAILER.
3	NUMLINE value.
4	Lines left on current page. Includes the page trailer and blank lines at the bottom. Includes the page header if used in the PAGE HEADER section. Returns zero for an infinite size page.
5	Lines left on effective page. Equal to effective page size minus NUMLINE. Returns zero for an infinite size page.
6	Returns 1 if last page break was caused by DETAIL LINE statement. Returns 0 if any other statement causes a page break.
7	PAGENUM value.
8	Number of pages left to suppress.
9	Number of logical pages produced. This number is not affected by PAGENUM, and increments even when output is suppressed.
10	LASTBREAK value.
11	LEFT MARGIN column. Reflects the value given in the LEFT MARGIN statement, even if the left margin is not used.
12	Current summary level. Set during all HEADER and TRAILER sections, and during BREAK statement evaluation. Returns zero when not in a break condition.
13	PAGE HEADER size.
14	PAGE TRAILER size.
15	Returns 1 if DETAIL LINE causes a break. Otherwise, zero is returned.

Syntax

RWINFO(*input_value*)

Parameters

input_value A number specifying the information that RWINFO is to return. See Table 5-8 above for specific values.

Examples

```
     110  In_value = 7
     120  RWvalue= RWINFO(In_value)  !RWvalue receives the PAGENUM value.
```

SCAN

The SCAN function returns an integer containing the position of the first common character in two strings, scanning from left to right. It returns the position of the character in the first string specified. An optional third string parameter will return that first common character. If the two strings do not have common characters, SCAN returns 0 and the third string returns the null string.

Syntax

SCAN(*S1* $,*S2* $[,*S3* $])

Parameters

S1 $ A string expression containing one of the two strings that will be scanned. The position that SCAN returns is the position of the common character in this string.

S2 $ A string expression containing the second of the two strings that will be scanned.

S3 $ An optional parameter that will contain the first character that is common to *S1* $ and *S2* $. SCAN assigns a value to *S3* $.

Examples

```
     10   A = SCAN("abc","xbzz",A$)  !A = 2 and A$ = "b"
     20   B = SCAN("abc","djq",B$)   !B = 0 and B$ = "", the null string
     30   C = SCAN("abc","cba",C$)   !C = 1 and C$ = "a"
```

SDECIMAL

The SDECIMAL function converts a number to short decimal. The result of this function is of type SHORT DECIMAL.

Syntax

SDECIMAL(*n*)

Parameters

n The number to be converted. This is of any numeric type.

Examples

```
     10 A = SDECIMAL(5.678)    !A = 5.678
```

SGN

The SGN function evaluates the sign of a number. It returns the following value:

-1 if *n* is negative.
0 if *n* is zero.
1 if *n* is positive.

Syntax

SGN(*n*)

Parameters

n The number that is to be evaluated. This can be of any numeric type.

Examples

```
10 A = SGN(-239)    !A = -1
20 B = SGN(9-(3*3)) !B = 0
30 C = SGN(78.8)    !C = 1
```

SHIFT

The SHIFT function moves each bit of a number a specified number of places. If the number of places is positive, the bits move to the right, and if negative, to the left. If a bit is shifted out of the number, it is dropped.

Syntax

SHIFT(N1,N2)

Parameters

N1 Binary representation of the value of a numeric expression. This is a short integer. This is the number whose bits are to be shifted.

N2 Binary representation of the value of a numeric expression, a short integer. This is the number that specifies how many places to shift the bits. N2 must be in the range [-32767, 32767].

Examples

The following shows the bit layout for N1, and several examples of the SHIFT function. Each example uses a different value for N2.

Bit Number:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
N1	0	1	1	0	0	0	1	1	0	1	0	1	1	0	1	0
SHIFT(N1,-1)	1	1	0	0	0	1	1	0	1	0	1	1	0	1	0	0
SHIFT(N1,1)	0	0	1	1	0	0	0	1	1	0	1	0	1	1	0	1
SHIFT(N1,-3)	0	0	0	1	1	0	1	0	1	1	0	1	0	0	0	0
SHIFT(N1,4)	0	0	0	0	0	1	1	0	0	0	1	1	0	1	0	1

SIN

The SIN function returns the sine of a number. The result can be expressed in angular units of degrees, grads, or radians. The result is of type REAL.

Syntax

SIN(*n*)

Parameters

n The number to be evaluated. This is of type REAL.

Examples

```
10  A = SIN(60)   !A = .87 (Degrees)
20  B = SIN(60)   !B = .81 (Grads)
30  C = SIN(60)   !C = -.30 (Radians)
```

SINTEGER

The SINTEGER function converts a number to a short integer. The result is of type SHORT INTEGER.

Syntax

SINTEGER(*n*)

Parameters

n The number to be converted. This can be of any numeric type.

Examples

```
     10 A = SINTEGER(5.678)   !A = 6
```

SIZE

The SIZE function returns the number of records in a file, including unused or empty records. The result is of type INTEGER.

Syntax

SIZE(*fnum*)

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer. An optional # can precede *fnum*.

Examples

```
     160 Filesize = SIZE(1)   !Filesize is the number of records in file 1
```

SLEN

The SLEN function returns the string length of the next datum in a file. If that item is not a string, SLEN returns -1. The result is of type INTEGER.

Syntax

SLEN(*fnum*)

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer. *Fnum* must specify a BASIC DATA file. An optional # can precede *fnum*.

Examples

```
     100 CREATE "File1",FILESIZE=1000 !BDATA file; series of data items.
     110 ASSIGN "File1" TO #1
     120 POSITION #1; BEGIN       !Rewind File1 before writing it.
     130 PRINT #1; "abc", 123     !"abc" is item 1, 123 is item 2.
     140 POSITION #1; BEGIN       !Rewind File1.
     150 DISP SLEN(1)             !Next item, "abc", is a string;
     155                          ! return its length, 3.
     160 READ #1; A$              !Read item 1.
     170 DISP SLEN(1)             !Next item, 123, is not a string;
     175                          ! return -1.
     999 END
```

SQR

The SQR function returns the positive square root of a number. The result is of type REAL.

Syntax

SQR(*n*)

Parameters

n The number to be evaluated. This is of type REAL.

Examples

```
     10 A = SQR(25)    !A = 5
```

SREAL

The SREAL function converts a number to a short real. The result is of type SHORT REAL.

Syntax

SREAL(*n*)

Parameters

n The number to be converted. This can be of any numeric type.

Examples

```
     10 A = SREAL(4)    !A = 4.0000
```

SUM

The SUM function returns the sum of the elements in a numeric array. If the array is a short integer array, the result is an integer; otherwise, the result is the same type as the array.

Syntax

SUM(*num_array*)

Parameters

num_array Structured collection of variables of the same numeric type. The structure is determined when the array is declared.

Examples

OPTION BASE 1 is assumed.

```
     A(2,2): 1 2   B(2,4):  1 2 3 4   C(4,3,2): 1 2  0 4   0 0  1 2
             4 5            5 6 7 8             5 1  1 0   4 5  0 0
                                                2 0  3 2   1 2  0 1

     D(3,3): 1 0 1  E(2,2): 8 3       F(5): 5 4 3 2 1
             3 5 7          4 7
             9 0 9

     SUM(A) = 1+2+4+5 = 12
     SUM(B) = 1+2+3+4+5+6+7+8 = 36
     SUM(C) = 1+2+4+5+1+1+2+3+2+1+2+4+5+1+2+1 = 35
     SUM(D) = 1+1+3+5+7+9+9 = 35
     SUM(E) = 8+3+4+7 = 22
     SUM(F) = 5+4+3+2+1 = 15
```

TAN

The TAN function returns the tangent of a number. The result is of type REAL.

Syntax

TAN(*n*)

Parameters

n The number to be evaluated. This is of type REAL.

Examples

```
     10  A = TAN(50)    !A = 1.19 (Degrees)
     20  B = TAN(50)    !B = 1.00 (Grads)
     30  C = TAN(50)    !C = -.27 (Radians)
```

TASKID

The TASKID function returns the current task number. The task number is the PIN (Process Identification Number) for a process (in this case the PIN for the HP Business BASIC/XL interpreter or the compiled program). The PIN is assigned by the operating system for keeping track of multiple processes. You can use the PIN to find out more information about a process.

Syntax

TASKID

Examples

```
     10  Pin = TASKID   !After this call, pin will contain the PIN
     11                 !for this process.
```

TIME

The TIME function returns information about the current time of day, and the actual time elapsed since a program began execution.

Syntax

TIME (*num_expr*)

Parameters

num_expr A numeric expression that evaluates to an integer.

If *num_expr* evaluates to a real value, the TIME function rounds it to the nearest short integer before returning information.

The TIME function returns the following information dependent on the value of *num_expr*:

num_expr **Value**	**Information Returned**
Less than zero	Clock time since interpreter or compiled program began running
Zero	Minute of current time of day
One	Hour of current time of day
Two	Current day
Three or greater	Current year

Examples

```
     100 Run_time = TIME(-1)     !Returns clock time since program started
     110 Minute = TIME(0)        !Returns the current minute
     120 Hour = TIME(1)          !Returns the current hour
     130 Day = TIME(2)           !Returns the current day
     140 Year = TIME(3)          !Returns the current year
     150 Year = TIME(4.8)        !Also returns the current year
```

TIME$

The TIME$ function returns the current system time. The TIME$ function without an argument returns the time in the form "hh:mm:ss". For example:

TIME$(0)

returns the time in the NATIVE-3000 format which is "hh.mm AP" where hh is in 12-hour format and AP is either AM or PM. TIME$ and TIME$(0) are not the same. TIME$(8) returns the time in the German format "hh.mm" where hh is in 24-hour format.

Syntax

TIME$[(*nl_num_expr*)]

Parameters

nl_*num_expr* A numeric expression that evaluates to a Native Language ID. When the Native Language ID is not supplied, the current default value is used.

TOTAL

The TOTAL function is a Report Writer function that returns accumulated totals. The level number must match the level number of a TOTALS statement. The level number can be zero. Zero accesses the GRAND TOTALS statement.

Totals are always returned as REAL or DECIMAL numbers, depending on the setting of OPTION REAL/DECIMAL in the report subunit. If the current subunit has a different setting, the value may be converted if used in an expression.

Syntax

TOTAL (*level*,*sequence*)

Parameters

level The summary level number. This is in the range [0, 9].

sequence Indicates which expression in the given TOTALS statement should be returned. The first expression is sequence number one. An error occurs if the sequence number is less than one or greater than the number of expressions in the TOTALS statement.

Examples

```
     100  Tot = TOTAL(Level1,Seq)  !Tot receives the accumulated totals for the
     101                           !level specified by Level1, and the
     102                           !expression specified by Seq.
```

TRIM$

The TRIM$ function returns a copy of a string without leading or trailing blanks.

Syntax

TRIM$(*S* $)

Parameters

S $ A string expression that is to be trimmed.

Examples

```
     10  A$ = TRIM$("  ab   ")    !A$ = "ab"
```

TRN

The TRN function returns an array whose elements are the exchanged rows and columns of a specified array. Both arrays must be the same type. It has the form

MAT *num_array1* = TRN(*num_array2*)

where the rows of *num_array1* are the columns of *num_array2*, and the columns of *num_array1* are the rows of *num_array2*.*num_array1* is dimensioned (*n*,*m*) and *num_array2* is dimensioned (*m*,*n*).

This function is used with the MAT = statement, with two-dimensional arrays.

Syntax

TRN(*array*)

Parameters

array Structured collection of variables of the same type. The structure is determined when the array is declared.

Examples

```
     10 DIM A(4,3),B(3,4)
     .
     .
     .
     80 MAT A = TRN(B)
```

If B is

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

then A is

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

TRUNC

The TRUNC function returns the integer part of a number. The result is of the same numeric type as the argument.

Syntax

TRUNC(*n*)

Parameters

n The number that is to be evaluated. This is of any numeric type.

Examples

```
     10 A = TRUNC(57.571)    !A =  57
     20 B = TRUNC(-57.541)   !B =  -57
```

TYP

The TYP function returns a number that represents the type of the next datum in a file. See Table 5-9 below.

Table 5-9. Numbers Representing File Data Types

TYP Result	**Type of Next Item in File**
0	Unrecognized
1	DECIMAL
2	Entire string
3	End-of-file (EOF) mark
4	End-of-record (EOR) mark
5	SHORT INTEGER
6	SHORT DECIMAL
8	Beginning of string
9	Middle of string
10	End of string
11	INTEGER
12	SHORT REAL
13	REAL

Syntax

TYP(*fnum*)

Parameters

fnum The file number that HP Business BASIC/XL uses to identify the file. It is a numeric expression that evaluates to a positive short integer. *Fnum* must specify a BASIC DATA file. An optional # can precede *fnum*.

Examples

```
     110  Type = TYP(2)      !Type is type of next datum in file 2
```

UPC$ Function

The uppercase function, UPC$, converts a string with any lowercase letters to a string that is entirely uppercase. An optional second parameter can be used to specify the native language number.

Syntax

UPC$ (*str_expr* [, *nl_num_expr*])

Parameters

str_expr A string variable, a quoted string, the value returned from a string function, or any expression using the appropriate string operators to construct an expression.

nl_num_expr A numeric expression that evaluates to a Native Language ID. If *nl_num_expr* is set to -1, the underlying native language number is used as the language specifier. If a nonnegative value is used, that number is taken directly as the language specifier. If the native language option is not specified, then the option defaults to zero.

Examples

```
     10  A$ = UPC$("Joe")    !A$ = "JOE"
```

USRID

The USRID function returns the User ID (logical device) number of the job/session input device.

Syntax

USRID

Examples

```
     10 SYSTEM "SHOWJOB" !Lists the system jobs and sessions on your terminal
     20 PRINT USRID       !The User ID and Logical device number of the session
     21              !or job that is running this program is displayed.
```

VAL

The VAL function returns a number representing the numeric string at the beginning of a string expression. It will ignore the rest of the string expression.

Syntax

VAL(*S* $)

Parameters

S $ A string expression to be evaluated. If *S* $ does not start with a legal integer or real number, an error occurs.

Examples

```
     10  A = VAL("12ABC")   !A = 12
     20  B = VAL("3.45pq")  !B = 3.45
     20  C = VAL("9.00")    !C = 9.00
```

VAL$

The VAL$ function returns the string formed by enclosing a number in quotes.

Syntax

VAL$(*N*)

Parameters

N A numeric expression that is to be evaluated. This can be of any numeric type.

Examples

```
     10  A$ = VAL$(12)    !A$ = "12"
     20  B$ = VAL$(3.45)  !B$ = "3.45"
```

VERSION$

The VERSION$ function returns a string indicating the current version of HP Business BASIC/XL. The string has the form V.bb.ff (V=Version, bb=build, ff=fix).

Syntax

VERSION$

Examples

```
     10  A$ = VERSION$    !A$ = A.00.00
```

WORD

The WORD function returns the position of an embedded substring within a string. The substring is considered embedded only if the characters surround the substring are nonalphabetic.

Syntax

WORD(*S1* $,*S2* $)

Parameters

S1 $ A string expression containing the string to be searched.

S2 $ A string expression containing the substring to be found in *S1* $.

Examples

```
     10 A = WORD("cat","a")     !A = 0
     20 B = WORD("a cat","a")   !B = 1
     30 C = WORD("c a t","a")   !C = 3
     40 D = WORD("c,a.t","a")   !D = 3
```

WRD

The WRD function returns the number of the word indicated by the file's word pointer. The result is of type INTEGER.

Syntax

WRD(*fnum*)

Parameters

fnum The file number by which HP Business BASIC/XL identifies the file. It is a numeric expression that evaluates to a positive short integer. *Fnum* must specify a BASIC DATA file. An optional # can precede *fnum*.

Examples

```
     10 CREATE "File1",RECSIZE=5,FILESIZE=10 !BDATA file; 10 5-wd recs.
     11 ASSIGN "File1" TO #1
     12 PRINT #1,9,2; 36           !Print 36 on record 9, word 2;
     13                            ! move pointer to word 3.
     14 PRINT #1,9,4; 567          !Print 567 on record 9, word 4;
     15                            ! move pointer to record 5.
     16 DISP WRD(#1)               !Display 5.
     17 PRINT #1,9,5; 98           !Print 98 on record 9, word 5;
     18                            ! move pointer to record 10, word 1
     19 DISP WRD(#1)               !Display 1.
     99 END
```

Chapter 6 Input and Output

Introduction

An HP Business BASIC/XL program can receive input from any of the following:

* A terminal keyboard.
* An input file.
* A data file.

It can produce output on any of the following:

* A terminal screen.
* A printer.
* A data file.

An output statement that specifies the output format produces *formatted* output; an output statement that does not specify the output format produces *unformatted* output.

This chapter explains the following:

* Receiving input from a terminal keyboard or an input file.
* Producing unformatted output on a terminal screen, a printer, or a data file supported by the operating system, but not a BASIC DATA file.
* The format specifiers available to produce formatted output.

When an HP Business BASIC/XL output statement passes a character sequence to an output device, the resulting output depends on the output device's interpretation of the individual characters. For example, a sequence that repositions the cursor on a terminal may be ignored by a printer. Information about how an individual output device interprets a specific character sequence is contained in the manual for that output device.

Input from the Keyboard or Input File

This section describes the use of the ACCEPT, INPUT, LINPUT, and TINPUT statements. These statements are defined in chapter 4. If HP Business BASIC/XL is running interactively, these statements accept input from a terminal keyboard. Each of these statements suspends an executing program so that you can enter values on the keyboard.

The program is in the *input state* while it is suspended waiting for input. The input state ends when you press RETURN. The input consists of all characters that you type before pressing RETURN. ACCEPT and TINPUT optionally allow the programmer to specify the length of the input item. The program leaves the input state when the designated number of characters is entered.

If HP Business BASIC/XL is running in a job stream, these statements take input from the job stream file or an input file. The input is obtained from the next record in the appropriate file.

The ACCEPT, INPUT, LINPUT, and TINPUT statements differ in the type of input that they accept and whether they echo input to the display. Options are available to print a specific prompt on the terminal, specify the maximum time allowed for input, monitor the amount of time required for input, specify the maximum input length, and suppress the line feed following input. Table 6-1, Table 6-2, and Table 6-3 present this information.

Table 6-1. Keyboard Input Statements

Statement	Acceptable Input	Variable(s) to Which Statement Can Assign Input
ACCEPT	Characters from ASCII or default foreign character set.	One string variable.
INPUT	List of literals.	One or more scalar variables, array elements, or arrays.
LINPUT	String literal.	One scalar string variable or string array element.
TINPUT	Literal.	One scalar variable or array element.

Table 6-2. Keyboard Input Statements

Statement	Echo Input to Display	Print a Prompt for Terminal Input
ACCEPT	No	No
INPUT	Yes	Yes
LINPUT	Yes	Yes
TINPUT	Yes	No

Table 6-3. Keyboard Input Statement Options

Statement	Specify Maximum Input Time Allowed	Monitor Time Required for Input	Specify Maximum Input Length	Suppress LF Following Input
ACCEPT	Yes	Yes	Yes	Yes
INPUT	No	No	No	No
LINPUT	No	No	No	No
TINPUT	Yes	Yes	Yes	Yes

Input Prompt

This section explains how prompts are displayed when HP Business BASIC/XL is running interactively. Input prompts are not displayed if HP Business BASIC/XL is running in a job stream.

A prompt can be supplied for each input element except within a FOR clause. The default prompt is a question mark.

Table 6-4 shows how the prompt option and the separator that follows affect the cursor position.

Table 6-4. Effect of Input Prompt and Separator on Cursor

Is Prompt Supplied?	**Prompt that HP Business BASIC/XL Uses**	**Separator Following Prompt**	**Where HP Business BASIC/XL Puts the Cursor After Printing the Prompt and Putting the Program in the Input State**
Yes	The prompt that was supplied.	;	HP Business BASIC/XL does not move the cursor.
Yes	The prompt that was supplied.	,	At the beginning of the next line.
Yes	The prompt that was supplied.	None	At the beginning of the next line.
No	Question mark (?).	Not applicable	At the beginning of the next line.

Interactive Input from a Terminal

This section explains how to enter input when HP Business BASIC/XL is running interactively. The rules for input from a keyboard also apply to input from an input file.

After the INPUT statement displays a prompt and puts the program into the input state, you can type values on the terminal keyboard. Individual values are separated by commas or semicolons. If numeric values are expressed in European format where either a comma or period is the radix indicator, then input values must be separated with semicolons.

Double quotes surrounding string values are optional. The string must be enclosed in quotes if it contains a comma, a semicolon, or leading or trailing blanks. Otherwise, these symbols are interpreted as item separators. If a string value that is enclosed in quotes contains a quote, the quote that it contains must be duplicated; for example, the unquoted string

```
"Hi," he said.
```

must be quoted (because it contains a comma), and the quotes that it contains must be duplicated:

```
"""Hi,"" he said."
```

Variable Assignment during Interactive Input

When you press RETURN, the INPUT statement assigns the values to the variables specified by the input list. The first value input is assigned to the leftmost variable in the input list, the second value to the next variable, and so on.

If you type more values than the number of variables listed in the input list, the INPUT statement ignores the extra values. If you type fewer values than the number of variables listed in the input list, the INPUT statement prompts you for more values until values have been assigned to all variables. If the input list contains an array reference, you must input one value for each array element. If a user prompt is not specified for the additional variables requiring values, the prompt is **??**.

If you input a value that cannot be assigned to its corresponding variable; for example, you input a string for a numeric variable, the INPUT statement reprompts you. Assignments to preceding variables are not affected.

Job Stream Input

This section explains how the INPUT statement reads input when HP Business BASIC/XL is running in a job stream. The input for an input statement in a jobstream is either included in the stream file or obtained from a file specified by a redirection of the interpreter's input. The method of redirecting the interpreter's input is discussed at the beginning of chapter 2. Here, we discuss the method for including the input for INPUT statements in a stream file. The records in the job stream file immediately following the command that begins program execution are used to satisfy the input items for the INPUT statement. The values in the stream file are separated by commas, semicolons, or EOR marks. HP Business BASIC/XL suppresses prompts specified in any prompt option in an INPUT statement when HP Business BASIC/XL is running a stream job.

Variable Assignment during Job Stream Input

The INPUT statement assigns the values in the job stream file to the variables specified in the input list. The first value in the stream file record is assigned to the leftmost variable in the input list, the second value to the next variable, and so on.

If the record in the stream file has more values than the number of variables in the input list, the additional values are ignored. If the record has fewer values than the input list needs, HP Business BASIC/XL reads the next record to find additional values. If records or values are not found, an error occurs. If the input list contains an array reference, the record must contain one value for each array element. Colons in the INPUT statement save unassigned input values or use input values saved from previous INPUT statements.

If a value in the input file record cannot be assigned to its corresponding variable, HP Business BASIC/XL aborts the program. This occurs if the record contains a string value that is to be assigned to a numeric variable.

If numeric values are expressed in the format in which a comma is the radix indicator, then input values must be separated with semicolons or EOR marks.

The rules for string input from a job stream file are the same as the rules for string input during an interactive session from a terminal.

Unformatted Output

This section explains how to produce unformatted output on an output device:

* Unformatted output statements that have one of the following characteristics:

    - Produce output, but do not specify format.
    - Are the BEEP, DISP, and PRINT statements.

- Numeric format statements that have the following characteristic:

    - Specify format for numeric output of the DISP and PRINT statements.

- Output device specification statements that performs one of the following:

    - Directs output to specific output devices.

    - If a program does not contain output device specification statements, then all output from the program and the HP Business BASIC/XL interpreter is displayed on the standard list device. If HP Business BASIC/XL is running interactively, the standard list device is the terminal. If HP Business BASIC/XL is running in a job stream, the standard list device is the line printer of the computer system that HP Business BASIC/XL is running on. The MARGIN statement sets the terminal screen margin.

Each statement is defined in chapter 4.

The Display List

Commas and Semicolons in Display List. A comma or semicolon in the display list separates individual output items in the *output_item_list*. Table 6-5 summarizes the differences between commas and semicolons as separators.

Table 6-5. Semicolon vs Comma in Display List

Separator	**Second item is displayed**	**Display Enhancements Active from First Item Remain Active**
Semicolon	Immediately after first item.	Yes.
Comma	At beginning of next output field.	No.

If a DISP or PRINT statement ends with a comma or a semicolon, then it does not print a carriage return and a line feed after its display list. Subsequent output appears on the same line.

If a DISP or PRINT statement does not end with a comma or semicolon, it prints a carriage return and a line feed after its display list. Subsequent output to the same device appears on the next line.

Array References in Display List. The DISP or PRINT statement prints an array in row-major order; that is, the rightmost subscript varies fastest. Each time the rightmost subscript reaches its maximum value, the DISP or PRINT statement prints a carriage return and a line feed.

The spacing of array elements depends on what follows the array specification in the display list, as shown in Table 6-6.

Table 6-6. Semicolon vs Comma After an Array

If array is followed by	**Numeric elements are printed**	**String elements are printed**
Semicolon	Side by side.	On consecutive output lines.

Comma	In consecutive output fields.	On consecutive output lines.
Nothing (it is the last item in the list)	In consecutive output fields.	On consecutive output lines.

The DISP or PRINT statement prints two blank lines after printing the entire array.

An array can also be printed with the MAT PRINT statement, described in chapter 4.

Output Functions in Display List. The display list of a DISP or PRINT statement can contain any of the following output function calls:

```
CTL (num_expr )
END
LIN (num_expr )
PAGE
SPA (num_expr )
TAB (num_expr )
```

Each output function call directs the DISP or PRINT statement to print one or more control characters on the output file or device. If a control character is sent to an output file, it affects the operation of the line printer that prints the output file. If a control character is sent to an output device, it affects the device itself.

The following paragraphs explain the individual output functions, using these terms:

n Value of *num_expr*; for example, *n* is 10 in "TAB 2*5".

cl Number of the current output line; for example, *cl* is 12 if the next output item is printed on the twelfth line of the output file.

cc Number of the current output character position; for example, *cc* is 20 if the first character of the next output item will be printed in the twentieth character position of the output line.

m The right page margin, output line length. It is set with the MARGIN parameter in the output device specification. See "Device Specification Syntax" for more information.

CTL The CTL function returns the carriage control character that is represented by *n*. On the operating system in which HP Business BASIC/XL is running, *n* must be the code for a carriage control character. HP Business BASIC/XL does not check this. The DISP or PRINT statement prints the output items that precede the CTL call prior to generating the carriage control character. For the effects of specific carriage control characters, see the manual for the operating system in which HP Business BASIC/XL is running.

END The END function returns the end-of-file character. It can only be used when the specified output device is a file.

LIN The LIN function returns ABS(*n*) line feed characters.

If *n* is positive, the following occurs:

"LIN (*num_expr*)" specifies *n* line feed characters and a carriage return character. The next output item is printed at the beginning of line *cl+n*.

If *n* is zero, the following occurs:

"LIN (*num_expr*)" specifies only a carriage return character (zero line feed characters). The next output item is printed at the beginning of line *cl* (line *cl* is overwritten).

If *n* is negative, the following occurs:

"LIN (*num_expr*)" specifies *-n* line feed characters and no carriage return character. The next output item is printed on line *cl+(-n)*, starting at character position *cc* +1.

PAGE The PAGE function returns a form feed character. When the file is printed, the form feed character advances the line printer to the next physical page. PAGE affects only ASCII files opened with carriage control specified. If the output device is a terminal, or an ASCII file with no carriage control specified, PAGE has no effect.

SPA The SPA function returns *n* spaces or a carriage return character if the current output line has fewer than *n* spaces left; that is, if *cc+n* exceeds *m*.

If *n* is positive, the following occurs:

The next output item is printed on the current output line, starting at character position *cc+n*, if possible.

If *cc+n* exceeds *m*, the following occurs:

The SPA call specifies a carriage return character. The next output item is printed at the beginning of line *cl+1*.

If *n* is negative, the following occurs:

An error occurs.

TAB The TAB function resets *cc* (and prints a carriage return character, if necessary).

If *n* is positive, the following occurs:

If TAB increases *cc* and *n* <=*m* then the next output item is printed on line *cl*, starting at character position *n*. If TAB increases *cc* and *n* exceeds *m*, the next output item is printed on line *cl* +1 starting at character position *n* MOD *m*. If TAB decreases *cc*, the next output item is printed on line *cl* +1, starting at character position *cc*.

If *n* is zero, the following occurs:

The TAB call has no effect.

If *n* is negative, the following occurs:

An error occurs.

Unless the output function call is the last item in the output list, HP Business BASIC/XL ignores the delimiter (comma or semicolon) following it. If that delimiter is immediately followed by one or more commas, HP Business BASIC/XL skips one output field for each comma. For example, the first comma in PRINT PAGE,,,A has no effect, but HP Business BASIC/XL skips one output field for the second comma and another for the third.

Numeric Format Statements

The FIXED, FLOAT, and STANDARD statements are numeric format statements. Each statement specifies a different default numeric format--fixed-point, floating-point, or standard, respectively. The unformatted output statements, DISP and PRINT output numeric values in the default numeric format.

Before the program executes a numeric format statement, the default numeric format is standard.

If a program contains more than one numeric format statement, the most recently executed statement applies, with one exception: numeric format statements in a subunit are canceled when control returns to the calling program unit.

Examples

```
  10 FIXED 2              !Fixed-point format goes into effect
  15 INPUT X
  16 PRINT X              !Print X in fixed-point format
  20 CALL Sub1(X)         !Call Sub1; format is fixed-point
  30 PRINT X              !Print X in floating-point format
  99 END
 100 SUB Sub1(N)
 105   PRINT N            !Print N in fixed-point format
 110   STANDARD           !Standard format goes into effect
 115   PRINT N            !Print N in standard format
 116   CALL Sub2(N)       !Call Sub2; format is standard
 117   PRINT N            !Print N in standard format
 120 SUBEND               !Return to line 30 and floating-point format
 200 SUB Sub2(P)
 210   PRINT P            !Print P in standard format
 220   FLOAT 4            !Floating-point format goes into effect
 230   PRINT P            !Print P in floating-point format
 240 SUBEND               !Return to line 117 and floating-point format
```

If 125.7689 is entered for x in line 5, the above program prints:

```
?125.7689
 125.77
 125.77
 125.7689
 125.7689
 1.2577 E+02
 1.2577 E+02
 1.2577 E+02
```

Output Device Specification

The SEND OUTPUT TO, SEND SYSTEM OUTPUT TO, and COPY ALL OUTPUT TO are output specification statements. Table 6-7 indicates what they specify.

Table 6-7. Device Specification Statements

Device Specification Statement	**Specifies**
SEND OUTPUT TO	Device for PRINT statement output.
SEND SYSTEM OUTPUT TO	System printer.
COPY ALL OUTPUT TO	Device for interpreter and program output.

Each of these statements is defined in chapter 4.

If an output specification statement specifies a spooled output device, HP Business BASIC/XL opens a spool file. If a subsequently executed output specification statement specifies the same spooled device, HP Business BASIC/XL closes the spool file that it opened for the first statement and opens another spool file. Unless the first spool file is the standard list device, it is ready for printing when HP Business BASIC/XL closes it. See "Spooled Output Devices" for more information.

If a program does not contain output device specification statements, then all output from the program and the HP Business BASIC/XL interpreter is displayed on the standard list device. If HP Business BASIC/XL is running interactively, the standard list device is the terminal. If HP Business BASIC/XL is running in a job stream, the standard list device is the line printer of the computer system HP Business BASIC/XL is running on.

An output specification statement in any program unit affects the entire program. If a program contains more than one SEND OUTPUT TO, SEND SYSTEM OUTPUT TO, or COPY ALL OUTPUT TO statement, the most recently executed one applies. It cancels any previously executed statement of its kind, but not output specification statements of another kind. For example, a SEND OUTPUT TO statement cancels any previously executed SEND OUTPUT TO statements, but not SEND SYSTEM OUTPUT TO or COPY ALL OUTPUT TO statements.

Spooled Output Devices. If an output device specification statement specifies a spooled device, HP Business BASIC/XL opens a spool file. If a subsequently executed output device specification statement specifies the same spooled device, HP Business BASIC/XL closes the spool file that it opened for the first statement and opens another one unless the spooled device is PRINTER. See the next paragraph for information about spooled device PRINTER. For example, if *LP is a file reference to a spooled device, then when HP Business BASIC/XL executes the statement SEND OUTPUT TO "*LP", it opens a spool file for PRINT statement output. If HP Business BASIC/XL then executes the statement SEND SYSTEM OUTPUT TO "*LP", it closes the spool file that it opened for PRINT statement output and opens another spool file for system output.

If the standard list device is a spooled device, then HP Business BASIC/XL opens a spool file when it executes the statement SEND SYSTEM OUTPUT TO PRINTER or COPY ALL OUTPUT TO PRINTER. However, if HP Business BASIC/XL then executes the statement SEND OUTPUT TO PRINTER, it does not close the spool file and open another one. Therefore, it sends system output and PRINT statement output to the same spool file.

Device Specification Syntax (dev_spec). Each output device specification statement specifies an output device. The output device is called *dev_spec* (device specification) in the syntax specification for each statement. If *dev_spec* is not a legal output device, an error occurs and HP Business BASIC/XL substitutes the standard list device for *dev_spec*. The syntax for *dev_spec* follows.

Syntax

     [[,MARGIN *num_expr1*], FIELD *num_expr2*]
dest [[,FIELD *num_expr2*], MARGIN *num_expr1*]

Parameters

dest Destination device. See Table 6-8.

num_expr1 A numeric expression that evaluates to the number of characters in an output line. *num_expr1* is rounded to an integer, *n1*, which is called the margin. It is best thought of as the number of characters to reach the right margin. After an output statement prints *n1* characters on a line, it prints a carriage return and line feed on that line. Remaining characters are printed on the next line.

The margin cannot be less than the output field width, *num_expr2*. If *n1* is less than the field width, the margin is set to the value of the field width. If the output file is an ASCII disk file with fixed-length records, the margin cannot exceed the record length. For these files, if *n1* is greater than the record length, the margin is set to the value of the record length.

Default margin: See Table 6-9. Also, see the MARGIN

statement.

num_expr2 A numeric expression that evaluates to the number of characters in an output field. *num_expr2* is rounded to an integer, *n2*, called the output field width.

The output line begins with *n1* DIV *n2* output fields of *n2* characters each. If *n1* MOD *n2* is not zero, the output line ends with one output field of *n1* MOD *n2* characters. For example; a line with margin 75 and output field width 20 begins with three 20-character fields and ends with a 15-character field.

If the length of an output item exceeds the output field width, it is still printed.

Default output field width: See Table 6-10.

NOTE If HP Business BASIC/XL is running interactively, an output specification statement that specifies a margin has a side effect: it sets the terminal margin to *n1*.

Table 6-8 gives the possible destination values and the devices that they specify.

Table 6-8. Destination Device Specifiers

Specifier	**Destination**
NULL	The system $NULL file. Usually used to discard output.
DISPLAY	Terminal if HP Business BASIC/XL is running interactively; equivalent to PRINTER if HP Business BASIC/XL is running in a job stream.
PRINTER	The system printer, or if the system printer is a spooled device, a spool file to be sent to the system's printer. In the statements SEND SYSTEM OUTPUT TO, and SEND OUTPUT TO, if you specify *dev_spec* to be PRINTER, HP Business BASIC/XL uses the file equation specified as your printer file. This information is kept in the HP Business BASIC/XL configuration file and can be changed by running CNFGHPBB.Pub.Sys.
formal_designator [{ ,}] [{ .}FILESIZE[=] fsize] [{ ,}]	*formal_designator* has the same syntax as *fname* described in "File Identification" in chapter 9. Additionally, the *formal_designator* syntax includes the quoted string literals "$STDLIST", "$NULL" and "**fname* ". The **fname* syntax is used to reference device files that have been previously defined using file equations. The *formal_designator* for *dev_spec* must reference a device or an ASCII or BDATA file. If the file does not exist, HP Business BASIC/XL creates an ASCII disk file with fixed length 80 byte records. FILESIZE is an optional parameter allowing specification of the maximum number of records in the file. *fsize* is a

	numeric expression that is evaluated and rounded to an integer as required.

Table 6-9 gives the default margins for different types of destination devices.

Table 6-9. Default Margins

Destination Device	**Default Margin if Margin Was Previously Specified for the Device**	**Default Margin if Margin Was Not Previously Specified for the Device**
Terminal	Last specified margin.	80
Line printer	Last specified margin.	132
Other	132	132

Table 6-10 gives the default output field widths for different types of destination devices.

Table 6-10. Default Output Field Widths

Destination Device	**Default Output Field Width if Output Field Width Was Previously Specified for the Device**	**Default Output Field Width if Output Field Width Was Not Previously Specified for the Device**
Terminal	Last specified output field width.	20
Line printer	Last specified output field width.	20
Other	20	20

HP Business BASIC/XL opens an existing ASCII disk file in *append mode*; new records are appended to the existing records in the file. "Data Files", later in this chapter, has more information.

If HP Business BASIC/XL tries to append an additional record to an ASCII disk file for which the end-of-file marker is at the physical end of file, an error message is displayed on the terminal. Redirect the output to DISPLAY at that point. A new file can then be specified to accept the redirected output. If you repeatedly encounter problems with the file size, use the FILESIZE option to create a larger file.

FORMATTED OUTPUT

This section explains the format specifiers available to produce formatted output. These are available with the DISP USING, PRINT USING, and IMAGE Statements. The DISP USING and PRINT USING statements are formatted output statements; they specify the output format to be used in printing data. A DISP USING or PRINT USING statement can specify output to format directly in a format string or indirectly by referencing an image statement.

The format string or IMAGE statement describes the output format exactly, specifying the following:

- Type of output.
- Spacing.
- Position of the following, if appropriate:
  - Plus or minus signs.
  - Radix indicators.
  - Exponents.
  - Dollar signs.
  - Blanks.
  - Control characters.

Format String

The format string specifies the output format for the output items in the display list of a DISP USING or PRINT USING statement. It is also used in an IMAGE statement.

Syntax

format_string

Parameters

format_string — *format_string* (if it belongs to an IMAGE statement) or its value (if *format_string* itself is the image of a PRINT or DISP statement) has the following syntax:

format_spec [, *format_spec*]...

[*num_expr*] (*format_spec* [, *format_spec*]...)

format_spec — One of the format specifiers described in "Format Specifiers" in the following section.

num_expr — Repeat factor. Rounded to a short integer, *n*. The *format_string n* (*format_spec_list*) is equivalent to *n* adjacent copies of *format_spec_list* (see examples).

Examples

The format strings of lines 100 and 200 are equivalent. In line 200, three is the repeat factor represented by *num_expr*, above.

```
100 DISP USING "DDD,XX,DDD,XX,DDD,XX"; A,B,C
200 PRINT USING "3 (DDD,XX)"; A,B,C
300 DISP USING "DDDDD,XX,ZZZ.DD"; P,Q
```

Format Specifiers

The *format_spec* in a format string or IMAGE statement is one of the specifiers listed in Table 6-11. Each numeric, nonliteral string, or compact specifier corresponds to one output item in the display list of a DISP USING or PRINT USING statement. A space, dollar, control character, or literal string specifier does not correspond to an item in the display list. Instead, it directs the DISP USING or PRINT USING statement to print or suppress characters.

Table 6-11 lists the format specifiers, tells what they specify and how they are symbolized, and whether they can contain repeat factors.

Table 6-11. Format Specifiers

Specifier	**Specifies**	**Symbolized by**	**Can Contain Repeat Factor**
Numeric	One numeric output item.	D,Z,*,.,R,S, M,C,P,E	Yes
String literal	That the literal be printed exactly.	Quoted string literal.	No
String	One string output item.	A	Yes
Compact	One numeric or string output item.	K	No
Space	One or more spaces.	X	Yes
Dollar	Dollar sign.	$	Yes
Control Character	That control characters be printed or suppressed.	#,+,-,@,/	No

Starting with the leftmost output item in the display list and starting at the beginning of the format string or IMAGE statement, HP Business BASIC/XL matches each output item to the next numeric, nonliteral string, or compact specifier. For example, in the statement

```
     100 DISP USING "2X,DD,3X,5A"; 12,"HELLO"
```

2X and 3X are space specifiers, the numeric specifier DD corresponds to the value 12 and the string specifier 5A corresponds to the value "HELLO".

If the specifiers outnumber the output items, HP Business BASIC/XL ignores the extra specifiers. For example, in the sequence

```
     200 PRINT USING 210; A,B
     210 IMAGE Z,X,D,2X,ZZ,3X,DD
```

the numeric specifier Z corresponds to the variable A, X is a space specifier, the numeric specifier B corresponds to the variable B, and the specifiers 2X,ZZ,3X, and DD are ignored.

If the output items outnumber the specifiers, HP Business BASIC/XL reuses the format string or IMAGE statement. For example, in the statement

```
     300 DISP USING "5A,X,2D,X"; "HELLO",12,"HOWDY",34
```

the string specifier 5A corresponds to "HELLO" and "HOWDY" and the numeric specifier 2D corresponds to 12 and 34.

An error occurs if a numeric specifier corresponds to a string value or if a string specifier corresponds to a numeric value. For example, the statement

```
     400 PRINT USING "DDZ.DD"; "GOOD-BYE!"
```

causes an error, since DDZ.DD is a numeric specifier and "GOOD-BYE!"is a string.

Numeric Specifiers

A numeric specifier specifies the output format for a numeric value. It can contain digit symbols, radix symbols, sign symbols, digit-separator symbols, an exponent symbol, and repeat factors (numeric expressions). Each symbol represents one printed character.

Syntax

```
              fraction_part
             [                 ]
integer_part  [{E fraction_part }]
             [{fraction_part  E}]
```

Parameters

```
                         {{D} }
                    [S]  {{Z} }
integer_part         [M][n ]{{*} }
                         {{C} }
                         {K...}

                    {.}            [S]
fraction_part        {R}[[n ]D[D]...][M][n]D[D]...
```

n Repeat factor; a numeric expression. The symbol that follows it is repeated *n* times; for example, 5D is equivalent to DDDDD.

See the sections "Digit Symbols" and "Digit-Separator Symbols" for restrictions on combinations of the symbols D, Z, *, C, and P that the above syntax specifiers do not reflect.

Table 6-12 summarizes the types of symbols that a numeric specifier can contain, what each type specifies, and the individual symbols of each type and their differences.

Table 6-12. Numeric Specifier Symbols

Symbol	**Symbol Type**	**A symbol of this type specifies**	**This symbol specifies that**
D	Digit	One digit position.	Each leading zero is replaced with a blank.
Z	Digit	One digit position.	Leading zeros are printed.
*	Digit	One digit position.	Each leading zero is replaced with an asterisk (*).
.	Radix	Position of radix; which	Radix is a period (.).

		separates integer and fractional parts of a number.	
R	Radix	Position of radix.	Radix is a comma (,).
S	Sign	Position of sign symbol (+ or -).	+ is printed if number is positive; - is printed if number is negative.
M	Sign	Position of sign symbol.	Blank is printed if number is positive; - is printed if number is negative.
C	Digit-separator	Position of digit-separator symbol (comma or period) that separates groups of digits (as in 1,000,000).	Digit-separator is comma (U.S. notation).
P	Digit-separator	Position of digit-separator symbol.	Digit-separator is period (European notation).
E	Exponent	Scientific notation and the position of the symbol E in that notation.	Not applicable.

Digit Symbols. Each of the three digit symbols, D,Z, and *, specifies one digit position. The DISP USING or PRINT USING statement prints one digit of the output value for each digit symbol in the format specifier.

The digit symbols vary in that:

D	Replaces each leading zero with a blank (" ").
Z	Prints leading zeros.
*	Replaces each leading zero with an asterisk (*).

A repeat factor can precede a digit symbol.

Examples

```
     20 DISP USING 50; 5,5,5
     30 DISP USING 60; 25,367,5448
     40 DISP USING 60; 12345,12345,12345
     50 IMAGE ZZZZZ,XX,DDDDD,XX,*****
     60 IMAGE 5Z,2X,5D,2X,5*
     99 END
```

The above program prints:

```
     00005      5  ****5
     00025    367  *5448
     12345  12345  12345
```

Lines 50 and 60 are equivalent (line 60 uses repeat factors). Each of the specifiers XX and 2X specifies two spaces (see "Edit Specifiers", later in this chapter, for more information). Notice that the specifiers

5Z, 5D, and 5* output a five-digit value the same way (because the value has no leading zeros).

The digits in the integer part of a number can be represented by any digit symbol; however, all of the digits must be represented by the same digit symbol, with one exception. The digit in the one's place can be represented by Z, regardless of the symbol that represents the other digits. For example, DDD.DD, ZZZ.DD, ***.DD, DDZ.DD, and **Z.DD are legal. DZD.DD, Z**.DD, and *DZ.DD are illegal. Each digit in the fractional part of a number must be represented by D.

Examples

```
100 A=123.45
110 B=67.8
120 C=90
130 D=0.2
140 E=0.76
150 PRINT USING 200; A,A,A,A,A
160 PRINT USING 200; B,B,B,B,B
170 PRINT USING 200; C,C,C,C,C
180 PRINT USING 200; D,D,D,D,D
190 PRINT USING 200; E,E,E,E,E
200 IMAGE DDD.DD,2X, ZZZ.DD,2X, ***.DD,2X, DDZ.DD,2X, **Z.DD
999 END
```

The above program prints:

```
123.45  123.45  123.45  123.45  123.45
 67.80  067.80  *67.80   67.80  *67.80
 90.00  090.00  *90.00   90.00  *90.00
   .20  000.20  ***.20    0.20  **0.20
   .76  000.76  ***.76    0.76  **0.76
```

If a numeric output format specifies x digits to the right of the radix, and the output value is precise to more than x digits, the DISP USING or PRINT USING statement prints the output value, rounded to x decimal places. Rounding the output does not actually change the value.

If a numeric output format specifies x digits to the right of the radix, and the output value is precise to fewer than x digits, the DISP USING or PRINT USING statement prints zeros in place of the missing digits.

Examples

```
100 X=1.2938
110 Y=3.7465
120 Z=4.99
130 DISP USING 160; X,X,X,X
140 DISP USING 160; Y,Y,Y,Y
150 DISP USING 160; Z,Z,Z,Z
160 IMAGE D.DDDD,2X, D.DDD,2X, D.DD,2X, D.D
170 DISP USING "D.DDDD,2X,D.DDDD,2X,D.DD"; X,Y,Z
999 END
```

The above program prints:

```
1.2938  1.294  1.29  1.3
3.7465  3.747  3.75  4.0
4.9900  4.990  4.99  5.0
1.2938  3.7465  4.99
```

Radix Symbols.   The radix symbols, (period (.) and R), specify the character that separates the integer and fractional parts of a number. It can be either a decimal point or a comma. In a numeric specifier, a period (.) specifies a decimal point and an R specifies a comma. A numeric specifier can have at most one radix symbol.

Examples

```
     DISP USING "DD.DD,2X,DDRDD"; 12.34, 12.34
```

The above statement prints:

```
     12.34 12,34
```

Sign Symbols. The sign symbols, (S and M), specify the sign character. A numeric specifier can have at most one sign symbol.

The sign symbols vary in that:

S Prints a plus (+) if the output value is positive, and a minus (-) if it is negative.

M Prints a blank if the output value is positive, and a minus if it is negative.

Examples

```
     100 IMAGE SDD,2X,SDD
     200 IMAGE MDD,2X,MDD
     300 DISP USING 100; 10,-10
     400 DISP USING 200; 10,-10
     999 END
```

The above program prints:

```
     +10  -10
      10  -10
```

The sign can be printed between digits.

Examples

```
     650 PRINT USING "2(DSD,2X,DMD,2X)"; -12,-34,56,78
```

The above statement prints:

```
     1-2  3-4  5+6  7 8
```

Digit-Separator Symbols. The digit-separator symbols, (C and P), specify the character that separates groups of digits as the commas do in "1,000,000". The symbol C specifies a comma; the symbol P, a period.

Before printing a digit-separator symbol, the DISP USING or PRINT USING statement prints at least one digit of the output value. That digit can be a leading zero, if leading zeros are printed.

Examples

```
     100 W=1234567
     110 X=800342
     120 Y=1234
     130 Z=150
     140 PRINT USING 300; W,W,W
     150 PRINT USING 400; W,W,W
     160 PRINT USING 300; X,X,X
     170 PRINT USING 400; X,X,X
     180 PRINT USING 300; Y,Y,Y
     190 PRINT USING 400; Y,Y,Y
     200 PRINT USING 300; Z,Z,Z
     210 PRINT USING 400; Z,Z,Z
     300 IMAGE 7Z,2X,ZC3ZC3Z,2X,ZP3ZP3Z
     400 IMAGE 7D,2X,DC3DC3D,2X,DP3DP3D
     500 PRINT USING "DCDDD.DD,2X,DCDDDPZZ"; 123456,123456
     600 PRINT USING "DDCDDCDD"; 123456
     999 END
```

The above program prints:

```
     1234567  1,234,567  1.234.567
     1234567  1,234,567  1.234.567
     0800342  0,800,342  0.800.342
      800342    800,342    800.342
     0001234  0,001,234  0.001.234
        1234      1,234      1.234
     0000150  0,000,150  0.000.150
         150        150        150
     1,234.56  1,234.56
     12,34,56
```

Exponent Symbol. The exponent symbol, E, specifies scientific notation. A numeric specifier must have at least one digit symbol before the symbol E. The DISP USING or PRINT USING statement prints the output value in the format

```
{+}                                   {+}
{-} digit [digit...][.digit [digit...]]E{-} digit digit
```

The exponent symbol can precede or follow the fractional part of the numeric specifier. The numeric specifier must contain a sign symbol if the output value is negative.

Examples

```
     100 N=123.45
     110 DISP USING "D.DDDE"; N    !1.235E+02 (rounded)
     120 DISP USING "DDDDD.E"; N   !12345.E-02
     130 DISP USING "3D.2DE"; -N   !Overflow error
     140 DISP USING "S3D.2DE"; -N  !-123.45E+00
     999 END
```

String Specifiers

A string specifier specifies the output format for a string value. The specifier can be nonliteral or literal. A nonliteral string specifier contains the symbol A, which can be preceded by a repeat factor (numeric expression). A literal string specifier is a quoted literal.

Syntax

Nonliteral string specifier:

[*num_expr*]A

Literal string specifier:

str_lit

Parameters

num_expr Repeat factor. Its value is the length of the output string. If this is not specified, the default is one.

str_lit Literal string specifier. It must be enclosed in quotes and it can only appear in an IMAGE statement (not in a format string). It does not correspond to an item in the display list; the DISP USING or PRINT USING statement prints *str_lit* itself.

A nonliteral string specifier specifies the output format for a string value in the display list. It can appear in either an IMAGE statement or a format string.

Examples

Legal:

```
     500 PRINT USING 310
     510 IMAGE 30X,"Title"
```

Illegal:

```
     600 PRINT USING "30X,"Title""
```

If *num_expr* has the value *n* and the corresponding output item is a string of length *s*, then:

If	**DISP USING or PRINT USING statement prints**
n =s	Entire string
n <s	First *n* characters of the string
n>s	Entire string, followed by *n-s* blanks

Examples

```
      99 S$="GOODBYE"
     100 DISP USING "7A"; S$     !Format length = output string length.
     110 DISP USING "4A"; S$     !Format length < output string length.
     120 DISP USING "8A"; S$     !Format length > output string length.
     130 DISP USING 140; S$
     140 IMAGE 7A,"TO YOU"       !image contains literal.
     150 DISP USING "4AX3A"; S$  !Insert blank in printed string.
```

Where "Å" represents a blank, the above program prints:

```
     GOODBYE
     GOOD
     GOODBYEÅ
     GOODBYETO YOU
     GOOD BYE
```

Standard Format Specifier

A standard format specifier represents one string or numeric value of any size. It consists of one symbol, K. If K represents a string value, the DISP USING or PRINT USING statement prints the entire string. If K represents a numeric value, the statement prints the value in the standard format, without leading or trailing blanks.

Syntax

K

Examples

```
      10 X=123
      20 Y=.4567
      30 Z=-1.234E+47
      40 A$="cat"
      50 B$="bird"
     100 PRINT USING "K"; X
     110 PRINT USING "K"; Y
     120 PRINT USING "K"; Z
     130 PRINT USING "K,K,K"; X,Y,Z
     140 PRINT USING "K"; A$
     150 PRINT USING "K,K"; A$,B$
     160 PRINT USING "K,K,K,K,K"; X,A$,Y,B$,Z
     999 END
```

The above program prints:

```
123
.4567
-1.234E+47
123.4567-1.234E+47
cat
catbird
123cat.4567bird-1.234E+47
```

Space Specifiers

A space specifier specifies one or more spaces.

Syntax

[*num_expr1*]X[X]...

Parameters

num_expr1   Repeat factor. Its value is rounded to a short integer.

The specifier nX is equivalent to a sequence of nX symbols. The DISP USING or PRINT USING statement prints one space for every X.

Examples

```
110 DISP USING "3D,XXX,3D,XXX,3D"; 123,456,789
120 DISP USING "3D,3X,3D,3X,3D"; 123,456,789
999 END
```

The above program prints:

```
123   456   789
123   456   789
```

Dollar Specifier

A dollar specifier specifies a dollar sign ($) and consists of one symbol, $. When the symbol $ precedes a numeric specifier, the DISP USING or PRINT USING statement prints a dollar sign ($) before printing the value that corresponds to the numeric specifier. The statement prints the dollar sign immediately before the first printed digit of the output value.

Syntax

$

Examples

```
10 A=1234
20 DISP USING "$DCDDD.DD"; A
30 DISP USING "$DDDCDDD.DD"; A
40 DISP USING "$DDDCDDZ.DD"; A
50 DISP USING "$ZZZCZZZ.DD"; A
99 END
```

The above program prints:

```
$1,234.00
$  1,234.00
$  1,234.00
$1,234.00
$001,234.00
```

Control Character Specifiers

A control character specifier specifies that one or more control characters for carriage return, line feed, or form feed be printed or suppressed. It consists of one symbol: #, +, -, @, or /.

Table 6-13 lists the control character specifiers, their positions in the *image*, (the item that is output) and their effect on the DISP USING or PRINT USING statement.

Table 6-13. Control Character Specifiers

Specifier	**Position in Image**	**Effect on Output**
#	First *format_spec*	Suppresses carriage return and line feed that would otherwise be printed after display list values.
+	First *format_spec*	Suppresses line feed that would otherwise be printed after display list values.
-	First *format_spec*	Suppresses carriage return that would otherwise be printed after display list values.
@	Any *format_spec*. Can also replace comma.	Prints formfeed.
/	Any *format_spec*. Can also replace comma except that two @s must be separated by a comma.	Formfeed followed by line feed.

Examples

```
10 A$="ABC"
20 DISP USING "3A"; A$
30 DISP USING "K"; "xyz"
40 DISP USING "#,3A"; A$  !Suppress carriage return & line feed
50 DISP USING "K"; "xyz"
60 DISP USING "-,3A"; A$  !Suppress carriage return only
70 DISP USING "K"; "xyz"
99 END
```

The above program prints:

```
ABC
xyz
ABCxyz
ABC
   xyz
```

The sequence:

```
100 DISP USING "+,3A"; A$
110 DISP USING "K"; "xyz"
```

prints ABC, followed by a carriage return character but not a line feed character. When the output file is printed on a line printer, xyz is printed over ABC.

The statement:

```
     200 DISP USING "DD,@,DD@DD@,@DD"; 12,13,14,15
```

prints 12, a form feed character, 13, a form feed character, 14, two form feed characters, 15. When the output file is printed on a line printer, 12 is printed on the current page, 13 on the next page, 14 on the next, and 15 two pages after 14.

The statement:

```
     300 DISP USING "Z,/,ZZ/ZZZ//ZZZZ/,//ZZZZZ"; 1,2,3,4,5
```

prints:

```
        1
        02
        003

        0004

        00005
```

Data Files

A data file contains data that an HP Business BASIC/XL program can read or has written. The file can be stored on a disk, magnetic tape, or cards. HP Business BASIC/XL uses program files as well as data files. The material in this section applies only to data files, unless otherwise noted. See chapter 2 for information about program files.

The following summarizes the material in this section:

TITLE	**CONTENT**
Data File Types	The three types of data files that HP Business BASIC/XL uses
File Identification	How HP Business BASIC/XL identifies a data file
File Input and Output	Read from or write to a data file.

Data File Types

HP Business BASIC/XL uses three types of data files: BASIC DATA, binary, and ASCII. Table 6-14 shows their similarities and differences.

Table 6-14. Data File Types

	ASCII	**BASIC DATA**	**Binary**
How Created	CREATE statement or operating system command.	CREATE statement or operating system command.	CREATE statement or operating system command.
Fixed Length	Yes, if created with CREATE statement. If created with an operating system command, it depends	Yes, if created with CREATE statement. If created with an operating system command, it depends	Yes, if created with CREATE statement. If created with an operating system command, it depends

	on the command.	on the command.	on the command.
Formatted	No	Yes	No
Input	READ statement, LINPUT statement.	READ statement.	READ statement.
Output	PRINT statement.	PRINT statement.	PRINT statement.
Misc.	Data items are separated by commas or record boundaries in an ASCII input file. PRINT statement must print commas between data items if READ statement is to read ASCII file after it is printed.	READ statement type-checks BASIC DATA file data before assigning it to variables. Direct word reads and writes are possible (see "File Input and Output"). Conceptually, a series of data items actually, a series of records.	No wasted space; no item separators as there are in ASCII files. No item descriptors as there are in a BASIC DATA file.

The BASIC DATA file is the only formatted file. It contains format words that describe each datum. When a program writes a datum to a BASIC DATA file, HP Business BASIC/XL writes the appropriate format words to the BASIC DATA file (the statement that writes to the file need not specify them). When a program reads a string datum from a BASIC DATA file, HP Business BASIC/XL checks the format words for its type and for its size.

Conceptually, a BASIC DATA file is a series of data items, rather than a series of records. Actually, it is composed of records; each record contains as many whole data items as it can, with one immediately following another. A datum never crosses a record boundary.

ASCII and binary files are unformatted; they do not contain format words that describe their data.

File Identification

The CREATE statement or operating system command that creates a file names the file; the ASSIGN statement assigns a file number to it. The CATALOG and file management statements reference files by their names; the file functions and other statements reference them by their numbers.

fname is a file name used in the Syntax Specification in chapter 4. *fname* is represented by one of the following:

* A quoted string literal (for example, "Myfile").

* An unquoted string literal (for example, Myfile).

* A string expression (for example, "File"+ A$).

The following restrictions apply to an unquoted string literal file representation:

* It must begin with a letter (uppercase or lowercase).

- Its first nonalphabetic character cannot be "$".

- It cannot contain the following characters:
  - comma *(,)*
  - semicolon *(;)*
  - space *( )*
  - exclamation point *(!)*
  - right parenthesis *())*

The format of the file name depends on the operating system. For example, if HP Business BASIC/XL is running on the HP 3000 under MPE XL, the format of *fname* is

filename [/*lockword*][.*groupname* [.*accountname*]]

where *filename, lockword, groupname,* and *accountname* are strings of one to eight alphanumeric characters. The first character must be alphabetic in each.

File Number Syntax (fnum). *fnum* is the file number that HP Business BASIC/XL uses to identify the file. In the syntax specifications in chapter 4, *fnum* is any numeric expression that evaluates to a positive short integer greater than zero. The operating system may identify the same file with another number (see the file function FNUM). The character # must precede *fnum*, except when *fnum* is a parameter in a call to one of HP Business BASIC/XL's predefined file functions (then the # is optional).

Examples

Legal fname	**Representation**
"*myfile"	Quoted string literal - file back reference
Abc$	String expression
"mylife. mygroup"	Quoted string literal
File$+Group$+ Account$	String expression
myfile	Legal unquoted string literal
myfile/password. mygroup	Legal unquoted string literal

Illegal fname	**Reason it is illegal**
*myfile	Does not start with a letter
Abc$.mygroup	First nonalphabetic character is "$"
Abc);def	Contains ")"

An HP Business BASIC/XL program must assign a file number to a file before it can access it; it must open the file. A program can assign more than one file number to a file; open it more than once. See the ASSIGN Statement for more information.

Filecodes. If you list your data or program files, you will see the following file code mnemonic associated with each type of file:

Filecodes

Mnemonic	**Filecode**	**Description**
BSAVE	1244	HP Business BASIC/V Save file.
BSVXL	1247	HP Business BASIC/XL Save file.

BDATA	1242	HP Business BASIC/V Data file.
BDTXL	1248	HP Business BASIC/XL Data file.
BBNCM	1249	MPE/V binary file.

The filecode associated with each of the data files is used to identify whether the file stores information in the MPE/V or MPE XL format. When the file is opened by HP Business BASIC/XL, the file code is used to determine the data storage format. If the file code indicates that the file is was created on MPE/V, all the subsequent work of floating-point real data conversion is done automatically. Therefore, it is possible to share data among MPE XL native mode applications and existing programs not yet migrated from compatibility mode. However, if the data file is only intended for native mode programs and the data file was created on MPE/V or in compatibility mode, run the conversion program BBCTMPEV.PUB.SYS to avoid the performance impact of data conversion.

File Input and Output

File input and output (I/O) statements read input from and write output to data files. The following input statements are available:

LINPUT
MAT READ
READ

The following output statements are available:

PRINT
UPDATE

In addition, the CATALOG statement is used to display directory information about specified files. All of these statements are explained in chapter 4.

Each data file has a record pointer and a word pointer associated with it. A BASIC DATA file has a datum pointer as well:.

record pointer   Indicates the next record to be read or written.

word pointer   Indicates the next word (within the next record) to be read or written.

datum pointer   Indicates the next datum to be read or the next place to write a datum.

After any file I/O operation, the record, word, and datum pointers advance to the next respective record, word, or datum depending on the type of I/O operation. The POSITION statement positions the record pointer at a specified record. The ADVANCE statement moves the record pointer forward or backward. These statements are defined in chapter 4.

Regardless of file type, a file I/O operation can be:

sequential   Sequentially reads or writes to the record in the file indicated by the position of the record pointer.

direct   The record pointer is moved directly to a specific record prior to reading or writing.

On a BASIC DATA file, a file I/O operation can also be:

direct word — Both the record and word pointers are moved to a specific word in the file prior to reading and writing.

Refer to Table 6-15 for the data storage and data item descriptor size for each data type in the BASIC DATA file. This is useful for direct record and word I/O to a BASIC DATA file.

Table 6-15. BASIC DATA File Contents

	Data Storage Size (in fileword)	**Descriptor Size (in fileword) and [Descriptor Value]**
Short Integer	1	1 [5]
Integer	2	1 [6]
Short Decimal	2	1 [7]
Short Real	2	1 [8]
Decimal	4	1 [9]
Real	4	1 [10]
Entire String (for string that fits into one record)	(total no. of chars + 1) div 2	2 [1], second word is the total no. of chars in the string
Beginning of String	(record size -2)	2 [2], second word - total no. of chars in string
Middle of String	(record size - 2)	2 [3], second - total no. of chars left
End of String	(total no. of chars left + 1) div 2	2 [4], second word = total no. of chars left

Table 6-15 Note: The length of each fileword is two bytes for consistency with the MPE XL file system.

Native Language Support

This section summarizes the features of HP Business BASIC/XL that facilitate the production of native language independent code. Refer to the *Native Language Programmer's Guide* for more information on Native Language Support or NLS.

Selecting a Native Language

HP Business BASIC/XL determines the native language number at the start-up of the interpreter and when a compiled program is executed by making the following checks in the order shown:

1. The initial default is NATIVE-3000 (Language #0).

2. The operating system default language is determined by the NLINFO intrinsic.

3. The HP Business BASIC/XL configuration file is checked for language specification.

4. The value of the MPE NLDATALANG *job control word* or jcw is used if defined.

At all times while running the HP Business BASIC/XL interpreter and executing a compiled HP Business BASIC/XL program, there is an associated native language number. This number is referred to as the *underlying native language number* in this section, and in the descriptions of NLS statements and in NLS functions.

Displaying the Native Language Number

The INFO command displays the language number and the name of the language in the following format:

```
Native Language  0(Native-3000)
```

Changing the Native Language Number

The underlying native language number can be changed with the RUN and SCRATCH ALL commands. Each time the RUN command is issued, HP Business BASIC/XL checks the value of the MPE jcw, NLDATALANG. If it is defined and has a different value from the current native language number, then the native language number changes. This causes HP Business BASIC/XL to open the message catalog appropriate for that language (HHBBCnnn.PUB.SYS for language nnn; for example, HPBBC009.PUB.SYS for Italian). If it is not possible to open that catalog, HHBBCAT.PUB.SYS is used instead.

The native language can also be changed by the SCRATCH ALL command. The SCRATCH ALL command follows the same procedure outlined under "Selecting a Native Language" for determining a language number. If this results in a number that is different from the current one, the native language number changes.

Changing the NLDATALANG jcw does not affect the underlying native language number until the next RUN or SCRATCH ALL command is executed. Obviously, the language number cannot change during the execution of a compiled program.
The ways of changing the NLDATALANG jcw include the following:

* Using HP Business BASIC/XL's "SYSTEM" command:

```
>SYSTEM "setjcw nldatalang=3"
```

* Using HP Business BASIC/XL's ":" escape:

```
>:setjcw nldatalang=3
```

String Functions

Relevant string functions have been enhanced to allow an option numeric argument that specifies a native language number. In each case, if the argument's value is -1, the underlying native language number is used as the language specifier. If a non-negative value is used, that number is taken directly as the language specifier. If the native language option is not specified, then the option defaults to zero. The following functions include parameters for NLS:

```
  LWC$
  UPC$
  LEX
  DATE$
  TIME$
```

These functions are defined and explained in chapter 5.

Chapter 7 The Report Writer

Introduction

The Report Writer consists of HP Business BASIC/XL statements that aid in report generation by doing various bookkeeping jobs. In the Report Writer certain control structures cause the statements to be executed at the appropriate times. The PRINT and IMAGE statements specify the actual printing of the report.

Report Writer statements are categorized into the following four classes:

* Report Writer Section Statements.

* Report Writer Block Statements.

* Report Writer Executable Statements.

* Report Writer Built-In Functions.

This chapter describes the four classes of the Report Writer in detail. Syntax and descriptions of each statement are in chapter 4.

General Information

Be aware of the following item, since it affects various Report Writer statements:

* The report sections (REPORT HEADER, REPORT TRAILER, and REPORT EXIT) are at level zero.

Report Writer section statements define the headers and trailers printed in the report. These statements are included within the report description. A REPORT HEADER section defines the beginning of the report description and the END REPORT DESCRIPTION statement defines the end of the report description. Both of these sections are required, whereas all other Report Writer sections are optional.

A Report Writer section starts with a section statement. It ends when the next section statement occurs in the report description. The section can contain any legal HP Business BASIC/XL program statements. These statements execute when the section is activated by the Report Writer.

The following are Report Writer section statements:

* REPORT HEADER

* REPORT TRAILER

* PAGE HEADER

* PAGE TRAILER

* HEADER

* TRAILER

* REPORT EXIT

* END REPORT DESCRIPTION

The WITH and USING clauses, used with the Report Writer section statements, are described later in this section.

All of the report writer section statements are made BUSY and their expressions are evaluated when BEGIN REPORT executes, preventing their modification and deletion. When the report ends, these section statements are no longer busy. That is, these report writer section statements are busy for the duration of an active report.

WITH and USING Clauses

The WITH and USING clauses control the automatic page break mechanism and to aid in the printing of each section. The WITH and USING clauses can occur in all of the Report Writer section statements except END REPORT DESCRIPTION and in the DETAIL LINE statements. These clauses are both optional; however if both clauses occur, the WITH clause must appear first.

The USING clause is an *implicit* PRINT USING statement.

                         [LINES]
Syntax. WITH *num_lines* [LINE]

USING *image* [; *output_list*]

Parameters.

num_lines The maximum number of lines the section statement expects to need. This can be any non-negative integer, including zero. This number reflects ALL output done by the section. For DETAIL LINE, all lines printed between any two detail lines is included.

image An image string or a line reference to an IMAGE line to control printing.

output-list A list of output items, identical to the list used by the PRINT USING statement.

Examples. The following are examples of the WITH and USING clauses:

```
     100 REPORT HEADER WITH 3 LINES
     110 DETAIL LINE USING 100;A, B
     120 PAGE TRAILER WITH 2 LINES USING Pt;PAGENUM, DATE$
```

Whenever a section becomes active, the first action executed is the section statement. The WITH clause is evaluated first. If the number of lines left on the page is smaller than the WITH value, an automatic page break results. Otherwise, the WITH clause has no effect.

The WITH clause ensures that a certain number of lines are available before the page trailer prints. If this condition is not satisfied, the page break ensures that enough lines are available. If a WITH clause is not present, the default is one.

The USING clause executes after the WITH clause. This clause is similar to a PRINT USING statement in the report section statement. See PRINT USING for more details.

If an error occurs during evaluation of the WITH clause, such as a negative number of lines specified, the USING clause does not execute. If the USING clause encounters an error, it stops printing. In either case, however, the rest of the report section executes. That is, if there is an error in the WITH clause, the USING clause will not execute, but the rest of the section will execute.

Exceptional Cases. The WITH clauses of the PAGE HEADER and PAGE TRAILER sections are exceptional. Instead of evaluating the WITH clause at each page break, the Report Writer evaluates the PAGE HEADER and PAGE TRAILER size only when BEGIN REPORT executes. This action allows the Report Writer to define the number of lines normally available for printing. The maximum size of the page header and the size of the page trailer are fixed throughout the report. Refer to the PAGE HEADER and PAGE TRAILER statements for more details.

The USING clauses of the PAGE HEADER and PAGE TRAILER sections are evaluated each time there is a page break.

Report Writer Block Statements

The Report Writer block statements further define a report by providing execution control as well as report layout. All of these statements must occur within a report description. Some of the statements must occur within certain sections of the report. The point each statement becomes busy at, or is evaluated, varies from statement to statement.

If a Report Writer block statement executes when a report is not active, an error occurs. When there is an active report, the direct execution of the statement acts as a comment. These statements execute only when certain other Report Writer statements execute, such as DETAIL LINE.

The following are Report Writer block statements:

- PAGE LENGTH
- LEFT MARGIN
- PAUSE EVERY
- SUPPRESS AT
- SUPPRESS FOR
- PRINT DETAIL IF
- TOTALS
- GRAND TOTALS
- BREAK IF
- BREAK WHEN

Report Writer Executable Statements

The Report Writer executable statements drive the report process. A report becomes active when a BEGIN REPORT statement executes. However, this is distinct from starting report output. Starting report output is caused by other Report Writer executable statements. The DETAIL LINE statement is the primary method of printing the report. END REPORT and STOP REPORT cease report activity.

These statements must appear in the same subunit as the report description they use. They can appear anywhere within the subunit, although some of these statements are not allowed inside the actual report description.

Activating and Starting a Report

A distinction must be made between activating a report and starting a report output. This distinction is important because of the interactions of PRINT with the report writer.

The BEGIN REPORT statement activates a report. This means that the report description is scanned and verified, and certain important expressions are evaluated. After activation, the Report Writer built-in functions are referenced without error, and all Report Writer executable statements, except BEGIN REPORT, execute without error. The errors returned when report section statements are seen changes when the report is activated. A report remains active until one of the following occurs:

- An END REPORT or STOP REPORT statement executes.
- The report subunit ends or stops.
- A GET statement executes.

Once a report is activated, report output can start. The following statements are the only statements that can start report output:

```
DETAIL LINE
TRIGGER BREAK
TRIGGER PAGE BREAK
END REPORT
```

When report output begins, the following steps take place:

1. The REPORT HEADER section executes to print the report header.
2. If present, the PAGE HEADER section executes to print the page header.
3. Any HEADER sections defined execute from level 1 to level 9, in ascending order.

Before report output starts, all PRINT statements do not affect the report. However, once the report output starts, PRINT statements count as lines in the report.

The following are Report Writer executable statements:

- BEGIN REPORT
- DETAIL LINE
- TRIGGER BREAK
- END REPORT
- STOP REPORT
- TRIGGER PAGE BREAK
- SUPPRESS HEADER
- SUPPRESS TRAILER
- SET PAGENUM

Report Writer Built-in Functions

The Report Writer built-in functions have two main purposes. Some of these functions retrieve information Report Writer has kept for you, such as the automatic totals. Other functions help you control Report Writer flow and output.

Unlike the Report Writer statements, the Report Writer built-in functions are used in subunits other than the one containing the report.

The functions are listed in Table 7-1, along with a brief description. They are defined and explained in chapter 5.

Table 7-1. Report Writer Functions and Returned Values

Function	Description
AVG (Level,Sequence)	Returns the average value of a totaled item.
LASTBREAK	Returns the level number of the last BREAK statement satisfied.
NUMBREAK (Level)	Returns the number of BREAK conditions satisfied for the given level.
NUMDETAIL (Level)	Returns the number of DETAIL LINES with a non-zero *totals_flag* executed for the given level.
OLDCV (Level)	Returns the value of a BREAK WHEN control expression.
OLDCV$(Level)	Returns the value of a BREAK WHEN control expression.
NUMLINE	Returns the number of lines printed on the current page.
PAGENUM	Returns the current page number.

RWINFO (Expression)	Returns various pieces of information that is useful in controlling the Report Writer.
TOTAL (Level, Sequence)	Returns accumulated totals.

Other Statements

The PRINT and PRINT USING statements produce report output after report output has begun. These statements are used in conjunction with the USING clauses of Report Writer statements to generate the report. Before report output begins and when a report is not active, these statements do not affect the report.

System output, such as LIST output and display output using the DISP statement, does not affect output even on the same terminal. This aids in debugging a report.

When error 260 "no lines left on page" occurs, be sure not to use PRINT to display an error message. This causes an infinite loop, because printing the message causes the error again. Instead, use DISP statements or trap on ERROR 260 and trigger a page break first.

The COPY ALL OUTPUT and SEND OUTPUT statements cannot execute once report output begins. These statements can execute for an active report before output starts.

The TAB function of PRINT always works relative to the left margin. If a report specifies a left margin of 10, a

```
     PRINT TAB(10);...
```

moves to the tenth column past the margin (column 19). A TAB(1) moves to the left margin column.

Chapter 8 User-Defined Keys

Introduction

User-definable keys, also called softkeys or programmable function keys, are the eight function keys, f1 - f8, that are on HP terminals. There are nine statements and two functions available in HP Business BASIC/XL that let you define and use these function keys. You have the following two options for specifying the actions to be taken after pressing a user-definable key.

Typing Aid Key - Pressing a key defined as a typing aid key displays strings of characters commonly used for editing or data entry. The attribute field of the function key determines whether the string is executed locally, transmitted to the host computer, or treated in the same manner as the alphanumeric keys.

Branch-During-Input - A branch-during-input key is pressed only when an input statement or READ FORM statement is being executed. The result of pressing the branch-during-input key is a program interrupt followed by resumption of program execution at a point specified in the HP Business BASIC/XL statement defining the key.

The default values for the user-definable keys are blank labels, local execution, and the key definition field set to ASCII character 7, BEL. Pressing a key that has default values rings the terminal's bell.

The type of terminal that you are using is automatically determined when you enter the HP Business BASIC/XL interpreter. A field in the configuration file, HPBBCNFG.PUB.SYS, can be set to specify whether the user-definable keys should be saved when you enter the interpreter so that the values can be restored upon exit. If you selected this option and you encounter problems with the interpreter's ability to save and restore the value of the keys, and you are not using a fully compatible HP terminal as described in Appendix E, set the Is an HP compatible terminal entry in the HP Business BASIC/XL configuration file to N. For more information about setting the HP Business BASIC/XL configuration file, refer to Appendix C.

When the HP Business BASIC/XL interpreter is a batch job, or when BASIN or BASLIST have been redirected, branch-during-input keys are still allowed, but key labels are ignored.

Typing Aid Keys

Typing aid keys set the key definition field to a character string. When you press the key, the stored string is sent to the input device. The key's attribute field determines the manner in which the key is interpreted. For example, consider the situation in which you are in the interpreter's editor, and the key definition field for key 1 is set to the value LIST and the key's attribute field is set to T (indicating the content of the key is to be "transmitted" to the host computer). When you press key 1, the LIST command is displayed on the terminal and subsequently executed. The following statements are related to defining typing aid keys:

- GET KEY - Retrieves the definition of the typing aid keys from a BKEY file.

- SAVE KEY and RESAVE KEY - Stores the current definitions of the typing aid key in a BKEY file.

* SCRATCH KEY - Restores the default key definitions for the terminal.

These statements are defined in chapter 4.

You can define user-definable keys either before or after entering the interpreter. Consult your terminal reference manual for the method used to set the fields for your terminal's user-definable keys.

A field in the configuration file can be set to indicate whether you wish to save the values of the user-definable keys prior to entering the interpreter.

* If the field in the configuration file is set to indicate that the user-definable keys are saved when you enter the interpreter or at the start of a compiled program, then when you execute the first *keys* statement the keys in the terminal is saved. The values of the user-definable keys are restored to the terminal when you exit the program.

* If the field in the configuration file is set to indicate that the values of the user-definable keys are not saved when you enter the interpreter, then the first KEY command except SAVE KEY causes the values of the keys to be set to the default values, blank labels, local, and BEL. Issuing a SAVE KEY command before executing any *keys* statement causes HP Business BASIC/XL to store the current typing aid key definitions.

Key values are retrieved from a file by issuing a GET KEY command. However, when you exit HP Business BASIC/XL with the SAVE KEY option in effect, the previous values are restored as the user-definable key definitions.

Branch-During-Input Keys

By defining a branch-during-input key you provide a method of altering program flow from within an input statement. For example, you can write a help facility that is accessed by pressing a branch-during-input key while the program is executing an input statement. Statements and functions used to define the branch-during-input keys are described below:

* ON KEY and OFF KEY - Activation and deactivation, respectively, of a single key or set of keys defined as branch-during-input keys.

* ENABLE - Specifies that any key-generated branch in the interrupt queue is to be processed. If the queue is empty, branch-during-input keys are processed immediately when pressed.

* DISABLE - Specifies that any key-generated branches are to be added to the interrupt queue without processing.

* PRESS KEY - Allows the simulation of pressing a branch-during-input key from within the program.

* CURKEY - A function that returns the number of the last branch-during-input key pressed.

* RESPONSE - A function that returns how input was terminated, including which softkey was pressed.

These statements are defined in chapter 4. CURKEY and RESPONSE are defined in chapter 5.

Branch-during-input keys are active only during program execution and only when pressed following an input prompt (that is, while INPUT, TINPUT, ACCEPT, or LINPUT statements execute) and before pressing RETURN. They are also active during execution of a READ FORM statement. Any input characters typed between the input prompt and the pressing of the user-definable key defined as a branch-during-input key are lost. Only one branch-during-input key can be pressed during a given input statement.

The resulting branch (GOTO, GOSUB or CALL) to be taken is specified in the ON KEY statement used to define the branch-during-input key. The definition of the branch-during-input key overwrites the current typing aid definition for that key. However, the HP Business BASIC/XL interpreter remembers the last previous typing aid definition for that key. When an OFF KEY statement for that user-definable key is executed, the typing aid definition is restored.

SAVE KEY *fname* and RESAVE KEY *fname* save only the typing aid definitions for the keys. If a key is currently defined as a branch-during-input key, the last previous typing aid definition is written to the file if either of these statements execute. Remember that the last previous typing aid definition is set by either a SAVE KEY, SAVE or RESAVE KEY *fname*, GET KEY *fname*, or SCRATCH KEY.

Priority of Handling the Branch after Pressing Branch-During-Input Keys.

The branching that is performed in response to the ON KEY statement can be considered a restricted interrupt of the normal program flow. As such, the order it is handled in depends on the number of higher priority interrupts that must be handled when the branch-during-input key is pressed. Chapter 4 contains the statements for interrupt handling for DBERROR, EOF (end of file), run-time errors, and HALT (CONTROL Y). The priority for handling these interrupts is:

HALT	16
SHIFT HALT	17
EOF (end of file)	17
run-time errors	17

The priority level for the branch-during-input keys can be set to any integer between 1 and 15, inclusive. If a priority level is not specified in the ON KEY statement, the priority is set to 1. The branches specified by the interrupt handlers and the branch-during-input keys are added to the interrupt queue. The branch with the highest associated priority is processed first. If there is more than one key-generated branch in the interrupt queue with the same priority, the branch resulting from pressing the highest numbered key is processed first.

There are now two conditions to consider:

* If the specified branch is a GOSUB or CALL, then the interrupt queue for the program unit that the key was pressed in is checked immediately following the execution of the RETURN statement that returns control to the calling program unit.

* If the branch is a GOTO, then the statement that is the target of the branch is executed. Following execution of the target statement, the interrupt queue is checked again.

In either case, if the interrupt queue is not empty, then the next branch in the queue with equal priority to that just executed or the branch with the highest remaining priority executes. The process continues until there are no more branches to execute remaining in the queue. At this point, program execution continues at the next executable statement in the program.

If only one GOSUB or CALL branch generated by a branch-during-input key is in the interrupt queue when the ENABLE statement executes, the GOSUB or CALL executes and then execution resumes at the statement following the input statement.

Execution of RUN, STOP, END, SCRATCH PROG, or SCRATCH ALL clears the interrupt queue of any key generated branches remaining to be executed.

The DISABLE statement lets the program add branches to the interrupt queue, but delays execution of the branches. The ENABLE statement allows the handling of queued branch information to continue or begin.

Subunits

The ON KEY CALL statement is active in all subunits called by the subunit that the statement is in unless the user-definable key is redefined within the called subunit. If the key is redefined, the definition on exit from the called subunit is restored to the ON KEY call that it had upon entry. ON KEY GOTO and ON KEY GOSUB are active only within the subunit that they are in. Similarly, an OFF KEY restores the typing aid key definitions to those keys specified only for the subunit that the OFF KEY is in. When you exit from the subunit, the values that the fields of the keys had upon entry to the subunit are restored. If a branch-during-input key is pressed within a compiled subunit called from a program running in the interpreter, the specified branch is added to the interrupt queue and handled when you return to the interpreter.

Using Function Keys in a Batch Job

Function keys can be used in a batch job, or when standard input is taken from a disk file. There are some restrictions, however.

The ON KEY and OFF KEY statements are used normally. However, batch jobs ignore the LABEL specified in the ON KEY statement. Only the action and the priority are used.

HP Business BASIC/XL always looks for keys from terminal input statements, such as INPUT and LINPUT. The file input statements do not expect or examine data for key presses. During batch processing, however, "terminal" input does come from a file, so HP Business BASIC/XL must look for key presses in the standard input. To press a key, you must know how HP Business BASIC/XL recognizes a key.

When input is requested, HP Business BASIC/XL accepts data from the standard input (BASIN) file until the end of the line occurs. A function key check is performed immediately, before any blanks are trimmed from the input line. A function key consists of two characters: an escape character (ASCII 27) followed by a lower case letter between p and w, inclusive. (These are the default terminal definitions and represent function keys 1 to 8 respectively.) To represent a key press, these two characters must appear as the last two characters in the input data. If the escape occurs anywhere else in the input, the sequence is part of the input.

You must exercise caution in creating batch jobs or disk files for HP Business BASIC/XL with key presses. If fixed format files are used, the escape sequence must appear as the last two characters of a record. Otherwise, the escape sequence will not be recognized as a key press. A sample input file might look like:

     The following example shows

```
     Column:             Column:     Remarks:
     0                     ...78
     1                     ...90
     ------------------------------
     this is a test.                  Data to an INPUT statement.
     <esc>p                           This will be taken as data
                                .p    (. represents <esc>)  Press key
```

Chapter 9 Compiler

Introduction

The compiler increases execution speed of programs that have been developed using the interpreter.

The interpreter is an extremely powerful development tool. It facilitates program creation, modification, and debugging by allowing the programmer to stop and start the program at will, examine or change the values of variables at any time, and trace program execution. The price of this power and flexibility is program execution speed.

The compiler produces relocatable object code files that can be linked and executed directly by the operating system. Compiled code executes significantly faster than interpreted code, but it is not easily examined or changed.

This chapter explains the following:

- Compiling and running an HP Business BASIC/XL program.
- Noncompilable statements that require the interpreter environment and therefore do not work in the compiler.
- CWARNINGS command (an interpreter command that lists noncompilable statements).
- Noncompilable program units (main programs or subunits) that must be modified in the interpreter before they can be compiled.
- COPTION and GLOBAL COPTION statements that specify compiler options and directives and are ignored by the interpreter.
- OPTION and GLOBAL OPTION statements in compiled programs.
- That the main program of a compiled program is a procedure rather than an outer block.
- Calling compiled subunits (procedures and functions) from an interpreted program.
- How ON ERROR CALL, ON HALT CALL, and ON END CALL statements behave across compiled subunit calls.

NOTE Not every program unit that can be interpreted can be compiled. Whether a program can be compiled depends on the number and type of statements it contains.

Non-compilable Statements and the CWARNINGS Command

Some HP Business BASIC/XL statements require the interpreter environment and therefore cannot be compiled. Non-compilable HP Business BASIC/XL statements cause compiler warnings. Some statements also generate code that causes a run-time error.

The following statements are effectively ignored by the compiler:

- All trace statements.
- All untrace statements.
- PAUSE statement.

When the compiler encounters one of these statements that are primarily for debugging, it issues a warning message and continues. The compiler does not generate code for the statement that caused the warning.

The following statements cause a run-time error:

COMMAND	GETSUB	RESAVE	SECURE
DEFAULT	LINK	SAVE	
DELETE	MERGE	SCRATCH	

When the compiler encounters one of these statements, it issues a warning message and generates code that causes run-time error #2103. The INTERPRETED built-in function can be used to avoid executing these statements in a compiled program.

The compiler must be able to determine the number of dimensions of every array at compile time. If it encounters an undeclared array or an array parameter for which the dimensions cannot be determined at compile time, for example, an array that appears only in a MAT PRINT statement, the compiler issues an error message. The interpreter command, CWARNINGS, lists noncompilable statements in the current program. The CWARNINGS command is a command-only statement.

Syntax

```
CWARNINGS
```

Non-compilable Program Units

A program unit cannot be compiled unless it is *well-formed*. A *well-formed* program unit has properly matching constructs, such as a NEXT for every FOR, and its array references are consistent with its array declarations.

The interpreter checks a program unit's form before executing or saving it. When a program containing a poorly formed program unit is saved, the interpreter issues a warning message and marks the program unit as noncompilable.

If the programmer attempts to compile the program, the compiler issues the error message

> VERIFY is needed on subunit *program_unit*

and does not generate code for *program_unit*. The compiler cannot diagnose the error; the programmer must return to the interpreter and use the VERIFY command.

COPTION and GLOBAL COPTION Statements

The COPTION and GLOBAL COPTION statements gives you control over the code and listing that the compiler generates.

The GLOBAL COPTION statement is allowed only in the main block of a program. It establishes defaults to be used throughout the program. The COPTION statement can be used in any program unit.

Syntax

```
                 {i_option }[  {i_option }]
[GLOBAL] COPTION {s_option }[, {s_option }]...
```

Parameters

GLOBAL — Allowed only if the statement is in the main block of the program. If GLOBAL appears, the statement is a GLOBAL COPTION statement; if GLOBAL is omitted, it is a COPTION statement. A GLOBAL COPTION statement affects every program unit in the program. A COPTION statement affects only the program unit that contains it.

A COPTION statement overrides a GLOBAL COPTION statement, but only while the program unit that contains it is being compiled or executed.

i_option — One of the in-line options listed in Table 9-2.

s_option — One of the subunit options listed in Table 9-3, with the restriction that USLINIT can appear only in a GLOBAL COPTION statement.

A GLOBAL COPTION statement is allowed only in the main block of a program. It changes the default options in the main program and in every subunit. If two GLOBAL COPTION statements contain opposite options (for example, ID TABLES and NO ID TABLES), the statement with the higher line number sets the option. If a GLOBAL COPTION statement contains opposite options, the rightmost reference sets the option.

A COPTION statement is allowed in the main program and in subunits. It sets program unit options only in the program unit containing it. See Table 9-1 and Table 9-3 for more information.

If two COPTION statements contain opposite options, the statement with the higher line number sets the option. If a COPTION statement contains opposite options, the rightmost reference sets the option.

In-Line verses Program Unit Options

Compiler options take effect in one of two methods: in-line or program unit. In-line options take effect when the COPTION or GLOBAL COPTION statement is processed normally; that is, when the statement is compiled in line number order. They remain in effect until another in-line option changes the setting of the option.

Program unit compiler options are processed before a program unit is compiled. Before the first line of a program unit is compiled, the compiler searches for and processes all of the program unit options. If a COPTION statement does not specify a particular program unit option, the setting of the GLOBAL COPTION statement applies. Program unit options normally apply ONLY to the subunit in which they occur.

The Compiler Options

The compiler options are split into four general categories. Each of the following categories control specific portions of the compilation process:

- Listing.
- Code Space and Performance.
- Data Space.
- Miscellaneous.

Table 9-1. Listing Options

Option 12	Effect	Type	Default
LINES [=] *num_lit* (10 <= *num_lit* <= 9999)	Sets the number of lines per page for the compiler listing.	INLINE	60

{ NOLIST }
LIST { NO LIST}
	Enables and disables compiler source listing and requested tables. ID TABLES and LABEL TABLES can be listed only if listing is enabled.	INLINE	LIST

{ NO ID} ID [TABLES] { NOID } [TABLES]	Prints identifiers, their types and their hexidecimal addresses at the end of each program unit (provided LIST is active). The NOID option suppresses this information.	PROGRAM UNIT	NO ID TABLES

{ NO LABEL}
LABEL [TABLES] { NOLABEL }
[TABLES]
	Prints each program line number and the code offset of the beginning of that line (provided that LIST is also active).
	Suppresses what LABEL TABLES would print.
		PROGRAM UNIT	NO LABEL TABLES

PAGE	Causes page eject. The next line of the compiler listing prints on a new page. (Compare to PAGESUB.)	INLINE	None.
PAGESUB	Generates a page break and page header before the first line of the program unit is printed. If used in a GLOBAL COPTION statement, every program starts on a new page; otherwise, only the current subunit starts on a new page.	PROGRAM UNIT	Program units do not start on new pages.
TITLE [=] *quoted_str_lit*	Replaces the standard HP Business BASIC/XL compiler pagetitle with *quoted_str_lit* at the top of each page of the compiler listing. (Compare to TITLESUB below.)	INLINE	See Note 1.

TITLESUB [=] *quoted_str_lit*	Substitutes *quoted_str_lit* in the page title at the beginning of the subunit.	PROGRAM UNIT	See Note 2.

{ NOWARN }
WARN { NO WARN}
	Enable or suppress compile-time warning messages. The final statistics includes a count of warnings even when warnings are suppressed.	INLINE	WARN

Table 9-1 Notes

1 An HP Business BASIC/XL compiler page header consists of a page number followed by a page title; For example:

```
     HP Business BASIC/XL Compiler  HP32715A.00.00  Copyright Hewlett-Packard Co.
1989
       SUN, JAN 1, 1989, 2:01 PM
```

2 If included in a subunit, HP Business BASIC/XL replaces the page title in the page header with *quoted_str_lit* on the next page break. The difference between TITLE and TITLESUB is that with the latter, the title change takes place the instant a subunit is entered. Thus if there are any page breaks within the subunit before the COPTION TITLESUB = "*quoted_str_lit* " statement, the new title is in effect. With COPTION TITLE = "*quoted_str_lit* ", the title will not change until after the actual statement.

Table 9-2. Code and Performance COPTIONS

Option	**Meaning**	**Type & Defaults**	**Effects on Compiled Program**
ERROR [HANDLING] { NOERROR } { NO ERROR} [HANDLING]	Emits or suppresses code to trap errors. When NOERROR is in effect, an ON ERROR statement causes a compile-time error. If a run-error occurs (with NO ERROR), HP Business BASIC/XL prints an error message. If the compiled program was called from the interpreter, control returns to the interpreter; otherwise, the program aborts.	PROGRAM UNIT ERROR HANDLING	NO ERROR saves approximately 3 words per line. (Not all statements perform ERROR checking); performance increases also.
HALT [CHECKING] { NOHALT } { NO HALT} [CHECKING]	Emits or suppresses code to check for the HALT key at the end of each line.	INLINE HALT checking	NO HALT saves approximately 3 words per line. (Not all statements perform HALT checking); performance increases also.

{ 0}
OPTIMIZE { 1}
	Level of optimization.	Subprogram	0 = No optimization.
			1 = Local optimization.
			The default is 1.

RANGE [CHECKING] { NO RANGE} { NORANGE } [CHECKING]	Emit or suppress code that causes a run-time error when one of the following occurs: 1. An array index or a substring index is out of bounds. 2. An integer to short integer conversion overflows. 3. The nested GOSUB level is greater than the default MAXGOSUB level or the value specified in COPTION MAXGOSUB. 4. The file number used is greater than the default MAXFILES or the MAXFILES value in the COPTION MAXFILES.	INLINE RANGE CHECKING	Code savings vary, but option NORANGE can save 12 to 16 words per array or substring access.
REDIM NO REDIM	Allows or disallows array redimensioning. Not allowing dimensions to change allows for more compile-time and less run-time checking of array bounds. GLOBAL COPTION [NO] REDIM affects arrays in COM.	PROGRAM UNIT REDIM	Reduces code for array access (unless array is variably dimensioned). Performance improves corresponding to code reduction.

Table 9-3. Dataspace COPTIONS

Option	**Meaning**	**Type**	**Default**
MAXFILES [=] *num_lit*	Specifies the largest file number used in this subunit. Each invocation of a subunit allocates 1 word for each legal file number.	Program Unit	16
MAXGOSUBS [=] *num_lit*	Allows GOSUB statements to be nested to a depth of *num_lit*. A run-time	None	10

	error occurs if more than *num_lit* GOSUB statements execute before a RETURN (in one subunit). Each invocation of a subunit allocates one word for each possible GOSUB.		

Table 9-4. Other COPTIONS

Option	**Meaning**	**Type**	**Default**
COPYRIGHT=*quoted_str*	Allows you to insert a copyright statement in the program. No effect on program execution.	Program Unit	none
RLFILE	Allows you to compile each of the main block and each subunit as a separate object file into an RL file. You can compile more than one program subunit into an RL.	Program	Compile current program as individual object file into an NMOBJ file.
RLINIT	Directs the compiler to initialize the RL before compiling code into it.	Program	RL not initialized.
LOCALITY= *quoted_str*	Allows you to group multiple object modules into a locality set when they are compiled into an RL. This will help in maintaining and using RL commands such as PURGERL.	Program Unit	No locality set specified.

Table 9-4 NOTE For a detailed description of relocatable libraries (RLs) see the *HPLINK EDITOR/XL Reference Manual*.

COPTION and GLOBAL COPTION Statements

The COPTION and GLOBAL COPTION statements gives you control over the code and listing that the compiler generates.

The GLOBAL COPTION statement is allowed only in the main block of a program. It establishes defaults to be used throughout the program. The COPTION statement can be used in any program unit.

Syntax

```
                 {i_option }[  {i_option }]
[GLOBAL] COPTION {s_option }[, {s_option }]...
```

Parameters

GLOBAL   Allowed only if the statement is in the main block of the program. If GLOBAL appears, the statement is a GLOBAL COPTION statement; if GLOBAL is omitted, it is a COPTION statement. A GLOBAL COPTION statement affects every program unit in the program. A COPTION statement affects only the program unit that contains it.

A COPTION statement overrides a GLOBAL COPTION statement, but only while the program unit that contains it is being compiled or executed.

i_option One of the in-line options listed in Table 9-2.

s_option One of the subunit options listed in Table 9-3, with the restriction that USLINIT can appear only in a GLOBAL COPTION statement.

A GLOBAL COPTION statement is allowed only in the main block of a program. It changes the default options in the main program and in every subunit. If two GLOBAL COPTION statements contain opposite options (for example, ID TABLES and NO ID TABLES), the statement with the higher line number sets the option. If a GLOBAL COPTION statement contains opposite options, the rightmost reference sets the option.

A COPTION statement is allowed in the main program and in subunits. It sets program unit options only in the program unit containing it. See Table 9-1 and Table 9-3 for more information.

If two COPTION statements contain opposite options, the statement with the higher line number sets the option. If a COPTION statement contains opposite options, the rightmost reference sets the option.

In-Line verses Program Unit Options

Compiler options take effect in one of two methods: in-line or program unit. In-line options take effect when the COPTION or GLOBAL COPTION statement is processed normally; that is, when the statement is compiled in line number order. They remain in effect until another in-line option changes the setting of the option.

Program unit compiler options are processed before a program unit is compiled. Before the first line of a program unit is compiled, the compiler searches for and processes all of the program unit options. If a COPTION statement does not specify a particular program unit option, the setting of the GLOBAL COPTION statement applies. Program unit options normally apply ONLY to the subunit in which they occur.

The Compiler Options

The compiler options are split into four general categories. Each of the following categories control specific portions of the compilation process:

* Listing.
* Code Space and Performance.
* Data Space.
* Miscellaneous.

Table 9-1. Listing Options

Option 12	Effect	Type	Default
LINES [=] *num_lit* (10 <= *num_lit* <= 9999)	Sets the number of lines per page for the compiler listing.	INLINE	60

{ NOLIST }
LIST { NO LIST}
	Enables and disables compiler source listing and requested tables. ID TABLES and LABEL TABLES can be listed only if listing is enabled.	INLINE	LIST

{ NO ID} ID [TABLES] { NOID } [TABLES]	Prints identifiers, their types and their hexidecimal addresses at the end of each program unit (provided LIST is active). The NOID option suppresses this information.	PROGRAM UNIT	NO ID TABLES

{ NO LABEL}
LABEL [TABLES] { NOLABEL }
[TABLES]
	Prints each program line number and the code offset of the beginning of that line (provided that LIST is also active).
	Suppresses what LABEL TABLES would print.
		PROGRAM UNIT	NO LABEL TABLES

PAGE	Causes page eject. The next line of the compiler listing prints on a new page. (Compare to PAGESUB.)	INLINE	None.
PAGESUB	Generates a page break and page header before the first line of the program unit is printed. If used in a GLOBAL COPTION statement, every program starts on a new page; otherwise, only the current subunit starts on a new page.	PROGRAM UNIT	Program units do not start on new pages.
TITLE [=] *quoted_str_lit*	Replaces the standard HP Business BASIC/XL compiler pagetitle with *quoted_str_lit* at the top of each page of the compiler listing. (Compare to TITLESUB below.)	INLINE	See Note 1.
TITLESUB [=] *quoted_str_lit*	Substitutes *quoted_str_lit* in the page title at the beginning of the subunit.	PROGRAM UNIT	See Note 2.
{ NOWARN }	Enable or suppress compile-time warning messages. The final	INLINE	WARN

WARN { NO WARN}	statistics includes a count of warnings even when warnings are suppressed.		

Table 9-1 Notes

1 An HP Business BASIC/XL compiler page header consists of a page number followed by a page title; For example:

```
        HP Business BASIC/XL Compiler  HP32715A.00.00  Copyright Hewlett-Packard Co.
1989
          SUN, JAN 1, 1989, 2:01 PM
```

2 If included in a subunit, HP Business BASIC/XL replaces the page title in the page header with *quoted_str_lit* on the next page break. The difference between TITLE and TITLESUB is that with the latter, the title change takes place the instant a subunit is entered. Thus if there are any page breaks within the subunit before the COPTION TITLESUB = "*quoted_str_lit* " statement, the new title is in effect. With COPTION TITLE = "*quoted_str_lit* ", the title will not change until after the actual statement.

Table 9-2. Code and Performance COPTIONS

Option	**Meaning**	**Type & Defaults**	**Effects on Compiled Program**
ERROR [HANDLING] { NOERROR } { NO ERROR} [HANDLING]	Emits or suppresses code to trap errors. When NOERROR is in effect, an ON ERROR statement causes a compile-time error. If a run-error occurs (with NO ERROR), HP Business BASIC/XL prints an error message. If the compiled program was called from the interpreter, control returns to the interpreter; otherwise, the program aborts.	PROGRAM UNIT ERROR HANDLING	NO ERROR saves approximately 3 words per line. (Not all statements perform ERROR checking); performance increases also.
HALT [CHECKING] { NOHALT } { NO HALT} [CHECKING]	Emits or suppresses code to check for the HALT key at the end of each line.	INLINE HALT checking	NO HALT saves approximately 3 words per line. (Not all statements perform HALT checking); performance increases also.

{ 0}
OPTIMIZE { 1}
	Level of optimization.	Subprogram	0 = No optimization.
			1 = Local optimization.
			The default is 1.

RANGE [CHECKING] { NO RANGE} { NORANGE } [CHECKING]	Emit or suppress code that causes a run-time error when one of the following occurs: 1. An array index or a substring index is out of bounds. 2. An integer to short integer conversion overflows. 3. The nested GOSUB level is greater than the default MAXGOSUB level or the value specified in COPTION MAXGOSUB. 4. The file number used is greater than the default MAXFILES or the MAXFILES value in the COPTION MAXFILES.	INLINE RANGE CHECKING	Code savings vary, but option NORANGE can save 12 to 16 words per array or substring access.
REDIM NO REDIM	Allows or disallows array redimensioning. Not allowing dimensions to change allows for more compile-time and less run-time checking of array bounds. GLOBAL COPTION [NO] REDIM affects arrays in COM.	PROGRAM UNIT REDIM	Reduces code for array access (unless array is variably dimensioned). Performance improves corresponding to code reduction.

Table 9-3. Dataspace COPTIONS

Option	**Meaning**	**Type**	**Default**
MAXFILES [=] *num_lit*	Specifies the largest file number used in this subunit. Each invocation of a subunit allocates 1 word for each legal file number.	Program Unit	16
MAXGOSUBS [=] *num_lit*	Allows GOSUB statements to be nested to a depth of *num_lit*. A run-time error occurs if more than *num_lit* GOSUB statements execute before a RETURN (in one subunit). Each invocation of a subunit allocates one word for each possible GOSUB.	None	10

Table 9-4. Other COPTIONS

Option	Meaning	Type	Default
COPYRIGHT=*quoted_str*	Allows you to insert a copyright statement in the program. No effect on program execution.	Program Unit	none
RLFILE	Allows you to compile each of the main block and each subunit as a separate object file into an RL file. You can compile more than one program subunit into an RL.	Program	Compile current program as individual object file into an NMOBJ file.
RLINIT	Directs the compiler to initialize the RL before compiling code into it.	Program	RL not initialized.
LOCALITY= *quoted_str*	Allows you to group multiple object modules into a locality set when they are compiled into an RL. This will help in maintaining and using RL commands such as PURGERL.	Program Unit	No locality set specified.

Table 9-4 NOTE For a detailed description of relocatable libraries (RLs) see the *HPLINK EDITOR/XL Reference Manual*.

OPTION and GLOBAL OPTION Statements

The OPTION and GLOBAL OPTION statements set options for interpreted programs (see chapter 2). The compiler ignores the options these statements set, with the exceptions given in Table 9-5.

Table 9-5. Interpreter Options That Affect Compiled Programs

Option	Effect on Compiled Program
DECIMAL REAL BASE 0 BASE 1 INIT	Same as effect on interpreted program.
NOINIT	The compiler does not generate code to initialize numeric variables (string lengths are still initialized to zero). If the compiled

	program accesses a variable before assigning a value to it, no error occurs, but the value of the variable is indeterminate.
MAIN	Defines this as the main program of a multi-program application. Outer block is generated.
SUBPROGRAM	Identifies this program as a module of a multi-program application. Code is produced for the main subunit, but no outer block is generated.
NEWCOM	The compiled main subunit deallocates undefined COM blocks when called, and allocates new defined commons.
NO NEWCOM NONEWCOM	The compiler only generates code to check common name and sizes. Commons are not allocated or deallocated. When used with option MAIN, code for initial allocation goes in the outer block instead of the code for the main subunit.

The defaults are: REAL, BASE 0, INIT, MAIN, NEWCOM

Examples

```
  10 GLOBAL COPTION LABEL TABLES, ID TABLES
  20 INTEGER I
  30 DIM Deck(52), Suit$(4)[8], Ranks$(13)[6]
  35 TRACE VARS Deck
  40 COPTION TITLE="Start of initialization", PAGE
  50 Suit$(1)="spades"
      :
 200 COPTION NORANGE CHECKING
 210 FOR I=1 TO 62
 220    Deck(I)=I
 230 NEXT I
      :
1000 DEF FNPrint$(INTEGER Row, Col, S$)
1010 COPTION TITLESUB="Function FNPrint",PAGESUB,NOWARN,NOLABEL TABLES
1015 PAUSE
1020 Move_to(Row, Col)
1030 RETURN S$
1090 FNEND
1095 !***** Subprogram to move the cursor.
2000 SUB Move_to (INTEGER Row, Col)
2005 COPTION TITLESUB="Subprogram Move_to", PAGESUB
2010 PRINT '27"&a";VAL$(Row);"r";VAL$(Col);"C";
2020 SUBEND
```

The compiler listing for the above program is:

```
PAGE 1 HP Business BASIC/XL Compiler HP32115B.00.00 (c) Hewlett-Packard
1985-1987  TUE, AUG 25, 1987, 11:19 AM

   10 GLOBAL COPTION LABEL TABLES, ID TABLES
   20 INTEGER I
   30 DIM Deck(52), Suit$(4)[8], Ranks$(13)[6]
   35 TRACE VARS Deck
WARNING 2050: TRACE or PAUSE statement found and ignored.

   40 COPTION TITLE="Start of initialization", PAGE

PAGE 2  Start of initialization
```

```
     50 Suit$(1)="spades"
     60 Suit$(2)="hearts"
     70 Suit$(3)="clubs"
     80 Suit$(4)="diamonds"
       .
       .
       .
    200 COPTION NORANGE CHECKING
    210 FOR I=1 TO 62
    220    Deck(I)=I
    230 NEXT I
        .
        .
        .
                     I D E N T I F I E R   T A B L E

IDENTIFIER          CLASS               TYPE               ADDRESS

Deck                ARRAY               REAL               SP- $774,I
I                   SIMPLE              INTEGER            SP- $778
Ranks$              ARRAY               STRING             SP- $76C,I
Suits$              ARRAY               STRING             SP- $770,I

PAGE 3 function fnprint

 1000 DEF FNPrint$(SHORT INTEGER Row, Col, S$)
 1010 COPTION TITLESUB="Function FNPrint",PAGESUB,NO WARN,NO LABEL TABLES
 1015 PAUSE
 1020 Move_to(Row, Col)
 1030 RETURN S$
 1090 FNEND
 1095 !***** Subprogram to move the cursor.
                     I D E N T I F I E R   T A B L E

IDENTIFIER          CLASS               TYPE               ADDRESS

Col                 SIMPLE PARAMETER    INTEGER            PSP-$28,I
FNPrint$            FUNCTION            STRING
Move_to             SUBPROGRAM
Row                 SIMPLE PARAMETER    INTEGER            PSP-$24,I
S$                  SIMPLE PARAMETER    STRING             PSP-$2C,I

PAGE 4  move_to

 2000 SUB Move_to (INTEGER Row, Col)
 2005 COPTION TITLESUB="Subprogram Move_to", PAGESUB
 2010 PRINT '27"&a";VAL$(Row);"r";VAL$(Col);"C";
 2020 SUBEND
                     I D E N T I F I E R   T A B L E

IDENTIFIER          CLASS               TYPE               ADDRESS

Col                 SIMPLE PARAMETER    INTEGER            PSP-$28,I
Move_to             SUBPROGRAM
Row                 SIMPLE PARAMETER    INTEGER            PSP-$24,I
                           C O D E   O F F S E T S

LINE   OFFSET   LINE   OFFSET   LINE   OFFSET   LINE   OFFSET   LINE   OFFSET

                                  MAIN

  10=000001EC     20=000001EC     30=000001EC     35=000001EC     40=000001EC
   50=000001EC     60=00002A8      70=00000364     80=00000420      90=000004DC
 200=00000594    210=00000594    220=000005C4    230=000005FC
```

```
                                   FNPrint$

     1000=000000D4   1010=000000D4   1015=000000D4   1020=000000D4   1030=00000100
     1090=00000118   1095=00000150

                                   move_to

     2000=000000D0   2005=000000D0   2010=000000D0   2020=000002C0

     Number of errors = 0       Number of warnings = 2
     Processor time = 00:00:01  Elapsed time = 00:00:02
     Number of lines = 26       Lines / CPU minute = 1560.0

     END OF PROGRAM
```

Notes on the Example

LINE **COMMENT**

---

10 Makes LABEL TABLES and ID TABLES the default throughout the program.

40 Changes the title and starts a new page.

200 Turns off range checking in the lines that follow. Because the FOR loop limit value is mistyped (62 when it should have been 52), the result of line 220 is unpredictable.

1010 Sets the title, prints the first line of the function on a new page, suppresses compile-time warning messages, and suppresses the label table for the current subunit.

The PAUSE statement on line 1015 causes a compile-time warning. Although there is no warning message, the final statistics reflect the warning.

1095 This line illustrates the problem of putting comments before the subunit to which they apply. Although an interpreter listing would look right, the comment at line 1095 actually belongs to the function FNPrint$. Therefore, in the compiler listing, it appears before the identifier map, the code offsets table, and the page break that the PAGESUB option causes.

Compiling and Running Programs

You can run the compiler from the interpreter or from the operating system. The commands are slightly different, but the steps are the same. Figure 9-1 shows how a BSVXL file becomes an executing program (files are boxed and steps are in capital letters).

```
                 |--------------------------|
                 |                          |
                 |       BASIC SAVE file    |
                 |       (BSVXL)            |
                 |--------------------------|
                              |
                              v
                      COMPILATION STEP
                              |
                              v
                 |--------------------------|
                 |                          |
                 | Relocatable Object file  |
                 | (NMOBJ or RL)            |
                 |--------------------------|
                              |
                              v
                          LINK STEP
```

```
                        |
                        v
           |--------------------------|
           |                          |
           |     NMPROG file          |
           |     (NMPRG)              |
           |--------------------------|
                        |
                        v
                    RUN STEP
                        |
                        v
           |--------------------------|
           |                          |
           |     Executing            |
           |     Program              |
           |                          |
           |--------------------------|
```

Figure 9-1. Steps to Compile and Run a Program

The compilation step translates a BASIC SAVE file into object code, machine instructions in binary form, and stores those instructions in a relocatable object module in a specially formatted disk file with file code NMOBJ. See the *HPLink Editor/XL Reference Manual* for more information about linking object files.

The running step binds the program to referenced externals from an Executable Library, moves the program and its data stack into main memory, and initiates execution.

NOTE The difference between a BASIC SAVE file and a program file is important. A BASIC SAVE file contains HP Business BASIC/XL program source code in a special format. The GET command can make that program the current program in the interpreter. A program file contains executable program code and runs under the operating system.

The compiling, linking, and running steps can be performed individually, or the first step and successive steps can be performed with a single command.

Table 9-6 gives the syntax of every command that performs one or more steps of the compilation process. It also gives the type of the default operating system file $OLDPASS if the command is successful. (See the *MPE XL Commands Reference Manual* for more information on $OLDPASS, and the :RUN MPE XL command.) Explanations of the command parameters follow Table 9-6.

Table 9-6. Compilation Process Commands

Step(s)	**Command from Interpreter**	**Command from Operating System**	**Effect on $OLDPASS**
Compilation	COMPILE [*infile*] { ,} [{ ;} OBJ[=]*objfile*] [ ]	:BBXLCOMP [text =] *infile* [,[obj=] *objfile* [,[list=] *listfile*]]	If *objfile* is not specified and command succeeds, then $OLDPASS is the NMOBJ relocatable object file.

	[{ ,}] [{ ;} LIST[=]*listfile*]		
Compilation Linking	COMPLINK [*infile*] { ,} [{ ;} PROG[=]*progfile*] [ ] [{ ,}] [{ ;} LIST[=]] [*listfile*]	:BBXLLK [text =] *infile* [,[prog=] *progfile* [,[list=] *listfile*]]	If *progfile* is not specified and command succeeds, then $OLDPASS is the program file.
Compilation Linking Running	COMPGO [*infile*] [{ ,}] [{ ;} LIST[=]*listfile*]	:BBXLGO [text =] *infile* [,[list=] *listfile* [,[xl=] *xlfile*]]	If the command succeeds, then $OLDPASS is the program file.
Linking	Use the SYSTEM command.	:LINK FROM=*objfile* TO=*progfile* [;cap=*cap_list*]	None
Running	SYSTEMRUN *progfile* [;xl=‘xlfile']	:RUN *progfile* [;xl='xlfile']	None

Table 9-6 Notes

1. COMPLINK and COMPGO link the program with the default capabilities of IA and BA, and any additional capabilities MR, DH, and PH that are consistent with the capabilities of the user.

2. *infile, objfile, listfile,* and *progfile* are *fname* s (see chapter 6).

3. OBJ, LIST, and PROG can be in any order.

4. All but SYSTEMRUN are command-only statements.

Command Parameters

Each of the following parameters is a file name, but its form depends on command type. A file name can be any string expression in an interpreter command. A file name must be a valid, unquoted file name in an operating system command.

Parameters

infile Name of BASIC SAVE source file containing HP Business BASIC/XL program. If *infile* is not a NMBSV file, an error occurs and compilation terminates.

This parameter is required in operating system commands, but is optional in interpreter commands. The default *infile* for an interpreter command is the current program, automatically saved in a temporary NMBSV file.

objfile Name of binary file that the compiler writes the object code into. It can be either an NMOBJ or an NMRL file. The default *objfile* is $OLDPASS if it exists and is type NMOBJ; otherwise, it is $NEWPASS. If the system uses $NEWPASS as *objfile*, it changes the name of the file to $OLDPASS when it closes it.

To create a new, permanent NMOBJ file, do any one of the following:

* Specify a filename that is not in a directory as this parameter. The operating system creates a permanent NMOBJ file of the correct size and type.

* $OLDPASS is the NMOBJ file, save it to the permanent file space with the operating system command :SAVE $OLDPASS, *filename*.

* Build an NMOBJ file with the link editor command, BUILDRL.

* Build a file with the operating system command :BUILD, using filecode parameter NMOBJ.

To create a new, permanent NMRL file, do any one of the following:

* Build an NMRL file with the link editor command, BUILDRL.

* Build a file with the operating system command :BUILD, using filecode parameter NMRL.

listfile Name of ASCII file that the compiler writes the compiler listing into. The default *listfile* is $STDLIST. *listfile* can be the terminal or a spoolfile.

progfile Name of binary file that the link editor writes the program into. The default *progfile*, $NEWPASS, is renamed $OLDPASS when closed.

To create a new, permanent program file, do any one of the following:

* Specify a filename that is not in a directory as this parameter. The system creates a temporary file of the correct size and type.

* If $OLDPASS is the NMPRG file, save it to the permanent file space with the operating system command :SAVE $OLDPASS, *filename*.

* Build a new file with the operating system command :BUILD, using filecode parameter NMPRG.

If *progfile* exists, the operating system reuses it. An error occurs if *progfile* is too small, or if its filecode is not NMPRG.

Main Program Procedure

The main program of a compiled program is not an outer block, but a procedure. The outer block of a compiled HP Business BASIC/XL program initializes the program and then calls the main program. The name of the outer block is always BB_PROGRAM. The name of the main program procedure is the upshifted name of the file that contains the program source code; for example, if the filename of the file containing the program is MYPROG.MYGROUP, the main program procedure name is MYPROG.

The main program procedure can be put into an executable library and called as an external subunit. If you compile a program from within the interpreter without specifying the name of the file, the name of the current program will be the name of the main procedure entry point. If the current program does not have a name, BBCINP will be the entry point name. See "Calling External Subunits from Interpreter" later in this chapter for more details.

Calling Compiled Subunits From the Interpreter

An interpreted program can call a compiled subunit under the following conditions:

- The compiled subunit and any subunits that it calls must be in an executable library. Use the link editor to add the relocatable object file to an executable library. See the *HPLink Editor/XL Reference Manual* for details. The interpreter can be run using the XL parameter to specify which executable library to search.

- The interpreted program unit must contain a definition of the compiled subunit. The subunit uses an EXTERNAL, INTRINSIC or ANYPARM statement, or it can be implicitly declared in an ANYPARM, underbar, or call.

An external subunit call is syntactically identical to an internal subunit call. The CALL statement calls an external procedure, and an external multi-line function call is legal wherever an internal multi-line function is legal.

The ON ERROR CALL, ON HALT CALL, and ON END CALL statements cannot reference external subunits.

An external subunit cannot call an interpreted subunit.

On Call Statements and Compiled Subunits

The following example illustrates the behavior of the ON ERROR CALL statement across compiled subunit calls. The ON HALT CALL and ON END CALL statements behave the same way.

Examples

```
            Interpreted Program                            External Subunits
-------------------------------------------------------------------------------------

     10 ON ERROR CALL Errsub                      100 SUB Extproc
      20 GLOBAL EXTERNAL Extproc                  110 ON ERROR CALL Blob
      30 CALL First_sub                           120 CALL Squiggle
            .                                            .
            .                                            .
            .                                            .
     500 SUB First_sub                            400 SUBEND
            .                                     410 SUB Squiggle
            .                                     420 PRINT 1/0
            .                                     430    .
     600 CALL Extproc                                    .
     610 PRINT 1/0                                       .
            .                                     500 SUBEND
            .                                     510 SUB Blob
            .                                            .
     700 SUBEND                                          .
     720 SUB Errsub                                      .
            .                                     590 SUBEND
            .
            .
     800 SUBEND
```

Line	Result of Executing Line
10	Errsub will handle errors.
600	Control transfers to Extproc. Now Errsub does not handle errors.
110	Blob will handle errors.
420	Error occurs (division by zero). Control transfers to Blob.
590	Control returns to line 430.
400	Control returns to First_sub (line 610). Now Errsub handles errors again.
610	Error occurs (division by zero). Control transfers to Errsub.

Appendix A Error Messages

Numbered Error Messages (2 - 216)

---

2 MESSAGE 1. Memory overflow.
2. Not enough data space available for the local variables.

CAUSE The system does not have enough memory to run the program. Most probable case is that the program is too big or too many variables are used in the program. Message 1 comes out at run time and message 2 comes out as a verify error.

ACTION Break the program up into smaller subprograms or use fewer variables.

---

3 MESSAGE 1. Line not found, or not in current program unit.
2. Renumbering label specified is not in the current program unit.
3. Execution line specified is not in the current program unit.

CAUSE 1. The line referenced is not found within the program unit being executed.

2. This occurs in the interpreter only. The line label specified as the first line to be read from an ASCII program file for a GET, LINK, or MERGE does not exist in the file.

3. Interpreter only: The parameter to the RUN command specifying the line on which execution is to begin is not within the current program. For example, if the current program has no lines then RUN; 10 will generate this error.

ACTION Be certain that the line number specified exists in the current program.

---

4 MESSAGE RETURN without GOSUB.

CAUSE A RETURN statement is encountered without a corresponding GOSUB statement.

ACTION Delete the statement or check the program for errors.

---

5 MESSAGE Encountered end of function before RETURN was executed.

CAUSE A user-defined multi-line function does not end in a RETURN statement.

ACTION Add a RETURN statement.

---

6 MESSAGE 1. FOR without NEXT.
2. No FOR loop active for this NEXT.
3. Illegal imbedded FOR loop variable.

CAUSE 1. Interpreter message only; a FOR loop is not ended by a corresponding NEXT. This usually happen in an embedded FOR loop.

2. In the interpreter, error occurs because execution of a NEXT in a FOR..NEXT construct does not have a corresponding active FOR. The situation occurs when a branch is made to the middle of a FOR..NEXT construct without execution of the corresponding FOR statement. This message also appears in compiled programs with COPTION RANGE CHECKING selected. This error occurs because execution of a NEXT in a FOR..NEXT construct does not have a corresponding active FOR. The situation occurs when a branch is made to the middle of a FOR..NEXT construct without execution of the corresponding FOR statement.

3. The same variable is being used as a FOR loop control variable for an outer and inner FOR loop in a nested FOR loop in one of the following statements: READ, READ FORM, WRITE FORM, PRINT DISPLAY, PRINT USING, INPUT, or READ #"fnum".

For example, 10 READ (FOR I=1 TO 3,(FOR I=4 TO 7,A(I,I)))

ACTION Check program for FOR loop errors, particularly branching into the middle of FOR loop. Each of the loop control variables in a nested FOR construct in one of the above statements must have a distinct name.

---

7 MESSAGE 1. Attempt to call an undefined subprogram or function.
2. Subunit "subunit_name" does not exist.
3. Attempt to FNCALL a non-existent function.

CAUSE 1. The function name in a CALL statement is not defined in the current program or an EXTERNAL statement is missing.

2. The procedure or function in the current program named "subunit_name" does not exist. "subunit_name" could have been specified in any statement that requires a line range, for example, LIST, TRACE statements, CHANGE, and VERIFY.

3. The FNCALL was made to an external function that does not exist.

ACTION Define the function or subunit.

---

8 MESSAGE 1. Improper parameter matching of parameter #N.
2. Improper parameter matching with VAR.

CAUSE The formal parameter defined in the EXTERNAL statement or retrieved from the intrinsic file does not match the type of the corresponding actual parameter in the parameter list for the call. Depending on the cause, one of the two above messages will appear; N is the parameter number and VAR is the name of the formal parameter that is mis-matched.

ACTION Replace the actual parameter in the call to the external with a variable of the correct type. The type of the variable can also be coerced with one of the built-in functions: REAL, SREAL, INTEGER, SINTEGER, or DECIMAL. Note that doing so will result in an expression being passed as the actual parameter. Expressions are evaluated at the time of the call and the value is assigned to a temporary cell. The temporary cell becomes the actual parameter.

---

9 MESSAGE Improper number of parameters.
CAUSE The number of parameters passed to a function in a CALL statement does not match the number of parameters specified in the function itself.

ACTION Check the function definition and add or delete the appropriate parameter.

---

12 MESSAGE Attempt to redeclare a variable, VAR, in line N.

CAUSE An already declared variable VAR is being redeclared in line N.

ACTION Only declare the variable once.

---

15 MESSAGE Invalid bounds on array dimension or string length.

CAUSE Strings or arrays are declared with an invalid bound. E.g. DIM A$[0]

ACTION Change the declaration to use a valid bound.

---

16 MESSAGE Improper array dimensions.

CAUSE 1. An array is defined with an invalid dimension or defined with more than 6 dimensions. For example, DIM A(-1).

2. A MAT statement detected that one of the operands is not properly dimensioned for that particular matrix operation. Example : MAT A = B * C where B is not dimensioned the same as C.

ACTION Redefine the array with a valid dimension.

---

17 MESSAGE Subscript out of range.

CAUSE When an array element is referenced, the subscript is outside the range of the array. For example, accessing element 3 of a 2-element array.

ACTION Check subscript for correct range.

---

18 MESSAGE 1. Substring out of range or substring too long.
2. Input string too long.

CAUSE 1. This error is most often caused by incompatible string or substring assignment. For example, assigning a string to another string of shorter length, or referencing a string beyond its actual length.

2. Trying to input A string that is longer than the max length of the receiving variable has been input.

ACTION 1. Make sure the string being referenced is within its actual and maximum length.

2. Input a shorter string, or use substring input.

---

19 MESSAGE 1. Improper value in program statement.
2. Computed GOTO/GOSUB expression out of range.

CAUSE Message 1 : An improper value is detected when evaluating binary built-in functions BITLR and BITRL.

Message 2 : An improper value is detected when executing the ON X GOTO/GOSUB statement. For example, X contains a value that is outside of its intended range.

ACTION Check program for correct value.

---

20 MESSAGE 1. SHORT INTEGER precision overflow.
2. Numeric expression for CAUSE ERROR must be in SHORT INTEGER

range.

CAUSE 1. Most often caused by assigning to a short integer an expression whose value is beyond a short integer (16 bit) range of (-32768, 32767).

2. The value supplied in a CAUSE ERROR statement is beyond the range of a short integer.

ACTION Check the expression for accuracy or use another data type.

----

21 MESSAGE SHORT DECIMAL precision overflow.

CAUSE Most often caused by assigning an expression whose value is beyond the range of a legal short decimal.

ACTION Check the expression for accuracy or use another data type.

----

22 MESSAGE DECIMAL precision overflow.

CAUSE Most often caused by assigning an expression whose value is beyond the range of a decimal quantity.

ACTION Check the expression for accuracy or use another data type.

----

24 MESSAGE TAN(N*PI/2) when N is odd.

CAUSE The argument of a TAN function will produce a result of infinity.

ACTION Check argument for accuracy.

----

25 MESSAGE Argument of ASN or ACS is >1 in absolute value.

CAUSE The call to the mathematical functions ARC SINE and ARC COSINE contains an invalid argument.

ACTION Check argument for accuracy.

----

26 MESSAGE Zero to negative power

CAUSE A zero raised to a negative power results in infinity, an invalid quantity.

ACTION Check the expression for accuracy.

----

27 MESSAGE Negative to nonintegral power

CAUSE A negative number raised to a nonintegral power will result in a complex number which is not supported by Business BASIC.

ACTION Check the expression for accuracy.

----

28 MESSAGE Argument of LOG or LGT is negative.

CAUSE The call to the log functions contains an invalid argument, in this case a negative number.

ACTION Check the argument for accuracy.

----

29 MESSAGE Argument of LOG or LGT is 0.

CAUSE The call to the log functions contains an invalid argument, in this case a zero quantity.

ACTION Check the argument for accuracy.

----

30 MESSAGE Argument of SQR is negative.

CAUSE The square root function SQR is called with a negative argument.

ACTION Check the argument for accuracy.

----

31 MESSAGE Division by zero, or modulo zero.

CAUSE A division by zero is detected during the evaluation of an expression.

ACTION Check the expression for accuracy.

----

32 MESSAGE String not a valid number, or string where numeric data required.

CAUSE The string parameter to the VAL function does not contain a number.

ACTION Check parameters for accuracy.

----

33 MESSAGE 1. Bad argument to the NUM function.
2. RPT$ number of repetitions must be zero or greater.
3. Repeated string would exceed the maximum string length.

CAUSE Message 1 : The string parameter to the NUM function contains an invalid character.

Message 2 : The repeat count parameter of the RPT$ function contains an invalid count; probably negative.

Message 3 : The result length of the repeated string exceeds the maximum limit for a string variable.

ACTION Check arguments for accuracy.

----

34 MESSAGE Line referenced is not an IMAGE statement

CAUSE The line referenced by a PRINT USING statement is not an IMAGE statement.

ACTION Add the required IMAGE statement.

----

35 MESSAGE Improper IMAGE format specification, character N.

CAUSE Character N of the IMAGE statement contains an invalid format specification.

ACTION Modify the wrong format with a valid specification.

----

36 MESSAGE Out of data.

CAUSE The READ statement tries to read more than what is contained in the DATA statements.

ACTION Use RESTORE statement to reuse data or add more data to the DATA statements.

----

40 MESSAGE 1. Improper REPLACE or DELETE.
2. RENUMBER cannot alter line sequence.

CAUSE 1. Because they affect the structure of the program, some statements may not be deleted or replaced individually. These are the SUB statement and the DEF FN statement. An attempt to replace or delete either of these statements will result in this error.

2. The sequence of lines in a program was altered by a TO clause which has a value outside the original range of values to be renumbered.

ACTION 1. Do not delete or replace the line, or delete the whole sub-procedure or function.

2. Be certain that the renumbering is done so that the sequence of statements in the current program is not altered.

----

42 MESSAGE Attempt to replace or delete busy line or subprogram.

CAUSE A line or subprogram was deleted or replaced in the middle of its execution.

ACTION Do not delete this line or subprogram until the end of its execution.

----

43 MESSAGE Matrix not square.

CAUSE The matrix passed to the built-in functions INV (matrix inversion) and DET (determinant) is not a square matrix.

ACTION Redefine the matrix to make it square.

----

44 MESSAGE Illegal operand in matrix transposition or matrix multiplication.

CAUSE The operands passed to the built-in matrix functions TRN (transpose) and MUL (multiply) are not declared correctly, so the operation cannot be performed.

ACTION Redefine the operands.

----

47 MESSAGE 1. VAR COMMON area does not exist.
2. Dimension or type of COMMON variable in line N doesn't match main.
3. Variable list in line N exceeds the COMMON declaration in main.
4. VAR COMMON area is larger than defined in the original program.
5. COMMON declaration in line N doesn't match the original program.
6. VAR COMMON area has more variables than the original program.

CAUSE All messages in this error number concern with errors with COMMON declarations; N is a line number where the error occurs and VAR is the name of the COMMON area.

1. A subprogram contains a named COMMON that does not exist in the main program.

2. A COMMON in a subprogram contains variables that do not match that declared in the main program.

3. A COMMON in a sub-program contains more variables than is

declared in the main program.

Messages four through six only occur when executing a GET in the interpreter.

4. A program that is brought into the interpreter by the GET command contains a COMMON that is larger than the program that contains the GET command.

5. A program that is brought into the interpreter by the GET command contains variables in COMMON that do not match those in the program that contains the GET command.

6. A program that is brought into the interpreter by the GET command contains more variables in a COMMON area than is declared in the program that contains the GET command.

ACTION Check COMMON or its variables for consistency with subprogram or main.

----

48 MESSAGE Recursion not allowed in single line functions.

CAUSE A single line function is calling itself.

ACTION Redefine the function to eliminate recursion.

----

49 MESSAGE Subunit specified in ON declaration not found.

CAUSE The subunit in an ON...CALL...statement does not exist.

ACTION Provide the missing subunit or call another subunit.

----

50 MESSAGE File number out of range.

CAUSE The file number in an ASSIGN # statement exceeds the range of a positive, non-zero 16 bit integer; (1, 32767).

ACTION Change the file number.

----

51 MESSAGE The file is not currently open.

CAUSE A file was accessed without first being opened.
ACTION Open the file before accessing it.

----

52 MESSAGE Improper group.account specifier.

CAUSE The group or account does not exist when a fully qualified file name is used in a file reference.

ACTION Use a correct group.account specifier for the file.

----

53 MESSAGE Improper file name.

CAUSE The file name used either contains characters that are illegal in file names or the file name is longer than the legal length.

ACTION Change the file name.

----

54 MESSAGE 1. Duplicate file name.
2. File already exists; use RESAVE to overwrite.

CAUSE 1. A file that already exists was created.

2. The current program was saved to a file that already

exists.

ACTION 1. Use a different file name or purge the existing file.

2. Use the RESAVE command or SAVE into a new file.

---

55 MESSAGE Permanent directory overflow.

CAUSE The file directory is full, no more new files can be created.

ACTION Purge some old and unused files.

---

56 MESSAGE File does not exist.

CAUSE A file that does not exist was referenced.

ACTION Use a different file or create the file.

---

58 MESSAGE 1. Operation inconsistent with file type or device type.
2. Invalid file type: Must be ASCII, BASIC DATA, or BASIC SAVE.
3. Invalid file type: Must be ASCII or BASIC DATA file.

CAUSE 1. A file was accessed in a way that is illegal either because of the file type or because of the device the file is on. Examples are :
A direct read/write on a tape file.

A direct word read on an ASCII file.

2. The file code on a file for a GET, RUN, or GET SUB command is not a BASIC data, BASIC SAVE, or ASCII file.

3. A MERGE or LINK command was issued for a file that is neither a BASIC data file nor an ASCII file.

ACTION 1. Change the access method or move the file to a different device.

2. Make sure these commands are used with the valid file code. Only BASIC DATA, BASIC SAVE, or ASCII files are valid file types.

3. Resave the program for the MERGE or LINK using the SAVE or SAVE LIST command.

---

59 MESSAGE End of file found.

CAUSE A file was accessed beyond its logical end. This is usually caused by trying to read more data that the file contains or by writing to a file that is already full.

ACTION Expand the file or don't access records beyond the logical end of the file.

---

60 MESSAGE Physical or logical end of record found in direct access mode.

CAUSE The record size of a file is not big enough to hold the entire output list during a direct record write.

ACTION Reduce the output list or re-create the file with a larger record size.

---

61 MESSAGE BASIC data file record size too small for data item.

CAUSE The record size of the HP Business BASIC data file is too small for the numeric data item being output.

ACTION Re-create the file and increase the record size.

----

62 MESSAGE File is protected, wrong lockword/password specified.

CAUSE The wrong lockword has been used in trying to open a protected file.

ACTION Use the correct lockword.

----

63 MESSAGE Invalid record size specification.

CAUSE The RECSIZE specification in a CREATE statement is invalid; most probably too large.

ACTION The maximum record size allowed is installation defined. Check your installation for RECSIZE limit and modify the CREATE statement.

----

65 MESSAGE Incorrect data type in BASIC data file.

CAUSE A numeric item from a BASIC data file was read into a string variable or vice versa.

ACTION Use a different variable or redefine the current one.

----

68 MESSAGE Syntax error at character N.

CAUSE All syntax errors are error number 68. N points to the character in the statement that produced the error. This message is usually followed by another more specific description. For a list of the syntax error messages, see the last section of this appendix.

ACTION Re-enter the statement.

----

92 MESSAGE Cannot access file because file is being accessed or is accessed exclusively.

CAUSE This error is most often caused when a file that is opened for exclusive access in another statement is accessed or a file that is still open (active) is purged.

ACTION Close the file before accessing it.

----

93 MESSAGE Operation inconsistent with file open mode.

CAUSE A file is accessed in a way that is inconsistent with its file open mode. For example, a file that is open for read only is written to.

ACTION Close the file and re-open it in a different mode.

----

100 MESSAGE IMAGE specification expects a numeric item.

CAUSE A string value has been assigned when the format specification in the IMAGE statement specifies a numeric format.

ACTION Change the format specification.

----

101 MESSAGE IMAGE specification expects a string item.

CAUSE A numeric value has been assigned when the format in the IMAGE statement specifies a character format.

ACTION Change the format specification.

---

102 MESSAGE Format specification too long.

CAUSE The output format of an item in an IMAGE statement is longer than the internal buffer can handle.

For example, IMAGE DD.DD,510X,K .

The specification 510X overflows the internal buffer used for formatted input.

ACTION Reduce the size of the format specification, or break it up into two or more separate formats.

---

103 MESSAGE No IMAGE format specifications exist.

CAUSE An item was output using formatted output, but the IMAGE statement does not contain any format specifier.

ACTION Add the appropriate format specifier to the IMAGE statement.

---

104 MESSAGE File open conflict with previous open mode.

CAUSE A file has been opened a second time after it has been opened already, and the second open mode conflicts with the first one. For instance, a file that is open for APPEND cannot be opened in any other mode.

ACTION Re-open the file in a different mode.

---

110 MESSAGE Program unit is too large. No space available to process this line.

CAUSE The program is too big to be processed by the interpreter.

ACTION See Error 2 - memory overflow.

---

111 MESSAGE Too many REMARKS or DATA in subunit or too many subunits in program.

CAUSE The program is too big to be processed by the interpreter.

ACTION See Error 2 - memory overflow.

---

112 MESSAGE Cannot add subunits because of size of largest subunit.

CAUSE Program is too big to be processed by the interpreter.

ACTION Reduce the subunit size.

---

113 MESSAGE Programs cannot be RUN when the subunit space has been set above 12400 words.

CAUSE The subunit space is set too large for the program to be run in the interpreter. This message is mainly for the Program Analyst.

ACTION Reduce the subunit space size.

---

114 MESSAGE Size requested for subunit space is too large. Default size will be used.

CAUSE The subunit space size requested is too large (larger than 20000), the default size of 10466 words is used.

ACTION None, the default size will be used.

---

115 MESSAGE Too much data space used in this subunit.

CAUSE The most probable cause is that the common area in this subunit contains a variable that is too big. For instance, an array A(10000) in a COM statement.

ACTION Reduce the size of large arrays.

---

117 MESSAGE Not enough memory available for local variables in subunit.

CAUSE The program is too big to be processed by the interpreter.

ACTION Reduce the size of the program.

---

119 MESSAGE Unable to allocate data space, request would cause total to exceed configured limit.

CAUSE The program is too big to be processed by the interpreter.

ACTION Reduce the size of the program.

---

131 MESSAGE Device unavailable.
CAUSE The device to which a file is assigned is not available. For example, a file is assigned to a tape drive that is either already assigned or is not turned on. This message is usually returned by the operating system.

ACTION Assign the file to a different device or resolve the problem with the requested device

---

132 MESSAGE Cannot READ a number from a quoted string.

CAUSE This error message only comes from a compiled program. The READ statement specifies a numeric variable but there is string data in the DATA statement.

ACTION Correct the DATA statement or read the string data into a string variable.

---

134 MESSAGE Unit not ready or online.

CAUSE The device to which a file is assigned is not ready to be used. For example, a tape drive may not be online when a tape file is being read.

ACTION Ready the device and continue.

---

150 MESSAGE Type of CASE expression does not match type of SELECT.

CAUSE The SELECT variable and the CASE expression do not match in data type. For example, the SELECT specifies a numeric variable but the CASE specifies a string variable.

ACTION Change the SELECT and CASE statements to use data of the same type.

---

151 MESSAGE This statement cannot occur in this report section

CAUSE The interpreter has detected a statement in a report section that does not belong there. This error is part of error 157, VERIFY error.

ACTION Delete the statement.

---

152 MESSAGE Structured construct mismatch with lines N and M.

CAUSE A multi-line construct is mismatched in its begin and end. For instance, an multi-line IF is closed with an ENDWHILE statement or vice versa. This error is part of error 157, VERIFY error.

ACTION Add the necessary statement.

---

153 MESSAGE Structured construct error with line N.

CAUSE The interpreter has detected a statement that is not meaningful in that context. For instance, an ELSE statement found in the program that is not part of any multi-line IF statement. This error is part of error 157, VERIFY error.

ACTION Delete line N.

---

154 MESSAGE GRAND TOTALS must go in REPORT HEADER, REPORT TRAILER or REPORT EXIT.

CAUSE The GRAND TOTALS statement is in the wrong part of a report writer section. This error is part of error 157, VERIFY error.

ACTION Move this line to the appropriate section.

---

155 MESSAGE TOTALS must go in HEADER or TRAILER section.

CAUSE The TOTALS line is in the wrong section of the Report Writer code. This error is part of error 157, VERIFY error.

ACTION Move the TOTALS line to the appropriate section.

---

156 MESSAGE This statement must occur within a report definition.

CAUSE The interpreter detected a statement in the report description section that should be in the report definition section. This error is part of error 157, VERIFY error.

ACTION Remove the statement.

---

157 MESSAGE VERIFY error(s) in program.

CAUSE The interpreter, before executing your program, must first verify that the program is correctly structured. Any errors detected during VERIFY are reported separately, followed by this error message.

ACTION Correct the error indicated and re-run the program.

---

158 MESSAGE This statement may not be used in a report definition.

CAUSE A statement that should not appear inside a report definition has been detected. This error is part of error 157, VERIFY error.

ACTION Delete the statement.

---

171 MESSAGE Statement can only occur in a SUB.

CAUSE A multi-line function is found to contain a statement that has meaning only in a SUB. For instance, a SUBEXIT statement is in a multi-line function. This error is part of error 157, VERIFY error.

ACTION Delete or replace the statement.

---

174 MESSAGE Statement on line N can only occur in a numeric function.

CAUSE A string function contains a numeric value in its RETURN statement. This error is part of error 157, VERIFY error.

ACTION Change the return value to the correct type.

---

175 MESSAGE Statement on line N can only occur in a string function.

CAUSE A numeric function contains a string value in its RETURN statement. This error is part of error 157, VERIFY error.

ACTION Change the return value to the correct type.

---

176 MESSAGE Statement on line N can only occur in a multi-line function.

CAUSE The statement on the cited line is allowed only in a multi-line function.

ACTION Make sure that this statement is in a function or use a different statement.

---

177 MESSAGE Dimensions of VAR use local variables.

CAUSE A variably dimensioned array VAR uses a local variable in its definition. For instance, DIM A(Loc) where Loc is a local variable in the SUB where A is defined. This error is part of error 157, VERIFY error.

ACTION Redefine the array.

---

178 MESSAGE Dimensions of VAR use single line functions containing local variables.

CAUSE A variably dimensioned array uses a single line function to return its dimension. The single line function uses a local variable of the SUB in which it is defined. This error is part of error 157, VERIFY error.

ACTION Redefine the variably dimensioned array.

---

179 MESSAGE Structured construct on line N not properly closed.

CAUSE The interpreter has detected a multi-line construct that has no closing statement. For instance, a SELECT has no corresponding ENDSELECT statement, or a multi-line IF has no ENDIF. This error is part of error 157, VERIFY error.

ACTION Add the required closing statement.

----

180 MESSAGE Illegal data in input.

CAUSE An illegal data item has been detected during input. If input data is numeric, this probably means there are non-numeric characters in the data. For string input, it could mean the data is longer than the receiving variable.

ACTION Check input data for accuracy.

----

182 MESSAGE Current CHARS value of N exceeds valid range of 1 to 500.

CAUSE In the ACCEPT or TINPUT statement, the CHARS option specifies N characters, more characters than allowed.

ACTION Input fewer characters.

----

200 MESSAGE Line referenced is not a PACKFMT statement.

CAUSE In a PACK or UNPACK statement, the referenced PACKFMT line is not a PACKFMT statement.

ACTION Check the PACK or UNPACK statement to make sure the line referenced is a PACKFMT line.

----

202 MESSAGE String in PACK/UNPACK statement not long enough for PACKFMT list.

CAUSE The string variable in a PACK or UNPACK statement is not long enough to accommodate all the variables specified in the PACKFMT statement.

ACTION Check the PACKFMT list for accuracy or redefine the string variable in the PACK or UNPACK statement.

----

210 MESSAGE Bad data base status array.

CAUSE The status array for the database statements is not a ten word array of short integers.

ACTION Correct the data type of the status array.

----

211 MESSAGE No DBASE IS statement active.

CAUSE A database has been sorted or searched without first defining it with a DBASE IS statement.

ACTION Add a DBASE IS statement.

----

212 MESSAGE Data set N in thread list not in data base.

CAUSE One of the data sets defined in a THREAD IS statement, number N, does not exist in the database.

ACTION Check the database for valid data sets.

----

213 MESSAGE Illegal items in IN DATASET statement.

CAUSE The IN DATASET statement used by SORT to locate the sort key contains an illegal specification.

ACTION Delete the illegal specification in question.

---

214 MESSAGE Substring not allowed in IN DATASET statement with SORT.

CAUSE The IN DATASET statement used by SORT to locate the sort key contains a substring variable.

ACTION Replace the substring variable with a string.

---

215 MESSAGE Variable dimensioned array not allowed in IN DATASET statement with SORT.

CAUSE The IN DATASET statement used by SORT to locate the sort key contains a variably dimensioned array variable.

ACTION Replace it with a regularly dimensioned array.

---

216 MESSAGE IN DATASET does not allow string parameters to be used.

CAUSE The IN DATASET statement used by SORT to locate the sort key contains a string variable which is passed into the procedure as a parameter from another procedure.

ACTION Use a locally defined string variable of the same length as the parameter.

Numbered Error Messages (219 - 1118)

---

219 MESSAGE Line referenced is not an IN DATASET statement.

CAUSE In the SEARCH, SORT, DBGET, DBPUT, and DBUPDATE statements, a line was referenced that was not an IN DATASET or PACKFMT statement, but should be.

ACTION Add the missing statement.

---

233 MESSAGE Data base not open.

CAUSE The database being accessed has not been opened.

ACTION Open the database.

---

234 MESSAGE Improper dataset linkage in a THREAD statement.

CAUSE Data sets specified in a THREAD IS statement can be linked together either by a path number or a variable name. The program has not linked the data sets in either way.

ACTION Correct the linkage or use the default one.

---

235 MESSAGE No WORKFILE is active.

CAUSE No WORKFILE is specified during the execution of the SORT or SEARCH statement.

ACTION Define the workfile by using a WORKFILE IS statement.

---

236 MESSAGE Unable to find the item in the IN DATASET list.

CAUSE The SORT statement cannot locate the key in any of the IN

DATASET statements referenced.

ACTION Add the key used in an IN DATASET statement.

---

238 MESSAGE Improper PATH or LINK specified in a THREAD statement.

CAUSE The PATH must be a valid path defined in the database schema and the LINK variable, if used, must be the same data type as the key.

ACTION Use the correct PATH number or LINK variable.

---

239 MESSAGE Workfile has wrong file type or open mode.

CAUSE A WORKFILE must be a binary file and opened for read and write. One of these conditions is not satisfied.

ACTION Correct the file type or open mode.

---

240 MESSAGE Line referenced is not a THREAD IS statement.

CAUSE A THREAD IS statement is not being referenced in a SORT/SEARCH operation.

ACTION Add the missing THREAD IS statement.

---

241 MESSAGE Workfile record size not long enough for thread list.

CAUSE The workfile's record size is too short.

ACTION Make sure the record size of a workfile is at least N 32-bit words (or 2N 16-bit words) long; where N is the number of datasets in the thread.

---

242 MESSAGE String variables not allowed in WORKFILE.

CAUSE Only numbers can be written into a workfile.

ACTION If a string is being written to a workfile, something is wrong in the program, possibly between HP Business BASIC/XL and the database. Check the program for errors and delete the string from the output list.

---

243 MESSAGE The workfile is empty for a FILTER or SORT ONLY statement.

CAUSE The workfile is empty when a FILTER or SORT ONLY statement is encountered.

ACTION Create the workfile first before these statements are executed.

---

244 MESSAGE Thread list contains more than 10 data sets.

CAUSE A maximum of 10 data sets is allowed in a thread.

ACTION Reduce the number of data sets in the thread.

---

245 MESSAGE Improper sort key used.

CAUSE Only simple variables can be used as key. Other types of variables, such as array elements or substrings will result in this error.

ACTION Change the key to use only simple variables.

---

246 MESSAGE The SORTINIT intrinsic failed during SORT.

CAUSE The SORT statement encountered some system level problem.

ACTION Use the CCODE function to check the condition code returned by SORTINIT and consult the *SORT-MERGE/XL General Users Guide* for an explanation of the error condition.

---

250 MESSAGE BEGIN REPORT does not reference a REPORT HEADER statement.

CAUSE The statement referenced by BEGIN REPORT is not a REPORT HEADER.

ACTION Make sure BEGIN REPORT reference a REPORT HEADER statement.

---

251 MESSAGE Report Writer is already active.

CAUSE A Report Writer has been executed more than once or more than one report was active at one time.

ACTION Only one active Report is allowed at any one time. Stop the active Report before starting another one.

---

252 MESSAGE Duplicate Report Writer Section statement with line {line number}.

CAUSE 1. A Report Writer Section which may only be defined once within a report has been defined more than once.

2. Two or more HEADER N or TRAILER N statements use the same value for N, resulting in two sections with the same level number.

ACTION 1. Remove or consolidate sections so that only one such section is defined in the report.

2. Change HEADER level values and TRAILER level values so that each level is used only once.

---

253 MESSAGE Duplicate Report Writer Block statement.

CAUSE 1. More than one of the following occurs in a report definition: PRINT DETAIL IF, GRAND TOTALS, PAGE LENGTH, PAUSE AFTER, SUPPRESS PRINT AT, SUPPRESS PRINT FOR, LEFT MARGIN

2. TOTALS ON occurs in both a HEADER and a TRAILER section with the same level number, or more than one TOTALS ON statement occurs within one section.

3. Two or more BREAK WHEN or BREAK IF statements are defined at the same level number.

ACTION 1. Remove the duplicate statements.

2. Consolidate the TOTALS ON statements for a section into one statement. Include this in either the HEADER or TRAILER statement, but not both.

3. All BREAK statements must specify different levels. Change the level numbers to ensure that each BREAK WHEN and BREAK IF statement refers to a unique level.

---

254 MESSAGE 1. Blank lines specified are larger than page size.
2. Blank lines value out of the range 0 to 255.

```
            CAUSE        These are both Report Writer errors:

                         1.  Blank_top or blank_bottom values are greater than the
                         page_length value on the PAGE LENGTH statement.

                         2.  Blank_top or blank_bottom values on the PAGE LENGTH
                         statement are out of range.

            ACTION       1.  (Blank_top + blank_bottom) must be < page_length.

                         2.  Use a value in the range 0 to 255.

---------------------------------------------------------------------------------

255         MESSAGE      1. Unacceptable value for Report Writer expression.
                         2. Subscript of report writer built-in function is out of
                         range.

            CAUSE         1.  A page_number expression is less than zero for one of the
                        following statements:  PAUSE AFTER, PAUSE EVERY, SUPPRESS FOR,
                         SET PAGENUM. Or, a PAGE LENGTH < 0 or > 32767 was specified
                         (maximum XL page length:  2147483647).

                         2.  Out-of-range subscript for one of the Report Writer
                         built-in functions:NUMBREAK, NUMDETAIL, RWINFO.

            ACTION       1.  Use a page_number value >= zero.

                       2.  Consult the reference manual for the legal subscript range.

---------------------------------------------------------------------------------

256         MESSAGE      1. Left margin too close to right margin of output file.
                         2. Left margin too close to right margin of COPY ALL OUTPUT
                         file.

            CAUSE        These are all Report Writer errors:

                        An attempt has been made to make the distance between the left
                         margin and the right margin less than the size of an output
                         field item.

           ACTION       The number of characters between the left margin and the right
                         margin must be set to at least the width of an output field
                         item.  An output field item is initially 20 characters wide,
                         but can be set to 15.
---------------------------------------------------------------------------------

257        MESSAGE     1. Report Writer statement illegal when a report is not active.
                         2. Attempt to evaluate report writer built-in when no report
                         active.
                         3. Report Writer operation outside the scope of an active
                         report.

            CAUSE        An attempt has been made to execute a report writer statement
                         or function which may not be executed when a report was not
                         active.  Or, it may be that a report is active, but the
                         statement or function in question does not appear in the
                         subunit in which the report is active.

            ACTION       These statements and functions may only be executed when a
                         report is active.  The BEGIN REPORT statement activates a
                         report.  If the report is active in another subunit then you
                        can put the report and the statement or function which caused
                        the error in the same subunit, or the desired information can
                         be computed in the report subunit and passed to the other
                         subunit.

---------------------------------------------------------------------------------

258         MESSAGE      Effective page size too small.

           CAUSE       This indicates that the Report Writer blank line specifications
                        at the top and bottom of the page plus the number of lines in
```

the page header and page trailer sections is too large for the size of page you are using.

ACTION Adjust the line specification on the PAGE LENGTH, PAGE HEADER, or PAGE TRAILER statements so that at least three lines are left on the page after subtracting out blank top, blank bottom, header size, and trailer size.

---

259 MESSAGE Illegal execution of a Report Description section statement.

CAUSE All GOSUB statements activated from a Report Writer section have not returned before the end of the section has been seen. The end of the report writer section is marked by the execution of any Report Section statement (HEADER, END REPORT DESCRIPTION).

ACTION The program logic must be changed so that all GOSUBs have been RETURNed from before the next report section statement is executed.

---

260 MESSAGE Insufficient space for printed output within the current page.

CAUSE 1. There are no lines left on the page for Report Writer output before the PAGE TRAILER or bottom blank lines are printed.

2. The PAGE TRAILER prints more lines than are reserved.

ACTION 1. Check the size of the PAGE. Make sure that all Report Section statements use the WITH <number> LINES clause, and that the number of lines includes all output produced by the Report Section. Make sure that DETAIL LINE uses the WITH clause and that it includes all output which may occur before the next DETAIL LINE or Report Section. Output from PRINT, PRINT USING, and COPYFILE is considered report output. DISP, DISP USING, DISP, and all Business BASIC system output (such as CAT, error messages) are not considered report output.

2. Change the number of lines reserved in the WITH clause, or change the number of lines printed by the PAGE TRAILER section.

---

261 MESSAGE Left margin specified is less than 1 or greater than printer width.

CAUSE 1. LEFT MARGIN is set to 0 or less.

2. LEFT MARGIN is too close to the right margin for output device.

3. LEFT MARGIN is too close to the right margin of COPY ALL OUTPUT device.

ACTION 1. Set value to at least 1.

2. Set value to (at most) one tab stop (20 characters) from the right margin.

3. LEFT MARGIN must also be at least one tab stop less than the device size of the COPY OUTPUT device. Change the value of left margin, turn off COPY ALL OUTPUT, or change the size of COPY device.

---

264 MESSAGE Level number is out of the range 1 through 9.

CAUSE 1. The level number is less than zero or greater than nine in one of the following: HEADER, TRAILER, BREAK WHEN, BREAK IF, SUPPRESS PRINT AT.

2. The level number out of range in TRIGGER BREAK.

ACTION 1. If a constant is used, change the report definition to use levels 1 thru 9. If an expression is used, this error is reported during BEGIN REPORT execution. Verify that all level number expressions return 0-9. (Level zero cause statement to be ignored.)

2. Check value used by TRIGGER BREAK. Modify program logic or TRIGGER BREAK statement to use 1-9.

----

265 MESSAGE (GRAND) TOTALS statement not active at the level requested.

CAUSE 1. There is no GRAND TOTALS statement in the report and TOTAL(0,n) is used.
2. TOTAL(L,N) is used, but there is no TOTALS ON statement at level L.

ACTION 1. Add a GRAND TOTALS statement, or change the TOTAL built-in function to use a defined level.

2. Add a TOTALS statement at level L, or change the TOTAL built-in function to use a defined level.

----

266 MESSAGE Sequence parameter out of range for (GRAND) TOTALS at desired level.

CAUSE TOTAL(L, N) is used, but fewer than N expressions are being totalled at level L.

ACTION Count the number of expressions defined at level L and change the TOTAL call to ensure N does not exceed this number or add more expressions to the TOTALS ON or GRAND TOTALS statement.

----

267 MESSAGE WITH number LINES value is negative or greater than page size.

CAUSE 1. The WITH value is less than zero.

2. The WITH value is larger than defined page size.

ACTION 1. Change the WITH clause to use zero or more lines. Check expressions to ensure that negative numbers are not being used.

2. The WITH value may not be larger than first value to the PAGE LENGTH command, unless PAGE LENGTH 0 is used. Change the WITH clause and check expressions used in the WITH clause. Check PAGE LENGTH command for correct page size specification.

----

268 MESSAGE OLDCV($) at requested level does not have a BREAK WHEN active.

CAUSE OLDCV(L) or OLDCV$(L) is being used, but level L does not have a BREAK WHEN statement.

ACTION If level L exists, add a BREAK WHEN statement, or change a BREAK IF statement into a BREAK WHEN statement. Check the value of level to ensure the correct level is being indicated.

----

269 MESSAGE OLDCV($) function does not match control variable data type.

CAUSE 1. OLDCV(L) has been used with a string control variable.

2. OLDCV$(L) has been used with a numeric control variable.

ACTION Make sure that level L is the desired level. Change program logic to ensure that the correct type is being checked.

---

270 MESSAGE Cannot redirect or copy output while a report is active.

CAUSE A SEND OUTPUT or COPY ALL OUTPUT statement was executed after an active report started printing report output.

ACTION Redirect or start copying output before any report output. The following statements will start report output: DETAIL LINE, TRIGGER BREAK, TRIGGER PAGE BREAK, and END REPORT. Alternatively, delay redirection or copy until the report has ended (after END REPORT or STOP REPORT).

---

271 MESSAGE 1. Statement illegal during DETAIL LINE or page break processing.
2. Statement illegal during page break processing.

CAUSE These are all Report Writer errors:

1. A DETAIL LINE or TRIGGER BREAK statement has been encountered while a DETAIL LINE or TRIGGER BREAK statement is executing.

2. A TRIGGER PAGE BREAK has been encountered while a page break is being performed.

ACTION 1. Program logic must be changed such that only one DETAIL LINE or TRIGGER BREAK statement executes at a time.

2. Program logic must be changed so that TRIGGER PAGE BREAK does not execute while the PAGE HEADER or PAGE TRAILER section is active.

---

272 MESSAGE END REPORT may not be executed while any report section is active

CAUSE An END REPORT statement is encountered while a report section (such as HEADER or REPORT TRAILER) is executing.

ACTION Program logic must be change so that END REPORT does not execute during any break or page break processing. Alternatively, use STOP REPORT to stop the report immediately.

---

273 MESSAGE STOP REPORT is already executing.

CAUSE A STOP REPORT statement is encountered during the processing of a STOP REPORT statement.

ACTION Program logic must be changed to ensure that only one STOP REPORT statement is executed at once.

---

284 MESSAGE Buffer for block read of JOINFORM is too small.

CAUSE The buffer in the call to bb_block_read is not large enough.
ACTION The buffer must be large enough to hold all characters from all fields on the form plus one byte per field.

---

285 MESSAGE Form file inconsistent, possibly corrupted.

CAUSE When JOINEDIT needs to make several writes to a JOINEDIT form file to complete an operation it marks the file "inconsistent" before the first write and "consistent" after the last one has been completed successfully. This is done because JOINEDIT might abort after the first write but before the last one. This protects against an internally inconsistent form file being considered consistent.

ACTION There may be little that can be done in this case. Call HP for assistance in piecing together your form file. It is helpful to know what changes were made since the last time the form file was consistent.

---

286 MESSAGE Error when writing to form. Form possibly not displayed correctly.

CAUSE After displaying a form, BASIC does a cursor position check to see if the cursor is where it should be after display of the form. If it is not in the expected position then this error occurs.

ACTION There are several reasons for this problem, such as a corrupted form or a terminal that does not have enough memory to display the form.

---

287 MESSAGE No input fields in form.

CAUSE The program has tried to read data from a form, but there are no input fields on that form.

ACTION The form must have an input field for input to occur.

---

288 MESSAGE No output fields in form.

CAUSE The program has tried to write data to a form, but there are no output fields on that form.

ACTION For output to occur while a form is active, there must be an output field in the form. The exception to this is the LDISP statement. See the JOINFORM appendix for an explanation of how LDISP works when a JOINFORM form is active.

---

289 MESSAGE Output too long for output field {field_number}.
CAUSE This indicates that an output statement has tried to display a value which is too long to fit in the current output field of the active JOINFORM form.

ACTION The field must be made larger or the data smaller. The largest field size is 80 characters.

---

291 MESSAGE Illegal operation inside form.

CAUSE The LENTER statement cannot be executed when the cursor is located within the currently-active JOINFORM form.

ACTION Before executing the LENTER statement, use the CURSOR statement to position the cursor to a location outside of the form.

---

292 MESSAGE Attempt to read past last input field of form.

CAUSE The input field pointer is undefined because all the fields on the form have already been read.

ACTION Use the CURSOR IFLD(input_field_number) statement to position the input field pointer to the desired field.

---

293 MESSAGE Attempt to write past last output field of form.

CAUSE The output field pointer is undefined because all the fields on the form have already been written.

ACTION Use the CURSOR OFLD(output_field_number) statement to position the output field pointer to the desired field.

---

294 MESSAGE Operation only allowed when a joinform is active.

CAUSE The IFLD, OFLD and CFLD clauses of the CURSOR statement are not legal when a JOINFORM form is not active. The bb_block_read routine cannot be called when a JOINFORM form is not active.

ACTION Use the OPEN FORM statement to activate a form.

---

295 MESSAGE Field number of CURSOR statement does not exist.

CAUSE There is no field on the form with the cursor field number indicated by the IFLD, CFLD, or OFLD item on the CURSOR statement.

ACTION Check your form definition (possibly by using JOINEDIT) to determine the field number of the field you want the cursor to be on.

---

296 MESSAGE Form not found in formsfile.

CAUSE An attempt to open a JOINFORM form failed because no form with that name exists in the specified form file.

ACTION Make sure the formfile and formname are specified in the correct order, "formname:formfile". Use JOINEDIT to display or print the directory of forms in the formfile.

---

297 MESSAGE Found invalid data in formsfile.

CAUSE The name of the JOINFORM in the directory of the currently open JOINFORM file does not match the name in the header of the actual JOINFORM. The JOINFORM file has probably been corrupted.

ACTION Re-create the JOINFORM file using the JOINFORM editor to salvage as much of the uncorrupted form information as possible.

---

298 MESSAGE Input field "input_field" too long for variable "item_number".

CAUSE The value of a JOINFORM field, input_field, was assigned to a string variable that is too short. The string variable is the item_number variable specified in a variable list following either an INPUT or ENTER statement.

ACTION Declare the length of the string variable to be greater than its currently declared length.

ACTION Use a substring specifier following the variable in the variable list. For example, 10 INPUT A$[1].

---

299 MESSAGE Numeric data expected in input field "input_field" for variable "item_number".

CAUSE The value of the input field, numbered input_field, of the currently displayed JOINFORM is not numeric. The item_number variable in the variable list following an INPUT or ENTER statement is numeric, so the non-numeric value cannot be assigned.

ACTION Reenter a numeric value in the appropriate field in the JOINFORM.

---

320 MESSAGE Invalid item name in PREDICATE statement.

CAUSE The value of the item name at the time of execution of the PREDICATE statement is the null string.

ACTION Ensure that the value of the string expression that represents the name of the database item to lock in the specified data set is correct.

---

321 MESSAGE Invalid relational operator in PREDICATE statement.

CAUSE Run-Time: The only valid relational operators in a WITH clause of a PREDICATE statement are one of:

=

>=

<=

Use of any other relational operators will cause an error.

ACTION Use only one of the above relation operators.

---

322 MESSAGE Predicate string too short for the predicate elements.

CAUSE The data set and item information that is being packed into the predicate string is greater in length than the maximum length declared for the string variable.

ACTION Increase the length of the string variable that is to be the predicate string for the DBLOCK statement by declaring the string with a length greater than 18 characters or increasing the length of the already declared string.

---

323 MESSAGE Improper data set or base name used.

CAUSE String data set parameter specifying the data set name for a BASIC statement that interfaces with TurboIMAGE exceeds the maximum length of 16 characters.

ACTION Check the string to be certain that is less than 16 characters in length before using the name as a data set name in a BASIC TurboIMAGE statement.

---

324 MESSAGE Buffer not long enough for information returned by DBMS.

CAUSE If this error occurs with a DBGET the input buffer into which the information in the database is to be transferred was too short for the information actually written to the buffer. As a result, the values of other program variables cannot be guaranteed.

If this error occurs with a DBINFO the return string to which the information is to be returned is too short for the information actually written to the buffer. As a result, the values of other program variables cannot be guaranteed.

ACTION Rewrite the program increasing the size of the input buffer or return string. Recovery using ON DBERROR during program execution is not advised because of possible program data corruption.

CAUSE The return message string for the DBMESSAGE statement is not the minimum length required for the call to the TurboIMAGE library routine.

ACTION Rewrite the program, increasing the size of the string variable used with the RETURN clause of the DBERROR statement.

---

800 MESSAGE Data Base Management System error "error_number".

CAUSE A TurboIMAGE database error with the error number, "error_number", has occurred.

ACTION Look up "error_number" in the *TurboIMAGE/XL Database Management System* and take the appropriate action. Additional information can be made available in the program by use of the ON DBERROR statement in conjunction with the DBINFO statement.

---

900 MESSAGE Error 2: Memory overflow.
Error 2: Not enough data space available for the local variables.
Error 51: The file is not currently open.
Error 52: Improper group.account specifier.
Error 53: Improper file name.
Error 54: Duplicate file name.
Error 55: Permanent directory overflow.
Error 56: File does not exist.
Error 58: Operation inconsistent with file type or device type.
Error 59: End of file found.
Error 62: File is protected, wrong lockword/password specified.
Error 92: Cannot access file because file is being accessed or is accessed exclusively.
Error 93: Operation inconsistent with file open mode.
Error 131: Device unavailable.
Error 134: Unit not ready or online.

CAUSE See the description corresponding to the respective error number.

ACTION See the description corresponding to the respective error number.

---

905 MESSAGE Improper fileset specified.

CAUSE An illegal file_set argument has been passed to the CAT command.

ACTION Check the *Accessing Files Programmer's Guide* for a description of file set specifications. Or, enter the MPE XL help system and ask for help on the PARMS of the LISTF command.

---

906 MESSAGE Improper filetype specified in CATALOG.
CAUSE The argument to the TYPE parameter of the CAT command is not legal. It is longer than five characters.

ACTION In CAT File_set$;TYPE=Type$ the Type$ string must not be longer than five characters.

---

910 MESSAGE Improper operating system filename.

CAUSE The filename specified is not a legal MPE filename.

ACTION Consult the *Accessing Files Programmer's Guide* for information on forming filenames.

---

911 MESSAGE Invalid lockword specified.

CAUSE An invalid file lockword was supplied.

ACTION A lockword must have no more than eight characters, beginning with an alphabetic character and followed alphanumeric characters. Check that the specified lockword has these characteristics.

---

912 MESSAGE CATALOG's work-file size has been exceeded.

CAUSE The CATALOG command directs the output from the MPE :LISTF command to a temporary file. This message appears when the file is not large enough for the number of files involved.

ACTION Create a larger LISTF temporary file with:

```
:FILE LISTF;REC=-68,64,F,ASCII;DISC=nnnnn,32;NOCTL;TEMP
```

where nnnnn is a number large enough to hold the :LISTF output. The default value of nnnnn is 10000. To create a file large enough to do a CATALOG on a given number of files,nr_files, use the following:

```
nnnnn >=nr_files+(CEIL(nr_files/53)*5)+2
```

---

913 MESSAGE Short real overflow during conversion from Compatibility Mode short real data.

CAUSE The range of Native Mode (IEEE) short real data is smaller than the range of Compatibility Mode (MPE/V) short real data.

ACTION If the data is in a BASIC DATA file then the conversion utility can be used to convert between REAL types. If the data is in a database then either use Compatibility Mode BASIC to read the data or manual conversion of the database will be required. If the data is internal to the program then changing the target variable to REAL may solve the problem.

---

914 MESSAGE Short real underflow during conversion from Compatibility Mode short real data.

CAUSE The range of Native Mode (IEEE) short real data is smaller than the range of Compatibility Mode (MPE/V) short real data.

ACTION If the data is in a BASIC data file then the conversion utility can be used to convert between REAL types. If the data is in a database then either use Compatibility Mode BASIC to read the data or manual conversion of the database will be required. If the data is internal to the program then changing the target variable to REAL may solve the problem.

---

915 MESSAGE Real overflow during conversion to Compatibility Mode Real data.

CAUSE The range of Native Mode (IEEE) real data is larger than the range of Compatibility Mode (MPE/V) REAL data.

ACTION This value can't be represented as a CM real value. Your program logic might allow the largest and smallest CM real values to be used to represent NM values that are out of CM real range.

---

916 MESSAGE Real underflow during conversion to Compatibility Mode Real

data.

CAUSE The range of Native Mode (IEEE) real data is larger than the range of Compatibility Mode (MPE/V) REAL data.

ACTION This value can't be represented as a CM real value. Your program logic might allow the CM values closest to zero to be used to represent NM values that are out of CM real range.

---

917 MESSAGE NM-specific real value, such as NaN and Infinity, occurs while converting to CM real value.

CAUSE NM real values (such as NaN - "Not a Number", and infinity) have no corresponding value in CM real representation.

ACTION If NaN occurs then check the logic of the program that produced it. If infinity occurs, then see if the logic of your program permits the maximum or minimum representable real to be used in place of infinity.

1101 MESSAGE New line number not between 1 and 999999.

CAUSE A line number has been typed (or created by AUTO) that is not within the legal range for line numbers.

ACTION Line numbers must be in the range: 1 to 999999.

1102 MESSAGE SYSTEM MESSAGE {number}.

CAUSE {number} identifies an operating system error that occurred during execution of a SYSTEM command.

ACTION See the appropriate operating system reference manual for more information.

1103 MESSAGE Number is incomplete.

CAUSE This occurs during execution of the :RUN command. The SYSTEMRUN command has a parameter of the form:

keyword=% or keyword=-

An octal number should follow the "%" and "-".

ACTION See the appropriate operating system reference manual for more information.

1104 MESSAGE Entrypoint name is missing.

CAUSE This occurs during execution of the :RUN command.

ACTION See the appropriate operating system reference manual for more information.

1105 MESSAGE Entrypoint name has more than 15 characters in it.

CAUSE This occurs during execution of the :RUN command.

ACTION See the appropriate operating system reference manual for more information.

1106 MESSAGE First character in entrypoint name is not a letter.

CAUSE This occurs during execution of the :RUN command.

```
                 ACTION      See the appropriate operating system reference manual for more
                               information.

----------------------------------------------------------------------------------------

1107        MESSAGE      Program name's File name is missing.

                 CAUSE         This occurs during execution of the :RUN command.  No program
                               file name was given to the SYSTEMRUN command.

                 ACTION      See the appropriate operating system reference manual for more
                               information.
----------------------------------------------------------------------------------------

1108        MESSAGE      Program name's File name longer than 8 characters.

                 CAUSE         This occurs during execution of the :RUN command.

                 ACTION      See the appropriate operating system reference manual for more
                               information.

----------------------------------------------------------------------------------------

1109        MESSAGE      First character in Program name's File name is not a letter.

                 CAUSE         This occurs during execution of the :RUN command.

                 ACTION      See the appropriate operating system reference manual for more
                               information.

----------------------------------------------------------------------------------------

1110        MESSAGE      Missing equals sign after the keyword {keyword}.

                 CAUSE         This occurs during execution of the :RUN command.

                 ACTION      See the appropriate operating system reference manual for more
                               information.

----------------------------------------------------------------------------------------

1111        MESSAGE      Missing quote after "{INFO | XL} =".

                 CAUSE         This occurs during execution of the :RUN command or SYSTEMRUN
                               command.

                 ACTION      See the appropriate operating system reference manual for more
                               information.

----------------------------------------------------------------------------------------

1112        MESSAGE      Missing ending quote of {INFO | XL | UNSAT} string.

                 CAUSE         This occurs during execution of the :RUN command or SYSTEMRUN
                               command.

                 ACTION      See the appropriate operating system reference manual for more
                               information.

----------------------------------------------------------------------------------------

1113        MESSAGE      {keyword} requires a number.

                 CAUSE         This occurs during execution of the :RUN command or SYSTEMRUN
                               command.

                 ACTION      See the appropriate operating system reference manual for more
                               information.

----------------------------------------------------------------------------------------

1114        MESSAGE      {keyword} must be less than or equal to {number}.

                 CAUSE         This occurs during execution of the :RUN command or SYSTEMRUN
                               command.

                 ACTION      See the appropriate operating system reference manual for more
```

information.

1115 MESSAGE {keyword} must be greater than or equal to {number}.

CAUSE This occurs during execution of the :RUN command or SYSTEMRUN command. The value given for parameter of the RUN command is less than the minimum value allowed for that parameter.

ACTION See the appropriate operating system reference manual for more information.

1116 MESSAGE Missing semicolon.

CAUSE This occurs during execution of the :RUN command or SYSTEMRUN command. The :RUN arguments must be separated by ";".

ACTION See the appropriate operating system reference manual for more information. For example,

```
10 SYSTEMRUN "report.utils;lib=g;maxdata=10000"
```

1117 MESSAGE No Help available on that {topic | subtopic}.

CAUSE The HELP system does not contain information about the topic requested.

ACTION Make sure that the request is correct. If it is correct, submit a Service Request for inclusion of help for that topic.

1118 MESSAGE {INFO, XL or UNSAT} string exceeds the maximum allowance.

CAUSE 1. The MPE INFO string is too long.

2. The XL="list" string is too long.

3. The UNSAT="list" string is too long.

ACTION 1. The MPE INFO string must be 200 characters or less.

2. The XL="list" string cannot be longer than 80 characters.

3. The UNSAT="list" string cannot be longer than 31 characters.

Numbered Error Messages (1119 - 1240)

1119 MESSAGE One of "S", "P", or "G" must come after "LIB=".

CAUSE This occurs during execution of the :RUN command.

ACTION See the appropriate operating system reference manual for more information.

1120 MESSAGE Missing keyword "NEW" after STDLIST specification.

CAUSE This occurs during execution of the :RUN command.

ACTION See the appropriate operating system reference manual for more information.

1121 MESSAGE Unrecognized keyword.

```
           CAUSE        This occurs during execution of the :RUN command.

          ACTION      See the appropriate operating system reference manual for more
                        information.

------------------------------------------------------------------------------------

1122       MESSAGE      Out of system resources for program "{program_name}".

           CAUSE        This occurs during execution of the :RUN command.

          ACTION      See the appropriate operating system reference manual for more
                        information.

------------------------------------------------------------------------------------

1123       MESSAGE      Program "{program_name}" does not exist.

           CAUSE        This occurs during execution of the :RUN command.

          ACTION      See the appropriate operating system reference manual for more
                        information.

------------------------------------------------------------------------------------

1124       MESSAGE      Invalid program "{program_name}".

           CAUSE        This occurs during execution of the :RUN command.

          ACTION      See the appropriate operating system reference manual for more
                        information.

------------------------------------------------------------------------------------

1125       MESSAGE      Entrypoint name does not exist or is invalid for program.

           CAUSE        This occurs during execution of the :RUN command.
          ACTION      See the appropriate operating system reference manual for more
                        information.

------------------------------------------------------------------------------------

1126       MESSAGE      Increased MAXDATA is larger than configuration MAXDATA.

           CAUSE        This occurs during execution of the :RUN command.

          ACTION      See the appropriate operating system reference manual for more
                        information.

------------------------------------------------------------------------------------

1127       MESSAGE      Hard load error for program "{program_name}".

           CAUSE        This occurs during execution of the :RUN command.

          ACTION      See the appropriate operating system reference manual for more
                        information.

------------------------------------------------------------------------------------

1128       MESSAGE      Specified $STDIN could not be opened for program
                        "{program_name}".

           CAUSE        This occurs during execution of the :RUN command.

          ACTION      See the appropriate operating system reference manual for more
                        information.

------------------------------------------------------------------------------------

1129       MESSAGE      Specified $STDLIST could not be opened for program
                        "{program_name}".

           CAUSE        This occurs during execution of the :RUN command.

          ACTION      See the appropriate operating system reference manual for more
                        information.
```

1130 MESSAGE Could not activate new process for program "{program_name}".

CAUSE This occurs during execution of the :RUN command.

ACTION See the appropriate operating system reference manual for more information.

1131 MESSAGE One of "BS", "CS", "DS", or "ES" must come after "PRI=".

CAUSE This occurs during execution of the MPE :RUN command or the HP Business BASIC/XL SYSTEMRUN command.

ACTION See the appropriate operating system reference manual for more information.

1132 MESSAGE In a job, the WAIT statement must specify a time limit.

CAUSE In your job, a WAIT statement does not specify a time limit.

ACTION Specify a value for the WAIT of less than 1.157920892373161E+74 seconds.

1133 MESSAGE The string specified is too long.

CAUSE

1. The string given in the COPTION TITLE="string" or COPTION TITLESUB="string" command is too long.
2. The string given in the COPTION COPYRIGHT="string" command is too long.
3. The string, Image$, given in a "PRINT USING Image$;Value" statement is too long.
4. The set name is too long in: COPTION LOCALITY="set_name".
5. The total length of the string supplied to the SYSTEMRUN command is too long.

ACTION

1. The TITLE string is limited to a length of 132.
2. The COPYRIGHT string is limited to a length of 268435455.
3. The maximum length of an IMAGE string is 500 characters.
4. The name is limited to 16 characters.
5. The string must be 500 characters or less. Make certain that the string specified is within these limits.

1134 MESSAGE Only the words Yes, No, or Exit are allowed as input.

CAUSE You typed something other than Yes, Y, No, N, Exit, E or // in response to the question "Do you want to see more on this topic (Yes, No, Exit)?" while in HELP mode.

ACTION Answer as requested.

1135 MESSAGE SYSTEM with no parameters is not allowed from a batch job.

CAUSE The SYSTEM statement or the ":" command has been executed in batch mode.

ACTION If you have a set of MPE commands to execute, recode the commands to:

SYSTEM "command1"

```
SYSTEM "command2"
...
```

It may be that you have simply forgotten the "exit" command and are running into the job's prompt character:

```
:job jobname,user.acct/passwd

:

:hpbb

10 print "Hi, Mom!"

run

:

:eoj
```

The ":" after the "run" command is read by HP Business BASIC/XL as the HP Business BASIC/XL "SYSTEM" command. In this case, insert an "exit" command between the"run" command and the following ":"...

```
:job jobname,user.acct/passwd

:

:hpbb

10 print "Hi, Mom!"

run

exit

:

:eoj
```

1136 MESSAGE This BASIC program has not been PREPed with Process Handling (PH) capability.

CAUSE The program tried to run another program (possibly with the SYSTEMRUN command) without previously having been given the Process Handling capability.

ACTION Re-PREP the program with CAP=PH.

1137 MESSAGE End of data on input device.

CAUSE 1. The interpreter has encountered the end of the command input file. (The file BASCOM has been redirected.)
2. The program has read beyond the end of the file to which BASIN has been redirected. The program is expecting more data in the file than it should or the file doesn't contain as much data as it should.

3. ":EOD" was typed in response to a request for input.

ACTION 1. End the command file with the "exit" command to terminate the interpreter without an error.

2. Correct the data file or the program logic.

3. This is usually intentional.

1138 MESSAGE Error {error_number} in reading input.

CAUSE A file system error has occurred while reading input.

ACTION Look up the error_number in the description of the fcheck intrinsic in the *MPE XL Intrinsics Reference Manual* for a text description of the problem.

1139 MESSAGE INTEGER precision overflow.

CAUSE 1. An arithmetic operation involving INTEGER operands has produced a result which is out of the range of an INTEGER (possibly after implicit type conversion to make both arguments have the same type).

2. A number that is out of the INTEGER range has been converted to an INTEGER.

3. A value that is out of the range of INTEGER has been read into an INTEGER variable.

4. The largest possible negative integer (-2,147,483,648) has been made into a negative number. This is the one negative integer value which cannot be represented as a positive integer value.

ACTION 1, 2, 3. A (SHORT) REAL or (SHORT) DECIMAL may provide a sufficiently larger range.

4. Assign the number to a type with a larger range (REAL or DECIMAL), then make that negative.

1140 MESSAGE REAL precision overflow.

CAUSE 1. An arithmetic operation involving REAL operands has produced a result which is out of the range of a REAL (possibly after implicit type conversion to make both arguments have the same type).

2. A number that is out of the range of REAL has been converted to a REAL.
3. A value that is out of the range of REAL has been read into a REAL.

ACTION A DECIMAL may provide a sufficiently larger range. (Be aware that the precision of DECIMAL differs.)

1141 MESSAGE SHORT REAL precision overflow.

CAUSE 1. An arithmetic operation involving SHORT REAL operands has produced a result which is out of the range of a SHORT REAL (possibly after implicit type conversion to make both arguments have the same type).

2. A number that is out of the range of SHORT REAL has been converted to a SHORT REAL.

3. A value that is out of the range of SHORT REAL has been read into a SHORT REAL.

4. The value to the EXP() function is out of the range [-87.3366 .. 88.7228].

ACTION 1, 2, 3. Use a REAL (or DECIMAL) for the type of the target variable instead of SHORT REAL. (Be aware that the precision of these types is different.)

4. The argument to EXP must be within the indicated range.

1142 MESSAGE Can't write to file {file_name}.

CAUSE 1. The interpreter was unable to write to the program file while trying to do a SAVE or RESAVE.

2. The interpreter was unable to mark the indicated file as "run only" when executing the RUN ONLY statement.

ACTION Further investigation of this problem is required. Please contact your Hewlett-Packard representative.

1143 MESSAGE Can't read from file.

CAUSE 1. The compiler was unable to read the file containing the program to be compiled.

2. The interpreter was unable to complete a GET SUB statement because of an error when reading from the file containing the subunit to GET.

3. The interpreter was unable to complete a GET statement because of an error when reading from the file containing the program to GET.

4. The interpreter was unable to mark the indicated file as "run only" when executing the RUN ONLY statement.

ACTION Further investigation of this problem is required. Please contact your Hewlett-Packard representative.

1144 MESSAGE Arithmetic overflow on exponentiation.

CAUSE An arithmetic exception was encountered during the processing of an arithmetic expression containing an exponentiation operator. Values of variables or literals cause the numeric value of the intermediate result to exceed the allowable range of the intermediate type.

ACTION Rearrange the expression so that no overflow will occur during the evaluation.

ACTION Trap values that you know will cause this condition.

ACTION Convert values so that an intermediate type with a larger range of values is used.

1145 MESSAGE File version does not match current HPBB version.

CAUSE The file version on the BASIC SAVE file indicates that the file was saved by an interpreter that is a later version than the interpreter or compiler being used.

ACTION Save an ASCII version with the SAVE LIST command from the later version and create a version with the older interpreter. The same file can now be used with the older compiler.

1147 MESSAGE Invalid file type: Must be BASIC SAVE file.

CAUSE The source file specified as the input file for the compiler does not have a BSVXL file code.

ACTION If you saved the file as an ASCII file, GET the file into the interpreter and resave it using the command:

SAVE "filename"

"filename" must not have previously existed.

1150 MESSAGE Bad decimal numeric data found.

CAUSE Some recent input operation, a read from file or database has resulted in a decimal value being read into a variable with a decimal type that does not have the correct format. In other words, the value in the file has been corrupted.

ACTION Correct the value in the file or database.

1151 MESSAGE Cannot RUN a program with OPTION SUBPROGRAM in effect.
CAUSE A program in the interpreter has been run with GLOBAL OPTION SUBPROGRAM in the main of the program.

ACTION Remove the GLOBAL OPTION SUBPROGRAM to run the program in the interpreter. You will probably have to add calls to the procedures and functions in the current program in the interpreter, if you have not already done so.

ACTION If there are only procedures or functions in the current program in the interpreter (this means that there are no executable statements in the main of the current program), you will have to compile the program and put it into an executable library. You can use a program with the appropriate calls to your procedures or functions from the interpreter to test your program.

1152 MESSAGE Irrecoverable error encountered.

CAUSE The interpreter cannot recover to a consistent internal state that guarantees that all user information is not corrupted. You will usually see this message as the interpreter is aborting and just prior to a stack trace indicating the location in the code that the abort occurred and the preceding internal procedure calls made by the interpreter.

ACTION Please copy the information that is on your screen and try to describe the steps that immediately preceded the problem. Submit this report to your System Administrator to forward to the HP Service Engineer.

1153 MESSAGE WARNING 1153: Procedure name "procedure_name" is too long.

CAUSE The procedure name, function name or the alias specified in an INTRINSIC or EXTERNAL statement or the name of the intrinsic in the intrinsic file exceeds the maximum length of 60 characters. This is an error rather than a warning, since processing of the statement is interrupted and the program cannot be run. The name returned from the intrinsic file may be displayed as "procedure name". Note that because of the definition of the intrinsic mechanism, the name to be called may not be the same as either the name in the INTRINSIC statement or the specified alias in the ALIAS clause.

ACTION Shorten the procedure name, function name or the alias name in the INTRINSIC or EXTERNAL statement or be certain that the name of the intrinsic in the intrinsic file is within the appropriate bounds.

1154 MESSAGE Not enough memory available.

CAUSE Current operation has exhausted the amount of space allocated to hold information in the interpreter. Since HP Business BASIC does not do garbage collection of its previously used space, you may have to do this.
ACTION If you obtained this message while entering a program in the interpreter, try to save the current version as an ASCII program file using the command:

SAVE LIST "filename".

If this is successful, then do a GET "filename"

If you obtained this message trying to run a program, then the run-time data structures used to store the values of variables could not be allocated because of space constraints. Try to reduce the size of large arrays or eliminate unnecessary or unused variables.

1155 MESSAGE Unable to open intrinsic file "intr_filename".

CAUSE intr_filename does not exist as specified or it is being accessed exclusively.

ACTION Check to be certain that the file exists or that it is not being accessed exclusively.

1157 MESSAGE Procedure "procedure" is not in the intrinsic file.

CAUSE Interpreter: The name or alias specified in the INTRINSIC statement was not found in the intrinsic file prior to running that main, program or function.

Compiler: The name or alias specified in the INTRINSIC statement in the intrinsic file was not found prior to compiling that main, program or function.

ACTION Check the name of the intrinsic file to be certain that you are using the one that contains the correct name or alias. If this does not work, change the name or alias to conform to the name in the intrinsic file.

1158 MESSAGE The procedure "procedure_name" is being used where a function is needed.

CAUSE A procedure is being called when a return value is expected, either as a function call or with the use of the FNCALL keyword.

ACTION Either change the procedure to a function or call the procedure as a procedure.

1159 MESSAGE Call to procedure "procedure_name" failed.

CAUSE The call to the HP Business BASIC/XL built-in function TASKID failed. The "procedure_name" is PROCINFO, the MPE intrinsic called to obtain the information.

ACTION The most likely cause of this is an operating system problem. The problem is not an HP Business BASIC/XL problem.

1167 MESSAGE Procedure "procedure_name" cannot be called as a function.

CAUSE The intrinsic file indicates that procedure_name is a procedure, but it has been defined in the INTRINSIC statement as having a return value.

ACTION Change the INTRINSIC definition for the procedure by removing the type declaration for the return value.

1169 MESSAGE Command "_" ANYPARM call allowed only in PAUSEd executing subunit with _"procedure_name" call.

CAUSE An ANYPARM external before execution of the main, procedure, or function that contains the ANYPARM call was called before the interpreter has scanned that routine to initialize the internal structures required for the call.

ACTION Add a PAUSE statement at the beginning of the main, procedure, or function that contains the call and execute the program. A call from the interpreter can be made when the program executes the PAUSE statement and returns control to the interpreter.

1177 MESSAGE Value for BYTE parameter # "parameter_number" exceeds range for BYTE type.

CAUSE A BYTE parameter to an external exceeded the range of [-256,255].

ACTION Check the value of the actual parameter to be certain that it is within the specified range. Note that BYTE types are not sign extended.

1180 MESSAGE Parameters out of range for the function.

CAUSE The parameters to the DAT3000$ built-in function must satisfy the following three conditions:

1. The first parameter must be greater than or equal to one.
2. The second parameter must be less than or equal to 27.
3. The first parameter must be less than or equal to the second parameter.

ACTION Be certain that the parameters to the DAT3000$ satisfy the above conditions before calling the built-in function.

1181 MESSAGE Error in accessing the HELP message file.
CAUSE The interpreter was unable to open the catalog HPBBHELP.PUB.SYS following the HELP command.

ACTION Be certain that the file HPBBHELP.PUB.SYS is present on your system and that no one is accessing the file exclusively.

1182 MESSAGE Call to "procedure_name" has "number_formal" parameter(s); declaration has "number_actual" parameters.

CAUSE The number of formal parameters defined in the EXTERNAL statement or in the intrinsic file does not match the number of actual parameters supplied for the call from the program.

ACTION Correct either the definition in the EXTERNAL statement or the call so that the number of parameters is the same.

1183 MESSAGE Too few parameters, or missing parameter in call to "procedure_name".

CAUSE A call to a procedure or function does not have the correct number of actual parameters that correspond to the number of formal parameters declared in the procedure or function header.

ACTION Correct the call to the procedure or function so that the number of actual parameters corresponds to the number of formal parameters defined.

1186 MESSAGE Interpreter BUILD file failure: please PURGE temp files prefixed BBTEMP.

CAUSE Certain HP Business BASIC/XL statements and commands create a file in the temporary file space to store information. HP Business BASIC/XL uses the "BBTEMP" prefix as the first six characters in the temporary filename. If you have created a large number of files with this prefix, HP Business BASIC/XL will be unable to process the information required.

ACTION Check the temporary file space using the command, ":listftemp" to confirm that there are too many files with the prefix "BBTEMP". Modify your program so that you use filenames with a different prefix and purge those unused files prefixed with "BBTEMP" that are in the temporary file space.

1187 MESSAGE Interpreter BUILD file failure: please check whether disk space full.

CAUSE Certain HP Business BASIC/XL statements and commands create a file in the temporary file space to store information. In this case, HP Business BASIC/XL was unable to build the required file.

ACTION Check with your System Administrator concerning the system's available disk space. You may have also reached your account or group disk space limits.

1189 MESSAGE FNCALL of a subprogram is not allowed.

CAUSE The name specified as a parameter to FNCALL is the name of a procedure in the current program.

ACTION Call the procedure using the CALL statement.

1190 MESSAGE ROUND argument cannot be power rounded in the range of decimals.

CAUSE The argument to be rounded is converted to a decimal value and a value to be added to that value for the round is determined. Overflow on the addition causes this error to be generated.

ACTION Check values before rounding so that overflow does not occur during the rounding operation.

1191 MESSAGE DROUND argument cannot be digit rounded in the range of decimals.

CAUSE The argument to be rounded is converted to a decimal value and a value to be added to that value for the round is determined. Overflow on the addition causes this error to be generated.

ACTION Check values before rounding so that overflow does not occur during the rounding operation.

1192 MESSAGE Matrix is singular, cannot be inverted.

CAUSE A matrix cannot be inverted because the current values of the individual elements are such that the matrix is singular.

ACTION Check for singularity prior to trying to invert the matrix.

1194 MESSAGE Bad data in form field for item "form_item_number".

CAUSE One of the following errors occurred when trying to assign the value of a field to a scalar or an element of the array specified in "form_item_number". This happened when a VPLUS form was read with the READ FORM statement.

1. The length of a numeric value in a form field exceeded the maximum length of a numeric literal.
2. There were illegal characters in the numeric literal in the form field.
3. (Scalars only) The conversion of the ASCII numeric literal in the form field to a numeric value of the appropriate type failed.

ACTION Use internal error handling using ON ERROR statements to trap these errors and specify recovery procedures.

1195 MESSAGE String not big enough for form field for item "form_item", subitem "array_element_number".

CAUSE A VPLUS form was read with the READ FORM statement and the conversion of the ASCII numeric literal in the form field to a numeric value of the appropriate type failed. The resulting value would have been assigned to the "array_element_number" of the array which is number "form_item" in list of READ FORM form items.

ACTION Use internal error handling using ON ERROR statements to trap this error and specify recovery procedures.

1196 MESSAGE Invalid file parameters specified in user's :FILE LISTF command.

CAUSE An error was detected in the LISTF file equation required to build the workfile for the CAT command.

ACTION Check the LISTF file equation by using the command,

":LISTEQ"

from the interpreter. The LISTF file equation is required to be in the following format:

```
FILE LISTF;REC=-68,64,F,ASCII;DISC=nnnnn,32;NOCCTL;TEMP
```

where nnnnn is some positive integer value reflecting the maximum estimated number of files that will be processed when using the CAT command. The FILE command can be set from within the interpreter using the SYSTEM command.

1197 MESSAGE No Form file specified now or in the past.

CAUSE A form was opened without first opening a form file.

ACTION Add the name of the form file containing the desired form to the OPEN FORM statement.

1198 MESSAGE No Form open.
CAUSE A CLEAR FORM statement for a VPLUS form was executed when no form was open.

ACTION Add code to open the form prior to trying to clear it.

1199 MESSAGE Account or user MAXPRI= does not permit this value for PRI=.

CAUSE The account or user priority is not set as high as that requested in the PRI clause of the SYSTEMRUN command.

ACTION Use a lower priority for the PRI clause or have your System Administrator raise your MAXPRI value.

1203 MESSAGE Line number "line_number" does not exist.

CAUSE "line_number" specified in a command, for example,

LIST 1+10

where line number 1 is not a line in the current program.

ACTION Re-enter the command using a line number that is present in the current program.

1204 MESSAGE Label not found in current program unit.

CAUSE The label specified in a command, for example,

LIST Label1

where Label1 is not a label on a line in the current program.

ACTION Re-enter the command using a valid label.

1211 MESSAGE Command-only statements are not allowed in a COMMAND statement.

CAUSE In the interpreter, entries can be either a command, a statement in a program, or both. Those keywords that can only be commands, such as LIST, cannot occur in the quoted string literal or as the value of the string variable following the keyword COMMAND.

ACTION Remove the command from the quoted string literal or assign a value to the string variable that does not include the command.

1212 MESSAGE A command is not allowed here; it must be a program line. MODIFY only accepts lines which begin with a line number. (message #1606)

CAUSE Use of the MODIFY command was used to change a line to a command such as LIST or RUN.

ACTION A modified line must always begin with a line number. Make sure that there is a line number.

1216 MESSAGE File record size too small for program line.

CAUSE The record size of the ASCII file to which the listing is being directed at the time of a LIST, FIND, MODIFY, or CHANGE is less than the minimum size of 22 characters.

ACTION Increase the record size of the file to greater than 22 characters.

1217 MESSAGE Not enough room to make copy or move between lines "line_number_1" and "line_number_2".

CAUSE COPY or MOVE failed because the number of available lines between "line_number_1" and "line_number_2" is not sufficient.

ACTION Neither MOVE nor COPY will renumber lines so that they will automatically fit in the designated target area. If you want to move or copy lines and the range is not sufficient, use RENUMBER to renumber the line immediately following the location to which the lines are targeted, "line_number_2". Renumber to a high enough number to allow sufficient space.

1218 MESSAGE Cannot MOVE lines into a subunit which is moving.

CAUSE The actual message is: Lines are not contained in the same subunit. During editing, an attempt is made to MOVE at least one entire procedure or function and part of an additional one.

ACTION MOVE only entire internal procedures and functions.

1219 MESSAGE Lines "line_number_1"/"line_number_2" copied but then deleted because of the error.

CAUSE A syntax error encountered during partially completed COPY of lines.

ACTION Correct the syntax error and COPY again.

1220 MESSAGE Destination line, "line_number", lies within source range, "low_range"/"high_range".

CAUSE A range of lines was moved or copied, and the destination of one of the lines is a line number that would be in the range ["low_range","high_range"].

ACTION It is possible to obtain the same results using two moves:

1. MOVE the lines to another destination where they fit.
2. Delete the original lines.

3. MOVE the lines to the desired destination.

1221 MESSAGE Program not running.

CAUSE The CONTINUE command was entered when a program is not paused or halted.

ACTION Be certain that the program is paused or halted before entering CONTINUE. This can be done by entering the command: "LIST *" to display the current line.

1227 MESSAGE Current line is not defined.

CAUSE The command: "LIST *" was used when no program is running in the interpreter.

ACTION The "LIST *" command will only list the current line when a program is paused or halted in the interpreter.

1228 MESSAGE Command is too long.

CAUSE The quoted string literal or the value of the string expression following COMMAND exceeds the maximum allowed value of 500 characters.

ACTION Shorten the length of the string literal or the value of the string expression.

1229 MESSAGE Program lines are not allowed in a COMMAND statement.

CAUSE The quoted string literal or the value of the string expression following COMMAND cannot begin with a numeric value.

ACTION Change the string so that it begins with a character in the set [a..z] or [A..Z].

1230 MESSAGE Tried to delete current line while there are still line ranges left.

CAUSE A program that deletes the current line has been run. For example:

```
10 DELETE 10,20

20 PRINT "line to delete"
```

ACTION Do not include the current line in the range of lines to be deleted.

1231 MESSAGE Not enough data space available to start GET SUB.

CAUSE The total amount of space available for the current program has been exhausted. You may be able to do your own garbage collection to condense space.

ACTION Since you are trying to do a GET SUB from a BASIC SAVE formatted file, you should do garbage collection on:

1. The original current program file if it is stored in the BASIC SAVE file format.

2. All previous files involved in GET SUB commands or statements from which procedures or functions that are part of the current program at the time of the error were obtained. In order to do the actual garbage collection, you will need to do a GET, SAVE LIST, GET, and RESAVE sequence for each of the above files:

```
GET "original_filename"

SAVE LIST "new_filename"

GET "new_filename"

RESAVE "original_filename"
```

You can then PURGE "new_filename" and repeat the sequence with the next file.

ACTION If you obtained this message while running a program, then the run-time data structures used to store the values of variables might not be allocatable because of space constraints. Try to reduce the size of large arrays or eliminate unnecessary or unused variables.

1232 MESSAGE Could not renumber subunit during GET SUB. Last subunit removed.

CAUSE Renumbering the procedure or function to be part of the current program would result in a line number greater than the maximum line number, 999999.

ACTION Use RENUMBER to increase the number of available line numbers in the current program before the GET SUB statement or command.

Use a different first line number to begin the numbering of the new function or procedure in the current program.

Use a smaller line increment in the GET SUB statement or command so that the entire procedure or function fits in the

```
                         current program.
---------------------------------------------------------------------------
1233       MESSAGE     Could not add subunit name during GET SUB.  Last subunit
                       removed.
           CAUSE       The interpreter was unable to obtain sufficient space to add
                       the name of a procedure or function during execution of a GET
                       SUB command or statement.
           ACTION      Take action similar to that recommended for error number 1231.
---------------------------------------------------------------------------
1234       MESSAGE     Renumbering lines invalid for GET SUB statement.
           CAUSE       A line number in the current program conflicts with one of the
                       line numbers for the procedure or function to be read into the
                       interpreter from the BASIC SAVE file specified in the GET SUB
                       statement.  Either the new line number already exists or a line
                       in the current program would now be included in the new
                       procedure or function.
           ACTION      Be certain that the entire range of lines into which the new
                       procedure is to be read has no program lines prior to execution
                       of the GET SUB statement or command.
---------------------------------------------------------------------------
1235       MESSAGE     First subunit specified in GET SUB statement does not exist.
           CAUSE       Either no subunit number exists or the range of values that
                       supposedly corresponds to the subunit numbers does not exist in
                       the BASIC SAVE file for the GET SUB statement or command.
           ACTION      The numbering of the procedures and functions in the specified
                       BASIC SAVE file begins with one and each subsequent procedure
                       or function is one greater.  Be certain that the value that you
                       are using corresponds to this numbering system.
---------------------------------------------------------------------------
1236       MESSAGE     GET SUB statement requires a BASIC SAVE file.
           CAUSE       The file code for the file specified in the GET SUB command or
                       statement does not have a BSVXL file code.
           ACTION      GET the file with the subunit into the interpreter, PURGE the
                       file and use the SAVE command to save a new version of the file
                       with the correct file code.
---------------------------------------------------------------------------
1238       MESSAGE     GET SUB subunit specifications must be greater than 0.
           CAUSE       Either no subunit exists in the GETSUB file or the subunit
                       number specified is less than or equal to zero.
           ACTION      The numbering of the procedures and functions in the specified
                       BASIC SAVE file begins with one and each subsequent procedure
                       or function is one greater.  Be certain that the value that you
                       are using corresponds to this numbering system.
---------------------------------------------------------------------------
1240       MESSAGE     Line range "line_range" contains a nonexistent line reference.
           CAUSE       An attempt was made to save or resave a portion of the current
                       program that contains a main and at least one procedure or
                       multi-line function when the "line_range" of the line ranges
                       specified has no program lines and the file to which the
                       current program is being saved is a BASIC SAVE file.
           ACTION      SAVE or RESAVE the information to an ASCII file.
           ACTION      Use only line ranges which contain at lease one line when using
```

the line range option of the SAVE or RESAVE command when writing to a BASIC SAVE file.

Numbered Error Messages (1241 - 1738)

1241 MESSAGE Line range "line_range" contains no lines.

CAUSE Part of the current program was saved or resaved to a BASIC SAVE file using the line range list option. The range of lines selected did not contain any lines. For example, you might have typed:

```
SAVE "save_filename",SUB A/20
```

when the current program was:

```
20 PRINT B

30 SUB A
```

ACTION The line range option with SAVE to a BASIC SAVE file is restricted to saving MAIN and individual procedures and multi-line functions. If you wish to save individual parts of the current program using the line range option, SAVE or RESAVE to an ASCII file using the SAVE LIST or RESAVE command.

1242 MESSAGE Line number is not first line of subunit in line range "line_range".

CAUSE Part of the current program was saved or resaved to a BASIC SAVE file using the line range list option. The range of lines selected did not specify an entire main, procedure, or function. For example, you might have typed:

```
SAVE "save_filename",20
```

when the current program was:

```
10 PRINT A

20 PRINT B
```

ACTION The line range option with SAVE to a BASIC SAVE file is restricted to saving MAIN and individual procedures and multi-line functions. If you wish to save individual parts of the current program using the line range option, SAVE or RESAVE to an ASCII file using the SAVE LIST or RESAVE command.

1243 MESSAGE Line number is not last line of subunit in line range "line_range".

CAUSE Part of the current program was saved or resaved to a BASIC SAVE file using the line range list option. The range of lines selected did not specify an entire main, procedure, or function. For example, you might have typed:

```
SAVE "save_filename",10
```

when the current program was:

```
10 PRINT A

20 PRINT B
```

ACTION The line range option with SAVE to a BASIC SAVE file is restricted to saving MAIN and individual procedures and multi-line functions. If you wish to save individual parts of the current program using the line range option, SAVE or RESAVE to an ASCII file using the SAVE LIST or RESAVE command.

1245 MESSAGE No SAVE/RESAVE done, program contains no lines.

CAUSE There are no lines in the current program to be saved.

ACTION Enter a program line in the interpreter prior to doing a SAVE or RESAVE.

1246 MESSAGE No SAVE/RESAVE done, default file does not exist.

CAUSE The current program has been saved or resaved, but no file name has been specified either for the current program or in the command.

ACTION Use the NAME command to name the current program and then do a SAVE or RESAVE.

ACTION Specify the name of the file to which the current program is to be saved as par of the SAVE or RESAVE command.

1247 MESSAGE RESAVE file must be of type BASIC SAVE, BASIC DATA, or ASCII

CAUSE The current program has been saved to a file which has no file code and is not an ASCII file or has a file code other than BSVXL or BDTXL.

ACTION Correct the spelling of the name of the current program in the interpreter using the NAME command and then do a SAVE or RESAVE.

ACTION Choose another filename to which to save the current program. Use SAVE LIST "filename", SAVE BDATA "filename" or SAVE "filename" to save the current program to the alternative filename.

1248 MESSAGE File type given doesn't match the type of the file named/implied.

CAUSE The current program has been resaved using the LIST option to a file that is not an ASCII file or using the BDATA option to a file that does not have a BSVXL file code.

ACTION Use RESAVE LIST to resave the current program to an ASCII file. Use RESAVE BDATA to store the current program to a file that has a BSVXL file code.

1249 MESSAGE No RESAVE done, unable to purge the old file.

CAUSE The RESAVE command failed either because of simultaneous file access by two users or because of a serious file system problem.

ACTION Retry the command. If it does not work a second time, use SAVE to save the current program. Try to purge the old file using the PURGE command and then rename the newly created file with the RENAME command. If this does not work correctly, contact your System Administrator.

1250 MESSAGE Execution label specified is not in the current program unit.

CAUSE This is a substitute message for error number 3: The parameter to the RUN command specifying the line on which execution is to begin is not present in the current program. For example, if the current program has no lines then RUN; Label will generate this error.

ACTION Be certain that the line label specified exists in the current program.

1253 MESSAGE Line number is invalid or missing, GET terminated.

CAUSE This occurs in the interpreter only. During the GET of an ASCII file a line in the file has been encountered which does not begin with a numeric literal that should correspond to a line number.

ACTION Use the SYSTEMRUN command from within BASIC to run an editor. Change the line displayed during the GET so that it begins with a numeric literal. Save the new version, exit the editor, and use the GET command to make the ASCII file the current program file.

1254 MESSAGE Line number would be illegal if renumbered, GET terminated.

CAUSE A GET of a program would result in a line number beyond the maximum line number in the interpreter, 999999.

ACTION Check the last line number and the range of the lines to be included in the current program prior to using GET. The RENUMBER command can create additional space in the interpreter by reducing the range of the program line numbers before using GET.

1260 MESSAGE Found 11th named COMMON area in subunit "subunit_name", only 10 are allowed.

CAUSE More than the maximum number of COMMON areas have been declared in the current MAIN, procedure, or function. The "subunit_name" is either MAIN or the name of the procedure or function in which the error occurred.

ACTION Delete one of the named COMMON areas in the specified subunit. If the variables have to be declared in a COMMON, add them to one of the named COMMON areas that are already declared or pass them to the procedure or function as parameters.

1261 MESSAGE Not enough memory available for the existing COMMON declarations.

CAUSE The total COM area declared in the current main, procedure, or function exceeds the maximum space allocatable to common declarations but is less than the total amount of space allocatable to all declarations.

ACTION Reduce the number of declarations in the COM area to include only those absolutely necessary for all procedures or functions that include that COM area. All variables not required in all can either be declared locally or passed as parameters.

1262 MESSAGE Unable to allocate enough memory for the COMMON variables.

CAUSE HP Business BASIC/XL was unable to allocate enough memory for the COMMON variables.

ACTION Reduce the number of declarations in the common areas. Reduce the size of the common area.

1270 MESSAGE Arrays need to be of the same type.

CAUSE The type of the floating point numeric argument to the matrix built-in functions, CSUM, INV, MUL, RSUM, and TRN must be the same as the type of the numeric target of the MAT assignment statement in which the built-in occurs. Both of the arguments to the DOT operation must be the same numeric type.

ACTION Coerce the argument prior to performing the built-in operation or coerce the result after the assignment.

1271 MESSAGE Wrong type for inverse.

CAUSE The array that is the argument for the INV matrix built-in function is not of type SHORT DECIMAL, DECIMAL, SHORT REAL or REAL.

ACTION Correct the type of the argument so that it is one of the floating point types supported in HP Business BASIC/XL.

1272 MESSAGE Could not purge the temporary save file, system error #"error_number".

CAUSE During execution of one of the statements or commands that requires the creation of a temporary BASIC SAVE file, a file was created and could not be purged following use. The statements requiring creation of a BASIC SAVE file for this purpose are COMPILE, COMPGO, and COMPLINK.

ACTION Note the return value of "error_number" and report the problem to your System Administrator. If you see this error message, check the contents of your temporary file space with the :LISTFTEMP command from within HP Business BASIC/XL. If you are unable to purge the files either from within HP Business BASIC/XL or the Command Interpreter of MPE, then log off and log on again.

1273 MESSAGE Could not "message" while compiling, system error # "system_error_number".

CAUSE message corresponds to the command that the interpreter could not process when trying to run the compiler:

```
FILE BBCTEXT="text_filename"
FILE BBCOBJ="obj_or_rl_filename"
FILE BBCLIST="list_filename"
RESET BBCTEXT="text_filename"
RESET BBCOBJ="obj_or_rl_filename"
RESET BBCLIST="list_filename"
```

"system_error_number" is the error number returned from the Command intrinsic.

ACTION These errors should only occur when the system is having trouble performing fundamental file operations. Try to repeat the same operation from the interpreter. If you again run into problems, contact your System Administrator.

1274 MESSAGE Could not "message" during link, error #"error_number".

CAUSE message corresponds to a command that the interpreter could not process when trying to link a compiled program. The interpreter must build a stdin and stdlist file for the linkeditor. The following commands are related to file manipulation:

```
BUILD the stdlist file "stdlist_filename"
BUILD the stdin file "stdin_filename"
FILE PFILE="stdin_filename",OLDTEMP
FILE LFILE="stdlist_filename",OLDTEMP
RESET PFILE
RESET LFILE
PURGE "stdin_filename",TEMP
PURGE "stdlist_filename",TEMP
```

For this set of messages, "error_number" is the error number returned from the command intrinsic indicating the system error encountered.

The interpreter must also enter command information into the stdin file. The following messages reflect possible file system problems encountered:

```
FOPEN the stdin file
FWRITE to the stdin file
FCLOSE the stdin file
```

For this set of messages, "error_number" is the file system error number.

ACTION These errors should only occur when the system is having trouble performing fundamental file operations. Try to repeat the same operation from the interpreter. If you again run into problems, contact your System Administrator.

CAUSE message corresponds to a problem encountered when trying to run the linkeditor to link the program. The problem can either be the result of a problem with the program or with the system. If the message is only: system run of linkeditor then try to repeat the error. If you can do so, this indicates a system problem and should be discussed with your System Administrator.

If the message is preceded by the output from the segmenter explaining the problems with the program, then you should consult the *HPLink Editor/XL Reference Manual*.

error_number is the JCW following the execution of the linkeditor program.

ACTION If the segmenter or linkeditor is aborting then repeat the same operation from the interpreter. If you again run into problems, contact your System Administrator.

For the second class of problems, look carefully at the error output from the segmenter or linkeditor, consult the appropriate reference manual and fix the problem.

1275 MESSAGE Terminal does not recognize escape sequences used by BASIC for terminal control.

CAUSE One of the HP Business BASIC/XL statements dependent on terminal interactions, such as FORM statements, CURSOR statements, or KEY statements, is being used on a terminal that does not return the correct escape sequences when queried or on a system on which the configuration file, HPBBCNFG.PUB.SYS, specifies that the terminal is not compatible with BASIC's terminal-specific features.

ACTION If you are working on a terminal that is compatible with HP Business BASIC/XL's terminal-specific features, check the configuration file.

ACTION If you are working on a terminal that is not compatible with HP Business BASIC/XL's terminal-specific features, you will not be able to use these features.

1276 MESSAGE Invalid KEY number. Must be between 1 and 8.

CAUSE The value of a numeric literal or a numeric expression corresponding to one of the keys specified in an ON KEY, OFF KEY, or PRESS KEY statement is not within the range [1,8].

ACTION Correct the value of the numeric literal or, if it is an expression, perform run-time checking of the value of the numeric variable or numeric expression prior to using that value in the statement.

1277 MESSAGE File is not a BKEY file.

CAUSE The name of the file specified in the GET KEY or RESAVE KEY statement does not have a BKEY file code.

ACTION Be certain that the file to which you are trying to resave keys does not exist or has a BKEY format.

ACTION Be certain that the file from which you are trying to obtain key definitions using the GET KEY statement has a BKEY format.

1278 MESSAGE Invalid PRIORITY number. Must be between 1 and 15

CAUSE When this occurs in the interpreter, this syntax error occurs because the numeric literal in the PRIORITY clause of the ON KEY statement is not within the range [1,15].

When this occurs at run-time, the value of the numeric variable in the PRIORITY clause of the ON KEY statement is not within the range [1, 15].

ACTION Re-enter the line using a valid numeric literal value.

ACTION Check the value of the numeric variable before using it in the PRIORITY clause.

1279 MESSAGE Invalid CURSOR parameter.

CAUSE One of the parameters for the position option is not within the limits of current screen memory on the terminal being used.

CAUSE The parameter specifying the screen enhancement has an invalid letter specified as a requested enhancement.

ACTION Be certain that the maximum value for the cursor position is within bounds of the current screen memory.

ACTION Be certain that the screen enhancement specified is one of: h, i, b, u, H, I, B, U, or the empty string.

1283 MESSAGE Program Analyst cannot be used if OUTPUT or SYSTEM OUTPUT are redirected.

CAUSE The [SEND] OUTPUT [TO] or [SEND] SYSTEM OUTPUT TO statement has been used to redirect the output to a device other than the reserved word DISPLAY or the system file $STDLIST.

ACTION Use the INFO command to determine the device to which the output is being sent and correctly reset the output to be directed to DISPLAY.

1284 MESSAGE Valid arguments for ANALYST command are M, O, E, S, G, C, D and P.

CAUSE Incorrect entry following the ANALYST command.

ACTION Re-enter the command followed by one of the valid characters displayed. These characters select which of the ANALYST's screens will be initially displayed.

1285 MESSAGE Program Analyst cannot be used when program contains no lines.

CAUSE The ANALYST command has been used when there is no program in the interpreter.
ACTION Only use the ANALYST when there is a program in the interpreter.

1286 MESSAGE Program Analyst can run on HP-supported terminals only.

CAUSE The ANALYST command has been used from a terminal that HP Business BASIC/XL does not recognize as fully supporting features such as CURSOR and ON KEY.

ACTION Use a different terminal. Also check to see if a configuration file that specifies that the terminal is not HP-supported has been used.

1287 MESSAGE Program Analyst cannot be run in batch mode.

CAUSE An ANALYST command has been encountered by the interpreter while in batch mode.

ACTION Remove the ANALYST command from the stream file.

1288 MESSAGE Program Analyst cannot be used while program is running.

CAUSE An ANALYST command has been used while a program was paused (either by the PAUSE statement or by an unhandled error).

ACTION Issue the STOP command before the ANALYST command.

1289 MESSAGE Program Analyst cannot be used with VERIFY errors in program.

CAUSE The ANALYST command was used when the program in the interpreter contained structural errors. Some examples are a WHILE without ENDWHILE or an ELSE without IF.

ACTION Modify the program so that there are no VERIFY errors.

1291 MESSAGE No VPLUS Form open.

CAUSE A Forms I/O statement has been used when no form is currently active.

ACTION Be sure that the OPEN FORM statement is executed before any Forms I/O operation is attempted.

1292 MESSAGE Bad form name "formname".

CAUSE The string in quotes was used in a OPEN FORM statement, but is not a valid form name.

ACTION Check the syntax and correct the program.

1293 MESSAGE "filename" is not a forms file.

CAUSE The filename in quotes is an existing file, but not a VPLUS forms file.

ACTION Use the correct filename for the forms file.

1294 MESSAGE Normal Input and Output cannot occur when a VPLUS form is open.

CAUSE A PRINT, DISP, INPUT, or similar operation has been executed while a VPLUS form is active.

ACTION If a form has been left active inadvertently, then insert a CLOSE FORM statement. Otherwise, use the READ FORM and WRITE FORM statements.

1295 MESSAGE Problem with reading from or writing to the terminal.

CAUSE A VPLUS error occurred while the program was reading from or writing to a terminal.

ACTION Check the hardware connection.

1296 MESSAGE Too many items specified in WRITE FORM or READ FORM.

CAUSE More items have been specified in a WRITE FORM or READ FORM statement than the declared in the FORM.

ACTION Check the FORM specification or the FORM I/O statement.

1297 MESSAGE Form field not big enough for item "num".

CAUSE The value of the item specified in the WRITE FORM will overflow the declared type for the FORM.

ACTION Correct the range or length of the item specified in the FORM.

ACTION Correct the value of the item in the WRITE FORM statement.

1298 MESSAGE Form field not big enough for item "num", subitem "array element num"

CAUSE The value of the array element used in this item number specified in the WRITE FORM statement will overflow the declared type for the FORM.

ACTION Correct the range or length of the item specified in the FORM.

ACTION Correct the value of the array element for the item in the WRITE FORM statement.

1299 MESSAGE String not big enough for form field for item "num".

CAUSE The string item used in the WRITE FORM statement cannot fit into the FORM field.

ACTION Correct the string length in the FORM specification or correct the string item in the WRITE FORM statement.

1340 MESSAGE NLS not installed.

CAUSE Native Language Support is not supported on your system.

ACTION You must log on as MANAGER.SYS and run the utility program, LANGINST.PUB.SYS to install native languages on your system. Please refer to the *Native Language Programmer's Guide* for more information about installing native languages.

1341 MESSAGE Native language #"num" is not configured.

CAUSE The specified native language is not configured.

ACTION You must log on as MANAGER.SYS and run the utility program, LANGINST.PUB.SYS to add the language number to the configuration file. Please refer to the *Native Language Programmer's Guide* for more information about adding native languages on your system.

1342 MESSAGE Illegal language specification for LEX.

CAUSE The language number specified in the LEX function is less than -1.

ACTION Make sure that the language number is greater than or equal to -1.

1356 MESSAGE Error in calculating new break limit for BREAK...WHEN...BY statement.

CAUSE An arithmetic error overflow or underflow occurred while trying to calculate the next multiple value to BREAK the report.

ACTION Verify the value specified on the BY clause. It is approaching the limit of real for OPTION REAL, or approaching the limit of decimal for OPTION DECIMAL.

1400 MESSAGE Uninitialized variable or array element used (!).

CAUSE When OPTION NOINIT is on, numeric variables are not initialized to zero. Referencing the uninitialized variable will result in a run time error in the interpreter.

ACTION Initialize the variable or array element or take out OPTION NOINIT. OPTION INIT is the default.

1403 MESSAGE Undeclared variable "name" found in subunit "name".

CAUSE When OPTION DECLARE is on, implicit variable declaration is illegal.

ACTION Either declare the variable or take out OPTION DECLARE. OPTION NO DECLARE is the default.

1404 MESSAGE Single line function and current subunit have the same name.

CAUSE A single-line function and the current subunit have the same name.

ACTION Change the single-line function name or the current function name.

1411 MESSAGE Parameter to DATE$ must be >= -2.

CAUSE The native language number specified in the call to DATE$ is less than -1.

ACTION Correct the native language number.

1413 MESSAGE Parameter to TIME$ must be >= -1.

CAUSE The native language number specified in the call to TIME$ is less than -1.

ACTION Correct the native language number.

1414 MESSAGE Language parameter to UPC$ or LWC$ must be >= -1.

CAUSE The native language number specified in the call to UPC$ or LWC$ is less than -1.

ACTION Correct the native language number in the second parameter.

1415 MESSAGE The form specified does not exist.

CAUSE The form name specified in the OPEN FORM statement does not exist in the forms file. For example,

```
10 OPEN FORM "MAIN: form1.test"
```

The form name, MAIN, does not exist in "form1.test".

ACTION Check the form name and the form file name.

1416 MESSAGE Field data reformatting (finishing) failed. (VPLUS #"error-num")

CAUSE An error occurred while processing specifications defined for the final phase of fields editing. Please refer to *Data Entry and Forms Management System VPLUS/3000* for a detailed description of the "error-num" returned in VFINISHFORM intrinsic.

ACTION Please read the ACTION section regarding "error-num" returned in VFINISHFORM intrinsic.

1417 MESSAGE CURSOR positioning failed. (VPLUS #"error-num")

CAUSE An error occurred in CURSOR positioning. Please refer to *Data Entry and Forms Management System VPLUS/3000* for a detailed description of "error-num" returned in VPLACECURSOR intrinsic.

ACTION Please read the ACTION section regarding the "error-num" returned in VPLACECURSOR intrinsic.

1418 MESSAGE Incompatible version of VPLUS installed.

CAUSE A current version of VPLUS is not installed.

ACTION Please consult the system manager.

1419 MESSAGE Field initialization data errors detected. (VPLUS #"error-num")

CAUSE An error occurred in data initialization. Please refer to *Data Entry and Forms Management System VPLUS/3000* for a detailed description of "error-num" returned in the VINITFORM intrinsic.

ACTION Please read the ACTION section regarding "error-num" returned in the VINITFORM intrinsic.

1420 MESSAGE Field editing data errors detected. (VPLUS #"error-num")

CAUSE An error occurred in data editing. Please refer to *Data Entry and Forms Management System VPLUS/3000* for a detailed description of "error-num" returned in the VFIELDEDITS intrinsic.

ACTION Please read the ACTION section regarding "error-num" returned in the VFIELDEDITS intrinsic.

1421 MESSAGE "error-num" (HPDERR "error-message").

CAUSE An error occurred in one of the HPDialog routines. Please refer to the *HPDialog Reference Manual* for a detailed description of "error-num" returned.

ACTION Please read the ACTION section regarding "error-num" returned in HPDialog intrinsic routines.

1490 MESSAGE VERIFY command not allowed while program is running.

CAUSE A VERIFY command has been used while a program was suspended (either by the PAUSE statement or by an unhandled error).

ACTION Issue a STOP command to stop your program.

1491 MESSAGE FORMATTED option to LIST not allowed while program is running.

CAUSE A LIST command with the FORMATTED option has been used while the program was suspended (either by the PAUSE statement or by an unhandled error).

ACTION Issue a STOP command to stop your program.

1492 MESSAGE XREF command not allowed while program is running.

CAUSE An XREF command has been used while the program was suspended (either by the PAUSE statement, by hitting HALT key, or by an unhandled error).

ACTION Issue a STOP command to stop your program.

1496 MESSAGE GETHEAP failure in library.

CAUSE This is a heap management problem. The system does not have enough memory to run the program. The most probable case is that the program is too big or too many variables are used in the program.

ACTION Break the program up into smaller subprograms or use fewer variables.

ACTION You need to do garbage collection by saving the program in ASCII format and GET the program again in the interpreter.

1497 MESSAGE RTNHEAP failure in library.

CAUSE This is a heap management problem due to an internal problem.
ACTION Further investigation of this problem is required. Please contact your Hewlett-Packard representative.

1498 MESSAGE Catastrophic program error.

CAUSE This is an internal problem.

ACTION Further investigation of this problem is required. Please contact your Hewlett-Packard representative.

1499 MESSAGE Catastrophic error in heap management (unable to return heap space).

CAUSE This is a heap management problem due to internal problems.

ACTION Further investigation of this problem is required. Please contact your Hewlett-Packard representative.

1500 MESSAGE Internal consistency check #"num" failed. Report the problem to HP.

CAUSE This is an internal problem.

ACTION Further investigation of this problem is required. Please contact your Hewlett-Packard representative.

1502 MESSAGE WARNING "warning message".

CAUSE This warning message is generated by the SYSTEM statement or ":" as a system command. When the STATUS clause is specified in the SYSTEM statement, you will not see the warning message.

ACTION None.

1503 MESSAGE WARNING 1503: Default STACKsize from program "prog-name".

CAUSE The stacksize specified in the SYSTEMRUN statement was less than 512.

ACTION None. The default stacksize is used to run the program.

1504 MESSAGE WARNING 1504: Default DLsize from program "prog-name".

CAUSE The dlsize specified in the SYSTEMRUN statement was less than zero.

ACTION None. The default dlsize is used to run the program.

1505 MESSAGE WARNING 1505: Default MAXDATA from program "prog-name".

CAUSE The maximum stack area value, MAXDATA, specified in the SYSTEMRUN statement is less than or equal to zero.

ACTION None. The default configuration maximum is used to run the program.

1506 MESSAGE WARNING 1506: DLsize rounded up to next 128 word multiple in program "prog-name".

CAUSE The dlsize specified in the SYSTEMRUN statement was not a multiple of 128.

ACTION None. The new dlsize value is used to run the program.

1507 MESSAGE WARNING 1507: MAXDATA decreased to configuration maximum in program "prog-name".

CAUSE The maximum stack area value, MAXDATA, specified in the SYSTEMRUN statement is larger than configured maximum for your system.

ACTION None. MAXDATA is decreased to the configured maximum.

1508 MESSAGE WARNING 1508: MAXDATA increased to DLsize + globsize + STACKsize in program "prog-name".

CAUSE The maximum stack area value, MAXDATA, specified in the SYSTEMRUN statement is smaller than the minimum required to run the program.

ACTION None. MAXDATA is increased to run the program.

1509 MESSAGE WARNING 1509: "parm-name" was specified more than once, last value taken.

CAUSE A parameter to the SYSTEMRUN statement was specified more than once.

ACTION None. Only the last value for that particular parameter is used. The rest are ignored.

1511 MESSAGE WARNING 1511: Extra semicolon ignored.

CAUSE An extra delimiter has been entered in the SYSTEMRUN statement.

ACTION None. The extra semicolon is ignored.

1536 MESSAGE WARNING 1536: The spelling of that "string" was corrected

CAUSE This is a HELP command warning in the interpreter.

ACTION The spelling of the "string" entered in the HELP command is corrected to the topic that is closest in spelling to the "string" entered. Help information is provided on the corrected topic.

1539 MESSAGE WARNING 1539: The spelling of that "string" was truncated

CAUSE This is a HELP command warning in the interpreter.

ACTION None.

1564 MESSAGE UnSAVEd source modifications will be lost. Do you really want to EXIT? Y

CAUSE In the interpreter, the EXIT command was issued before a program that was modified had been saved.

ACTION Save the program first or press the return key to exit without saving the modifications.

1602 MESSAGE Modify command found at or past continuation character. Try again

CAUSE A line was modified at or past the continuation character, "&".

ACTION Redo the modification.

1603 MESSAGE Line "linenum" is busy and cannot be changed.

CAUSE A "linenum" is busy if one of the following condition is true:

The line made a call which has not returned.

The line was interrupted with the halt key before it finished executing.

ACTION Change or modify the line when it has finished execution.

1605 MESSAGE Line "linenum" secured and cannot be modified.

CAUSE The line number specified is secured and cannot be modified. A secured line cannot be listed, only an asterisk is displayed.

ACTION Re-enter the line.

1608 MESSAGE TRACE VARS list can only be used in a program or when PAUSEd.

CAUSE A TRACE VARS list is not allowed in COMMAND mode.

ACTION Use TRACE VARS list in a program.

1733 MESSAGE The formal parameter space request for "proc_name", "num_bytes", exceeds the maximum value available of "limit_num_bytes" bytes.

CAUSE During processing of the EXTERNAL and INTRINSIC, the formal parameter space for the definition exceeds the maximum value allowable.

ACTION Reduce the number of formal parameters to the external, or redefine the intrinsic file definition so that fewer parameters are required.

1734 MESSAGE Unable to close intrinsic file "intr_file_name.

CAUSE An invalid file system value was returned when trying to close the intrinsic file, "intr_file_name".

ACTION This is a file system problem that might be circumvented by terminating all processes that reference the file.

1735 MESSAGE Error reading intrinsic file "intr_file_name".

CAUSE There is either invalid parameter information in the intrinsic file for the parameter or a file system problem.

ACTION Rebuild the intrinsic file.

1736 MESSAGE Parameter #parm_num to procedure proc_name is invalid as a formal parameter.

CAUSE The formal parameter obtained from the intrinsic file is not a type that is supported in HP Business BASIC/XL. The types supported in HP Business BASIC/XL are 16 bit integer, 32 bit integer, 32 bit IEEE floating point, 64 bit IEEE floating point, and string.

ACTION If possible, define the intrinsic as an external and use a formal parameter that has the same size as that in the intrinsic file.

1737 MESSAGE Error reading intrinsic file "intr_file_name" while processing procedure information for proc_name.

CAUSE There is either invalid procedure information in the intrinsic file or a file system problem.

ACTION Rebuild the intrinsic file.

1738 MESSAGE Type mismatch for parameter #parm_num, formal parameter type is formal_parm_type while actual parameter type is actual_parm_type.

CAUSE The type of the formal and actual parameters do not match.

ACTION Correct the actual parameter number parm_num so that it is formal_parm_type type.

Numbered Error Messages (1739 - 2103)

1739 MESSAGE The dimensionality of formal and actual parameter #parm_num do not match.

CAUSE Scalar formal parameter and array actual parameter or vice versa do not match, or formal and actual array parameters do not have the same number of dimensions.

ACTION Correct actual parameter number parm_num so that it is the same dimensionality as the formal parameter.

1740 MESSAGE Actual parameter #parm_num to be passed by reference is not a variable.

CAUSE A literal or an expression actual parameter is being passed where the formal parameter specifies that a variable by reference is required.

ACTION Assign the value of the literal or expression to a variable of the type and dimensionality that corresponds to that of the formal parameter. Substitute the variable for the literal or expression that is parameter number parm_num.

1741 MESSAGE Missing actual parameter without default value specified for formal parameter #parm_num of type parm_type.

CAUSE A required parameter in the call is missing. When HP Business BASIC/XL attempts to provide the default value, none is present. The error will occur when a parameter is missing:

```
     CALL Ext(A,,B)
```

For example, the error will occur when three parameters are required and only two are provided:

```
     CALL Need_three(A,B)
```

Normally, HP Business BASIC/XL will provide the defaults that allow the externals to be called by using the information present in the intrinsic file.

ACTION Add the default parameter to the definition in the intrinsic file. Supply the actual parameter, parm_num, of type parm_type.

1742 MESSAGE Invalid formal parameter type for parameter #parm_num: parm_type.

CAUSE The actual parameter number parm_num does not have a type that corresponds to one of the HP Business BASIC/XL data types.

ACTION Change the definition in the intrinsic file or use the default by leaving out the actual parameter in the actual parameter list.

```
--------------------------------------------------------------------------

1743     MESSAGE     Non-numeric or non-scalar actual parameter #parm_num cannot be
                       passed by value.

         CAUSE        The actual parameter number parm_num is either non-numeric or
                       is not a scalar parameter.

         ACTION       Only scalar numeric values can be passed by value.  Alter the
                       actual parameter so that it is a scalar.

--------------------------------------------------------------------------

1744     MESSAGE     The actual parameter space requested for the call to proc_name
                    exceeds the maximum value of num_words at parameter #parm_num.

         CAUSE       The space allocated by the interpreter for parameter space was
                       exhausted when loading parameter number parm_num during the
                       call to proc_name.

          ACTION       Reduce the number of actual parameters for the call to
                       proc_name.

--------------------------------------------------------------------------

1745      MESSAGE     The structure of the HP Business BASIC/XL string array actual
                    parameter is incompatible with the formal parameter #parm_num.

          CAUSE        An HP Business BASIC/XL string array was passed to a non-HP
                       Business BASIC/XL external.

          ACTION       Assign the element of the string array to a scalar string.
--------------------------------------------------------------------------

1746      MESSAGE      Actual parameter #parm_num to be passed by anyvar is not a
                       variable.

         CAUSE       The intrinsic file specifies that formal parameter parm_num for
                       the call to the intrinsic must be a variable passed by
                       reference.

          ACTION       Assign the value of the literal or expression to a variable.
                    Substitute that variable for the literal or expression that is
                       parameter number parm_num.

--------------------------------------------------------------------------

1747      MESSAGE      The type of the value returned by function func_name,
                       function_return_type, has no equivalent type in Business
                       BASIC/XL.

         CAUSE        function_return_type is not a valid HP Business BASIC/XL data
                       type that can be returned by a function.

          ACTION       Use the CALL statement to call the function without a return
                       value.

--------------------------------------------------------------------------

1748      MESSAGE      The type or dimensionality of formal and actual parameter
                       #parm_num do not match.

          CAUSE        Either the type or dimensionality of the formal and actual
                       parameters do not match.

         ACTION       Check formal parameter number parm_num to be certain that the
                       corresponding actual parameter in the call has both the same
                       type and dimensionality.

--------------------------------------------------------------------------

1749     MESSAGE     ALIAS name provided, alias_name, exceeds the maximum length of
                       max_length characters.

          CAUSE        The length of an alias name is too long.
```

ACTION Reduce the length of the alias name.

1750 MESSAGE Formal parameter #parm_num of type parm_type to procedure proc_name has no corresponding Business BASIC/XL data type and no default value.

CAUSE An error occurred while processing the definition of the intrinsic proc_name, specifically while looking up information for formal parameter number parm_num. HP Business BASIC/XL will not be able to call proc_name because the data type of the formal parameter has no corresponding type and no default value is supplied.

ACTION Redefine the external proc_name in a new intrinsic file and supply the default.
Define the external proc_name in an EXTERNAL statement and supply the appropriate formal parameters.

1751 MESSAGE A BYTE type array parameter is an invalid data type for formal parameter #parm_num.

CAUSE An external definition contained a BYTE type parameter. For example,

```
EXTERNAL A(BYTE VALUE(A(*))
```

BYTE keywords are not allowed.

ACTION Remove the BYTE keyword from the list of formal parameters.

1752 MESSAGE BYTE type is an invalid data type for formal parameter #parm_num to an Business BASIC/XL external.

CAUSE BYTE type formal parameters are not valid in an HP Business BASIC/XL external definition.

ACTION Remove the BYTE keyword form the list of formal parameters.

1753 MESSAGE A scalar BYTE reference parameter is an invalid data type for formal parameter #parm_num.

CAUSE A reference parameter of type BYTE is invalid. For example,

```
EXTERNAL PASCAL A(BYTE A)
```

ACTION Change the external's definition so that parameter number parm_num is passed by value.

1754 MESSAGE Array formal parameter #parm_num by value is invalid.

CAUSE Only scalar formal parameters can be passed by value.

ACTION Remove the VALUE form the definition of the formal parameter number parm_num in the formal parameter list.

1755 MESSAGE String or BYTE string formal parameter #parm_num by value is invalid.

CAUSE An external definition has a string parameter passed by value. For example,

```
EXTERNAL PASCAL A(VALUE A$)

EXTERNAL PASCAL B(BYTE VALUE A$)
```

ACTION Remove the VALUE keyword from the definition of the formal parameter number parm_num.

1756 MESSAGE Intrinsic filename "intr_file_name" exceeds the maximum filename length of max_file_name_length characters.

CAUSE An invalid filename has been provided.

ACTION Correct the name of the file.

1757 MESSAGE Formal parameter #parm_num passed by value to procedure proc_name has incorrect default size of num_bytes bytes.

CAUSE The information for the default size supplied in the intrinsic file contains an error.

ACTION Rebuild the intrinsic file.

1758 MESSAGE Formal parameter #parm_num passed by reference to procedure proc_name has incorrect default size of num_bytes bytes.

CAUSE The information for the default size supplied in the intrinsic file contains an error.

ACTION Rebuild the intrinsic file.

1759 MESSAGE Unable to lead procedure proc_name when searching in library list beginning with xl_name.

CAUSE The entry point name specified does not exist in any of the libraries in the library list.

ACTION Check the spelling of the external. Use the linkeditor to check the names of the entry points in the libraries.

1760 MESSAGE Procedure or function proc_name not found in intrinsic file "intr_file_name".

CAUSE The intrinsic entry does not exist in the intrinsic file, intr_file_name.

ACTION Check the entries in the intrinsic file to be certain that the entry exists.

1761 MESSAGE EXTENSIBLE value provided exceeds the valid range of lower_bound to upper_bound.

CAUSE The value following the EXTENSIBLE keyword is not within the bounds of lower_bound to upper_bound.

ACTION Change the definition so that the value is within the specified range.

1762 MESSAGE External procedure proc_name has been previously defined.

CAUSE The procedure or function name to be used to call the external, proc_name, is not unique in the main, procedure, or function. Calls to the procedure or function will be ambiguous.

ACTION Change the spelling of one of the names of the procedures in the external or intrinsic definitions.

1763 MESSAGE External name provided, proc_name, exceeds the maximum length of max_num_characters .

CAUSE The external name length is too long.

ACTION Shorten the name so that it is less than max_num_characters in length.

1764 MESSAGE Intrinsic intr_name parameter parm_num by reference has invalid address type specification of address_type.

CAUSE Parameter number parm_num of the intrinsic intr_name has an invalid address type specified in the intrinsic file.

ACTION Rebuild the intrinsic file.

1765 MESSAGE External proc_name has an entry point name, ent_point_name, returned from the intrinsic file, that exceeds the maximum of max_num_chars characters.

CAUSE The entry point name is too long.

ACTION Rebuild the intrinsic file using a shorter entry point name.

1766 MESSAGE The actual parameter space request for the call to proc_name of num_words words exceeds the maximum value of max_num_words words.

CAUSE An ANYPARM call requires num_words words of parameter space when only max_num_words words are available.

ACTION Each actual parameter in the call to the ANYPARM procedure requires two words of actual parameter space. Reduce the number of actual parameters to the call.

1800 MESSAGE WARNING 1800: No closing quotation mark found!.

CAUSE A string literal had no closing quotation mark.

ACTION None. The interpreter will insert the missing quotation mark.

1801 MESSAGE WARNING 1801: String too long; re-enter from item "item-no"

CAUSE A string that is longer than the declared string variable was entered.

ACTION Re-enter a string that is within the declared length or modify the program to extend the declared string length.

1802 MESSAGE WARNING 1802: Input too long. Please re-enter.

CAUSE A string that is longer than the declared string variable in the INPUT statement has been entered.

ACTION Re-enter a string that is within the declared length or modify the program to extend the declared string length.

1804 MESSAGE WARNING 1804: The file "filename" did not previously exist

CAUSE A file that did not previously exist has been resaved.

ACTION None. The program will be saved with the SAVE command, rather than the RESAVE command.

1805 MESSAGE MESSAGE 1805: Statement not implemented in HPBB (at character "char-num")

CAUSE The program uses an unimplemented feature.

ACTION Do not use the unimplemented feature.

1806 MESSAGE WARNING 1806: Name at character "number" too long. Name Truncated.

CAUSE The identifier specified is longer than 64 characters.

ACTION None. The identifier is truncated to 64 characters.

1807 MESSAGE WARNING 1807: Bad numeric input; re-enter from item "item-no"

CAUSE The value entered does not match the type of the numeric variable in the INPUT statement.
ACTION Re-enter the correct numeric value.

1809 MESSAGE WARNING 1809: The PROTECT word "string" was truncated to "string".

CAUSE The lockword specified is longer than 8 characters.

ACTION None. The new lockword is truncated to 8 characters.

1811 MESSAGE WARNING 1811: COMMON area name too long, truncated to "string".

CAUSE The common name specified is longer than nine characters.

ACTION None. The new name is truncated to nine characters.

1812 MESSAGE WARNING 1812: This statement is not compilable.

CAUSE This is a warning message issued by the interpreter command, CWARNINGS, that lists statements that are not compilable. Any statement that modifies an HP Business BASIC/XL program at run time or requires the interpreter environment cannot be compiled.

This warning occurs with the following statements:

COMMAND GET MERGE SCRATCH DEFAULT GETSUB RESAVE SECURE DELETE LINK

ACTION The following change will prevent the execution of a non-compilable statement in a compiled program:

100 GET "abc"

change to

100 IF INTERPRETED THEN GET "abc"

1813 MESSAGE WARNING 1813: This statement is not compilable. (generates no code)

CAUSE This is a warning message generated by the interpreter command, CWARNINGS, that lists statements that cause compiler warnings. These statements are primarily for debugging and the compiler does not generate any code for them.

ACTION None.

1814 MESSAGE WARNING 1814: Only one copy of subunit "name" will be saved.

CAUSE Multiple copies of a subunit were saved under the same name. For example

> SAVE FILEX, SUB A, SUB B, SUB A (SUB A is entered twice)

> SAVE FILEX, 10/100, SUB B (SUB B is already saved in 10/100)

ACTION None. Only one copy of subunit will be saved.

1815 MESSAGE WARNING 1815: The file contains invalid (SECUREd) program lines.

CAUSE A program that has secured program lines in ASCII or in BASIC DATA format has been saved. These secured lines will cause syntax error during GET because only an "asterisk" is stored for each secured statement.

ACTION When program lines are secured, always save the program in BASIC SAVE format.

1816 MESSAGE WARNING 1816: Renumbering line ignored when GETting a BASIC SAVE file.

CAUSE A line was renumbered during a BASIC SAVE GET.

ACTION None. You are allowed to renumber program lines only when getting a BASIC DATA or an ASCII file.

1817 MESSAGE WARNING 1817: Unable to do a required purge of the temporary file.

CAUSE A temporary file was created and could not be purged following its use.

ACTION None. This is just a warning.

1818 MESSAGE WARNING 1818: NLS not installed, unable to open native message catalog.

CAUSE Native Language Support is not installed on your system.

ACTION You must log on as MANAGER.SYS and run the LANGINST program to add languages to the configuration file. Please refer to the *Native Language Programmer's Guide* for more information about installing native languages.

1819 MESSAGE WARNING 1819: Native language "num" is not configured

CAUSE The specified native language is not configured.

ACTION You must log on as MANAGER.SYS and run the LANGINST program to add the language number to the configuration file. Please refer to the *Native Language Programmer's Guide* for more information about installing native languages.

1820 MESSAGE WARNING 1820: Error message text is now inaccessible.

CAUSE The message catalog file, HPBBCAT.PUB.SYS, is inaccessible.

```
          ACTION       Exit Business BASIC and find out what is wrong with
                       HPBBCAT.PUB.SYS.

------------------------------------------------------------------------------------

1821     MESSAGE     WARNING 1821: Recommend that you exit Business BASIC and retry.

          CAUSE        The message catalog is inaccessible.  This message is an
                       additional message to warning 1820.

          ACTION       Exit Business BASIC and find out what is wrong with
                       HPBBCAT.PUB.SYS.

------------------------------------------------------------------------------------

1822      MESSAGE      WARNING 1822: Unable to open message catalog for native
                       language "num".

         CAUSE        The message catalog for the native language number num is not
                       available.

         ACTION       None.  The default message catalog file, HPBBCAT.PUB.SYS, for
                       language number 0 is used instead.

------------------------------------------------------------------------------------

1823     MESSAGE     It is recommended that you VERIFY and then RESAVE this program.

          CAUSE        This warning message is generated if you have an old file
                      version.  This does not mean the file cannot be read.  It is a
                       suggestion.

          ACTION       VERIFY and RESAVE the program.

------------------------------------------------------------------------------------

1830      MESSAGE      WARNING 1830:  Programs cannot be RUN with this amount of
                       subunit space.

         CAUSE        The amount of space available on the system is not sufficient
                       to run HP Business BASIC/XL.

          ACTION       Consult your system manager.

------------------------------------------------------------------------------------

1831      MESSAGE      WARNING 1831: RLINIT, RLFILE, and LOCALITY apply to MPE/XL
                       only.
         CAUSE       These compiler options apply to a native mode program on MPE/XL
                       only and will be ignored on other systems.

          ACTION       None.

------------------------------------------------------------------------------------

1832      MESSAGE      WARNING 1832: USLINIT applies only to MPE/V  systems.

         CAUSE       COPTION USLINIT applies to MPE/V system only and is ignored on
                       MPE/XL system.

          ACTION       None.

------------------------------------------------------------------------------------

2001      MESSAGE      VERIFY is needed on subunit "name".

          CAUSE        A program containing a poorly formed program unit has been
                      saved.  The interpreter issues a warning message and marks the
                       program unit as noncompilable.

         ACTION       Use the VERIFY command in the interpreter to find and correct
                       the problem.

------------------------------------------------------------------------------------

2004      MESSAGE      An expression is not allowed here.
```

CAUSE A parameter to the compiler options contains an expression.

ACTION Only use numbers or quoted strings for parameters in compiler options.

2005 MESSAGE ERROR, HALT, or KEY statement found while NO ERROR HANDLING option in effect.

CAUSE When COPTION NO ERROR HANDLING is used, ON ERROR, ON HALT, or ON KEY statements cause a compile time error.

ACTION Take out the COPTION or do not use the ON ERROR, ON KEY, or ON HALT statements.

2006 MESSAGE Parameter on "coption name" option is out of range.

CAUSE A numeric parameter to a compiler option is outside of it's legal range. For example, a parameter of the LINES option is outside the range of [0..9999].

ACTION Change the value to be the legal range for the compiler option.

2008 MESSAGE Error creating process: "error-num "

CAUSE The CREATEPROCESS intrinsic failed with error-num.

ACTION Please refer to the *MPE XL Intrinsics Reference Manual* for the error numbers returned in the CREATEPROCESS intrinsic, or consult your system manager.

2009 MESSAGE Error "error-num" in COMMAND intrinsic: "error-msg"

CAUSE The COMMAND intrinsic failed.

ACTION Please refer to the *MPE XL Intrinsics Reference Manual* for error numbers returned in the COMMAND intrinsic, or consult your system manager.

2010 MESSAGE Couldn't open input file.

CAUSE The input file specified for the compiler could not be opened. Probable causes are that the file does not exist or it is opened in a conflicting mode.

ACTION Check the compiler input file.

2011 MESSAGE Number of dimensions for array "name" not known in subunit "name"

CAUSE The number of dimensions for the array name cannot be determined at compile time.

ACTION Use the interpreter to explicitly dimension the array, specify the exact number of asterisks in an array parameter, or access a specific array element.

2012 MESSAGE Total space needed for variables is too big.

CAUSE The number or size of variables in the program exceeds the limit.

ACTION Reduce the number or size of variables in the program.

2013 MESSAGE Total space needed for parameters is too big.

CAUSE The number or size of parameters to the subunit exceeds the limit.

ACTION Reduce the number or size of parameters to the subunit.

2014 MESSAGE Total space needed for DATA is too big.

CAUSE The number or size of values in the DATA statements exceeds the limit.

ACTION Reduce the number or size of values in the DATA statements.

2017 MESSAGE Fatal compiler error; compile terminated.

CAUSE A fatal compiler error was encountered, and the compile terminated because of one of the following errors:

47 "Name" COMMON area does not exist.

47 Dimensions or type of COMMON variable in line "num" doesn't match main.

1143 Can't read from file.

1499 Catastrophic error in heap management (unable to return space).

2001 VERIFY is needed on subunit "name".

2012 Total space needed for variables is too big.

2013 Total space needed for parameters is too big.

2014 Total space needed for DATA is too big.

2011 Number of dimensions for array "name" not known in subunit "name".

ACTION Use the interpreter to correct the problem, and recompile.

2018 MESSAGE Can't open internal communication file. File system error "error-num".

CAUSE An HP Business BASIC/XL internal file could not be opened.

ACTION Consult your system manager or refer to the *MPE XL Intrinsics Reference Manual*.

2019 MESSAGE Expression too complicated.

CAUSE The expression in the statement is too complicated, it might cause stack overflow or code segment overflow.

ACTION The expression should be made simpler by putting parts of it into temporary variables, and then using the variables in the expression.

2020 MESSAGE Redimension of "array-name"() illegal because of NO REDIM compiler option.

CAUSE When COPTION NO REDIM is used to disallow redimensioning of arrays, any statements that attempt to change the dimension of arrays will cause a compile time error. For example:

```
10 COPTION NO REDIM

20 DIM A(1,2)
```

```
30 MAT READ A(1,1) !  attempt to redim.  A
```

ACTION Change the compiler option or change the statements so that no redimensioning is done.

2021 MESSAGE BASIC Compiler Backend Error: [in procedure proc_name] 'Actual backend error message.' Fatal compiler error; compile terminated.

CAUSE A problem has been detected by one of the code generating subsystems of the compiler. The error has occurred in either the optimizer or the code generator itself. The procedure name being compiled when the error occurred will be substituted for proc_name, if it is known. In order to clarify the nature of the error, the actual backend error message is printed as the second line in the error message. All of these errors will cause the compiler to abort. Serious errors will result in a stack trace as well. The stack trace is helpful as documentation for resolving the problem with your HP representative.

ACTION Some of the problems can be corrected by reading the text of the 'Actual backend error message' and rectifying the problem. Other problems are internal compiler code generation problems that should be reported to your HP representative.

Examples of 'Actual backend error message':

** MESSAGE Cannot open object file obj_file_name (5209)

CAUSE The object code file specified in the command to run the compiler cannot be opened because the system is out of disk space or because your disk space limit, as set by the system administrator, has been reached.

ACTION Make sure that a sufficient amount of disk space exists.

** MESSAGE Invalid file code for object file obj_file_name (5211)

CAUSE The object code file specified in the command to run the compiler does not have an NMOBJ or NMRL file code.

ACTION Check the file code for the file named obj_file_name or the file specified by you as the object code file. BBCOBJ is the file that the compiler uses after it has been equated to your file.

** MESSAGE File file_name has invalid file code; expected NMRL (5381)

CAUSE The object code file specified in the command to run the compiler does not have an NMRL file code.

ACTION Check the file code for the file named file_name or the file specified by you as the object code file. BCOBJ is the file that the compiler uses after it has been equated to your file. Either build an RL file using the linkeditor or do not use the RL compile options.

** MESSAGE File file_name has invalid record size. Expected 128W records (5383)

CAUSE The object code file specified in the command to run the compiler is an RL file with an NMRL file code that does not have 128 word records.

ACTION Check the record length for the file named file_name or the file specified by you as the object code file. BBCOBJ is the file that the compiler uses after it has been equated to your file. Build a new RL file using the linkeditor.

2050 MESSAGE WARNING 2050: TRACE or PAUSE statement found and ignored.

CAUSE The compiler did not generate any code for a TRACE or PAUSE statement.

ACTION None. These statements are used primarily for debugging.

2051 MESSAGE WARNING 2051: Multiple copy of subunit "sub-name" found and not compiled.

CAUSE Multiple copies of a subunit that have the same name were found in the program.

ACTION If a program has more than one subunit with the same name, only the one with the lowest line number is compiled. To compile a higher-numbered subunit, remove the lower-numbered one with the same name.

2053 MESSAGE WARNING 2053: Noncompilable statement; run-time error will result.

CAUSE Any statement that attempts to modify an HP Business BASIC/XL program at run time or requires the interpreter environment will result in a run-time error. The following statements generate this warning.

COMMAND

GET

MERGE

SCRATCH

DEFAULT

GETSUB

RESAVE

SECURE

DELETE

LINK

ACTION The following change will prevent the execution of a non-compilable statement in the compiled program:

100 GET "abc"is changed to 100 IF INTERPRETED THEN GET "abc"

2054 MESSAGE WARNING 2054: "array-name"() may be redimensioned despite NO

```
                    REDIM compiler option.
        CAUSE      A matrix operation that might cause an implicit redimensioning
                  of an array will generate this warning message at compile time.
                    For example,

                    10 COPTION NOREDIM

                    20 DIM A(4,2), B(2), C(5)

                    30 MAT C = MUL(A,B) ! C may be redimensioned

        ACTION     None, if you know exactly how the array will be redimensioned.
                   Otherwise, the results will be unpredictable when the array is
                    redimensioned.  The compiler will not generate code to check
                    the array bounds with COPTION NOREDIM.
```

```
2055       MESSAGE     WARNING 2055: Redim of "array-num"() possible; check REDIM
                       coption of actual parms

           CAUSE       The array that is passed in the actual parameter might be
                       redimensioned and COPTION NOREDIM is used in the caller
                       subroutine.

           ACTION      None if you know exactly how the array will be redimensioned.
                       Otherwise, the results will be unpredictable when the array is
                       redimensioned and you are not aware of the changes.  The
                       compiler will not generate code to check the array bounds with
                       COPTION NOREDIM.
```

```
2056       MESSAGE     WARNING 2056:  [on line line_num:  ]'Actual backend warning
                       message.'

                       or

                     WARNING 2056:  in procedure proc_name:  'Actual backend warning
                       message.'

                       or

                     WARNING 2056:  on line line_num in procedure proc_name: 'Actual
                       backend warning message.'

           CAUSE       This warning describes a non-fatal event that occurred during
                       program compilation.  The line_num and proc_name are printed,
                       if available.

           ACTION      None, other than to be aware that the event may have an effect
                       on results.

                       Example of 'Actual backend warning message':

                        ** MESSAGE               Previous version of entry proc_name
                                                  was replaced (5080)

                       CAUSE                    The object code for the entry listed
                                                has been replaced in the specified RL
                                                  file.
```

```
2100       MESSAGE     Too many GOSUBs before a RETURN.  Use MAXGOSUBS option to
                       increase maximum.

           CAUSE       Too many GOSUB statements were executed before a RETURN
                       statement was executed.

           ACTION      Use the MAXGOSUBS compiler option to increase the maximum
                       number of GOSUB statements allowed before a RETURN.
```

2101 MESSAGE An unknown arithmetic error occurred.

CAUSE This is caused by an internal problem.

ACTION Further investigation of this problem is required. Please contact your Hewlett-Packard representative.

2103 MESSAGE Attempt to execute a noncompilable statement.

CAUSE Any statement that attempts to modify an HP Business BASIC/XL program at run time or requires the interpreter environment will result in run-time error. The following statements generate this message.

COMMAND

GET

MERGE

SCRATCH

DEFAULT

GETSUB

RESAVE

SECURE

DELETE

LINK

ACTION The following change will prevent the execution of a non-compilable statement in the compiled program:

100 GET "abc" is changed to 100 IF INTERPRETED THEN GET "abc"

Syntax errors

The following error messages are the syntax errors. They are all error 68, although in some cases you will get these messages instead of the message for error 68 (Syntax error at character N). Those errors are marked as substitute errors.

MESSAGE One of the clauses is not allowed with this statement.

CAUSE One of the clauses following an HP Business BASIC/XL database keyword incorrectly occurs following that keyword.

ACTION Check the syntax of the database statement in the Help Catalog or the *HP Business BASIC/XL Reference Manual* to be certain that you are using the correct syntax.

MESSAGE One of the clauses occurred more than once.

CAUSE One of the clauses following an HP Business BASIC/XL database keyword has been repeated.

ACTION Check the syntax of the database statement in the Help Catalog or the *HP Business BASIC/XL Reference Manual* to be certain that you are using the correct syntax.

MESSAGE The statement is missing one or more clauses.

CAUSE This is a substitute message for error number 68. One or more of the clauses following an HP Business BASIC/XL database keyword is missing.

ACTION Check the syntax of the database statement in the Help Catalog or the *HP Business BASIC/XL Reference Manual* to be certain that you are using the correct syntax.

MESSAGE The line number is not between 1 and 999999.

CAUSE This is a substitute message for error number 68. The line number associated with GOTO, CONTINUE, BEGIN REPORT, GOSUB, or CONVERT, or the line number used in a command such as SAVE, FIND, or GET is not in the line range [1, 999999].

ACTION Use a line number in the range [1, 999999].

MESSAGE This statement is not allowed in a COMMAND statement.

CAUSE This is a substitute message for error number 68. Illegal syntax in a COMMAND statement. For example

```
10 COMMAND "if a then input b"
```

The error will only be generated with command strings that contain the following keywords: FLUSH INPUT, ENTER, INPUT, LENTER, ACCEPT, COMMAND, LINPUT, MAT INPUT, MAT READ, PAUSE, or TINPUT.

ACTION Modify the quoted string literal or the value of the string variable following the COMMAND keyword so that it does not include any of the above keywords.

MESSAGE The class of an active subunit may not be changed.

CAUSE This is a substitute message for error number 68. A program line that is a procedure header line, (SUB), has been replaced during editing with a program line that is a function header, (DEF), or vice versa.

ACTION Procedure and function header lines cannot replace each other. If you want to change a procedure to a function or vice versa, enter the new header line at the end of the current program and use the COPY command to copy the body. Next, do a DEL SUB of the original header and body.

MESSAGE The generic type (string/number) of an active function may not change.

CAUSE This is a substitute message for error number 68. The type of value returned by a function was changed when the program was paused or halted while in the function. A numeric type was changed to a string type or vice versa.

ACTION Allow the program execution to terminate and then make the required changed.

MESSAGE Number after ' in string is not between 0 and 255.

CAUSE The singe quote (') in a string literal is used to denote a character by its ASCII equivalent number. For 8-bit characters, these numbers can range from 0 to 255. A single quote was encountered followed by a number greater than 255.

ACTION Determine the correct number for the desired character.

MESSAGE A number between 0 and 255 must follow ' in the string.

CAUSE A single quote was encountered in a string literal and was not followed by a digit.

ACTION Determine the correct ASCII character number for the desired character and place it after the '.

MESSAGE Unknown character " " found (ASCII nnn).

CAUSE A character was encountered which was not in the set of legal HP Business BASIC/XL characters.

ACTION If the character was entered inadvertently, retype the line (using REDO is not recommended). If the illegal character is part of a string, use the singe quote (') notation to specify the character.

MESSAGE Parser stack overflow. Statement too complex.

CAUSE The statement entered was so complex that HP Business BASIC/XL was unable to process it without overflowing internal tables.

ACTION Break up the line into at least two less complex statements.

MESSAGE This statement is not allowed in this context.

CAUSE A statement has been entered in a place in the program where it cannot legally go. For example, a SUB statement was entered in the middle of another subunit or a GLOBAL statement was used in a subunit other than MAIN.

ACTION Add new SUBs and multi-line DEFs only at the end of the program. Use GLOBAL statements only in MAIN.

MESSAGE Could not create number from input line text.

CAUSE A statement was entered with an invalid line number.

ACTION Use an integer line number in the range [1, 999999].

MESSAGE OPTION BASE must use 0 or 1.

CAUSE An OPTION BASE statement was entered specifying a base other than zero or one.

ACTION If arrays are to have lower bounds other than zero or one, those bounds must be explicitly stated in the array declaration. Only zero or one is specified as the default lower bound with the OPTION statement or HPBBCNFG.Pub.Sys.

MESSAGE Whole array reference illegal in this context.

CAUSE A whole array reference (an array name followed by an asterisk with parentheses) has been used where only a scalar or simple data item is allowed.

ACTION Use the MAT statements to manipulate whole arrays.

MESSAGE Variably dimensioned arrays and strings illegal in MAIN program.

CAUSE A variable has been used to declare the array size or the string length in a string array declaration, and this is not allowed in MAIN.

ACTION Declare the array dimension or the string length explicitly in MAIN.

MESSAGE Improper string length declaration.

CAUSE A variable instead of a constant has been used to declare the length of a string declaration in MAIN.

ACTION Declare the sting length explicitly in MAIN.

MESSAGE Either DATASET or ITEMS clause must be given.

CAUSE A statement that requires a DATASET or ITEMS clause has been entered with neither a DATASET nor an ITEMS clause.

ACTION Provide the required clause.

MESSAGE You may not specify both the DATASET and the ITEMS clause.

CAUSE The DBINFO statement does not allow both the DATASET and ITEMS clauses.

ACTION Check the DBINFO mode to specify either the ITEMS clause or the DATASET clause, but not both.

MESSAGE You may not specify both the DATASET and the DESCRIPTOR clauses.

CAUSE The DBLOCK statement does not allow both the DATASET and the DESCRIPTOR clauses.

ACTION Check the LOCK mode. If the LOCK mode is 3 or 4, specify the DATASET clause. If the LOCK mode is 5 or 6, specify the DESCRIPTOR clause.

MESSAGE This statement is not allowed in an IF statement typed from the keyboard.

CAUSE A statement performing terminal input has been included as part of an IF statement entered without a line number.

ACTION None. Terminal input statements are not allowed from the keyboard. Place the statement in a program.

MESSAGE No more than 8 keys may be specified in this state.
CAUSE An ON KEY or OFF KEY statement has a list of more than eight key numbers.

ACTION Modify the statement to specify no more than eight keys.

MESSAGE Between 1 and 8 keys must be specified in this statement.

CAUSE An ON KEY statement did not specify any key numbers or specified more than eight key numbers.

ACTION Correct the statement to include at least one but not more than eight keys.

MESSAGE Only =, >=, and <= are allowed here.

CAUSE An invalid relational operator was specified in the PREDICATE statement.

ACTION Check the relational operator. Only =, >=, and <= are allowed in the PREDICATE statement.

MESSAGE Illegal line number: !

CAUSE GET (of an ASCII file), LINK, or MERGE has created a line reference that is greater than 999999.

ACTION Renumber using a lower line number.

MESSAGE Built-in function xxx has wrong type in parameter nn.

CAUSE In the parameter list of built-in function xxx, a string was found where a numeric value was required or a numeric value was found where a string was required.

ACTION Check the syntax for the built-in function and use the correct parameter types.

MESSAGE Wrong number of arguments for built-in function xxx.

CAUSE A built-in function xxx was used with the wrong number of arguments.

ACTION Check the syntax for the built-in function and use the correct number of arguments.

MESSAGE Operators not allowed in array built-in functions.

CAUSE Only an array variable is allowed in the MAT built-in function.
ACTION Put an array variable inside the MAT built-in function. Put a parentheses around the built-in function to assign a value to an array.

MESSAGE Built-in function "name" not allowed here.

CAUSE The built-in function used in the MAT assign statement is invalid.

ACTION Check the built-in function name.

MESSAGE Illegal character in data item or missing data items.

CAUSE Only the following are legal separators for the data items in the DATA statement:

,
;
!

ACTION Check the syntax of the DATA statement.

MESSAGE Empty arguments not allowed in built-in functions.

CAUSE Unexpected empty arguments, ", ,", are specified as parameters to a built-in function.

ACTION Specify a value between the commas.

MESSAGE New array bounds must be specified in REDIM statement.

CAUSE New array dimensions were not specified in a REDIM statement.

ACTION Specify the new array bounds.

MESSAGE Undefined variable or improperly used keyword.

CAUSE A line has been entered for immediate execution that has an unknown identifier. This may be an undefined variable or a keyword that has been misspelled or misplaced.

ACTION Do not attempt to use an undefined variable in the calculator.

MESSAGE BREAK ... WHEN ... BY requires a numeric control expression.
CAUSE A numeric control expression representing a "step" value for triggering breaks is required in the BY clause.

ACTION Change the syntax to use only numeric expressions in the BY clause.

MESSAGE The WRITE FORM statement requires at least one clause.

CAUSE You must specify at least one clause in the WRITE FORM statement.

ACTION Check the syntax of the statement in the Help Catalog or the *HP Business BASIC/XL Reference Manual* to be certain that you are using the correct syntax.

MESSAGE This CHANGE command cannot change the type of a variable.

CAUSE The CHANGE <vars> command allows variable names to be changed to a new name. However, you are not allowed to change the type of a variable from numeric to string or vice versa.

ACTION Do not attempt to change the type of a variable with the CHANGE <vars> command.

MESSAGE Line too long to process the CHANGE command.

CAUSE The CHANGE command exceeds 500 characters.

ACTION Shorten the CHANGE command to 500 characters or fewer.

MESSAGE This statement or phrase applies only on MPE V systems.

CAUSE You have used a feature that is only available on MPE V systems. For example, SPL language is not allowed in the EXTERNAL statement on MPE XL systems.

ACTION Check your program and use the correct syntax for MPE XL. Rewrite any SPL programs that will be called by HP Business BASIC/XL.

Appendix B Statement Groups

Table B-1 is a list of Business BASIC/XL statements, grouped by functionality. Each statement is defined and explained in chapter 4.

Table B-1. Functional List of HP Business BASIC/XL Statements.

Functionality	**Statement**
Array Operations	MAT = MAT INPUT MAT PRINT MAT READ REDIM
Control	COMMAND FOR NEXT GOSUB GOSUB OF GOTO GOTO OF IF THEN IF THEN ELSE LOOP ON GOSUB ON GOTO REPEAT UNTIL RETURN SELECT STOP WAIT WHILE DO
Database Management	BEGIN TRANSACTION DBASE IS DBCLOSE DBDELETE DBERROR DBEXPLAIN DBFIND DBGET DBINFO DBLOCK DBMEMO DBOPEN DBPUT DBUNLOCK DBUPDATE END TRANSACTION FILTER IN DATASET ON DBERROR OFF DBERROR

	PACK
	PACKFMT
	PREDICATE
	SEARCH
	SORT
	SORT ONLY
	THREAD IS
	UNPACK
	WORKFILE IS
Data Files	ADVANCE
	ASSIGN
	LINPUT
	LOCK
	POSITION
	PRINT
	READ
	UNLOCK
External Routines and Intrinsics	EXTERNAL
	INTRINSIC
	SETLEN
Forms	CLEAR FORM
	CLOSE FORM
	OPEN FORM
	READ FORM
	WRITE FORM
Input and Output	ACCEPT
	BEEP
	COPY ALL OUTPUT TO
	DISP
	DISP USING
	FIXED
	FLOAT
	IMAGE
	INPUT
	LDISP
	LINPUT
	MARGIN
	PRINT
	PRINT USING
	SEND OUTPUT TO
	SEND SYSTEM OUTPUT TO
	STANDARD
	TINPUT
Interrupt Handling	OFF ERROR
	OFF HALT
	OFF KEY
	ON ERROR
	ON HALT
	ON KEY
	WARNINGS OFF
	WARNINGS ON

JOINFORM	ACCEPT CLEAR FORM CLOSE FORM CURSOR DISP ENTER INPUT LDISP LENTER LINPUT OPEN FORM PRINT
Operating System Access	SYSTEM SYSTEMRUN
Report Writer	BEGIN REPORT BREAK IF BREAK WHEN DETAIL LINE END REPORT END REPORT DESCRIPTION GRAND TOTALS HEADER LEFT MARGIN PAGE HEADER PAGE LENGTH PAGE TRAILER PAUSE EVERY PRINT DETAIL IF REPORT EXIT REPORT HEADER REPORT TRAILER SET PAGENUM STOP REPORT SUPPRESS AT SUPPRESS FOR SUPPRESS HEADER SUPPRESS TRAILER TOTALS TRAILER TRIGGER BREAK TRIGGER PAGE BREAK
Screen Formatting	CURSOR ENTER LENTER
Subunits	CALL DEF FN FNEND RETURN SUB SUBEND SUBEXIT SUBPROGRAM

User-definable Keys	CURKEY DISABLE ENABLE GET KEY OFF KEY ON KEY PRESS KEY RESAVE KEY SAVE KEY SCRATCH KEY
Variable Operations	COM CONVERT DATA DEFAULT OFF DEFAULT ON DIM OPTION READ RESTORE

Appendix C HP Business BASIC/XL Configuration Utility

The configuration file (HPBBCNFG.PUB.SYS), supplied with HP Business BASIC/XL, is a convenient way of supplying defaults to HP Business BASIC/XL for the language features shown in Table C-1 at the end of this appendix. The HP Business BASIC/XL configuration utility (CNFGHPBB.PUB.SYS) is a program that allows you to create and change configuration files.

A configuration file is used to customize the HP Business BASIC/XL environment to the conventions of your installation. For example, suppose that most applications are financial and it is a convention that all variables be declared. In this case, you can use the configuration utility to create a configuration file with all of the original defaults, except with OPTION DECIMAL instead of OPTION REAL and OPTION DECLARE instead of OPTION NODECLARE. Programmers then could avoid having to include a line such as the following in programs that are run in this environment:

```
          10 GLOBAL OPTION DECIMAL,DECLARE
```

MPE XL file equations can be used to cause HP Business BASIC/XL to use a file other than HPBBCNFG.PUB.SYS for configuration information. This is useful, for example, when an individual user wishes to have a different HP Business BASIC/XL environment than the site standard. This alternate configuration file does not have to reside in PUB.SYS, it can be in any place you have access to. For example, to run a version of the HP Business BASIC/XL interpreter that has Swedish as the native language, you could create a user defined command in MPE XL that uses a configuration file in the local group, as shown below:

```
          SWEDHPBB
          FILE HPBBCNFG.PUB.SYS=SWEDCNFG
          BBASIC
          RESET HPBBCNFG.PUB.SYS
```

If you type SWEDHPBB, the configuration file is set to SWEDCNFG, the HP Business BASIC/XL interpreter is invoked, and when you exit from the interpreter, the configuration file is reset to the system default configuration file. The file, SWEDCNFG, is a configuration file that has the native language parameter of 13.

The standard HP Business BASIC/XL defaults take effect when one of the following occurs:

 * HP Business BASIC/XL runs without a configuration file.
 * HP Business BASIC/XL runs with a configuration file that was created by running the configuration utility and accepting the original defaults that it supplied.
 * HP Business BASIC/XL runs with a configuration file, but is unable to access it.

Although OPTION statements in your program will override the defaults in the configuration file, the defaults contained in the configuration file take effect when one of the following occurs:

 * The HP Business BASIC/XL interpreter runs.

 * A SCRATCH ALL or a SCRATCH PROG command executes (this only applies in the interpreter).

 * A compiled HP Business BASIC/XL program runs.

How to Run the Configuration Utility

To run the configuration utility, issue the MPE XL command:

```
          :RUN CNFGHPBB.PUB.SYS
```

The configuration utility looks for the file, HPBBCNFG.PUB.SYS. To create or change a configuration file that has another name, you must set up a file equation before running the configuration utility. For example:

```
          FILE HPBBCNFG.PUB.SYS = HPBBCNFG.mygroup.myacct
```

If the file is not found, it is created (assuming that you have the required capabilities). It is then filled in with the original defaults and you are given the chance to override them. If the file already exists, you can change the contents.

Appendix D ASCII Character Codes

Table D-1 maps each ASCII character to its decimal and hexadecimal code, its symbol, and its name. Each code is stored in eight bits; so the decimal codes are in the range [0, 255] and the hexadecimal codes are in the range [0, FF].

Table D-1. ASCII Character Codes

Decimal Code	Hexadecimal Code	Symbol	Name
0	00	NUL	Null
1	01	SOH	Start of heading
2	02	STX	Start of text
3	03	EXT	End of text
4	04	EOT	End of transmission
5	05	ENQ	Inquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal tab
10	0A	LF	Line feed
11	0B	VT	Vertical tab
12	0C	FF	Form feed

Table D-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	Symbol	Name
13	0D	CR	Carriage return
14	0E	SO	Shift out
15	0F	SI	Shift in
16	10	DLE	Data link escape
17	11	DC1	Device control 1
18	12	DC2	Device control 2
19	13	DC3	Device control 3
20	14	DC4	Device control 4
21	15	NAK	Negative acknowledgement
22	16	SYN	Synchronous idle
23	17	ETB	End of transmission block
24	18	CAN	Cancel
25	19	EM	End of medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File separator
29	1D	GS	Group separator

Table D-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	Symbol	Name
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	SP	Space
33	21	!	Exclamation mark
34	22	"	Quotation mark
35	23	#	Number sign
36	24	$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand
39	27	'	Apostrophe
40	28	(	Left parenthesis
41	29	)	Right parenthesis
42	2A	*	Asterisk
43	2B	+	Plus sign
44	2C	,	Comma
45	2D	-	Minus sign

Table D-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	Symbol	Name
46	2E	.	Full stop
47	2F	/	Solidus
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less-than sign
61	3D	=	Equal sign
62	3E	>	Greater-than sign

Table D-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	Symbol	Name
63	3F	?	Question mark
64	40	@	Commercial "at" sign
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O

Table D-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	Symbol	Name
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Left bracket
92	5C	\	Reverse solidus
93	5D	]	Right bracket
94	5E	^	Circumflex accent
95	5F	_	Underline

Table D-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	Symbol	Name
96	60	‘	Grave accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p

Table D-1. ASCII Character Codes (continued)

Decimal Code	Hexadecimal Code	Symbol	Name
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Left brace
124	7C	|	Vertical line
125	7D	}	Right brace
126	7E	~	Tilde
127	7F		Delete

Appendix E HP Terminals and Language Features

This appendix contains information about HP terminals that are fully and partially compatible with HP Business BASIC/XL's terminal-specific language features. Redirecting output can make a terminal appear to HP Business BASIC/XL as a batch job.

Fully Compatible Terminals

The following are terminals compatible with all of BASIC's terminal-specific language features:

150
2382
2392
2393A
2394
2397
2622
2623
2624
2626
2627

Valid Terminal-Specific Statements for Fully Compatible Terminals:

The following statements perform correctly when used on fully compatible terminals. (All forms features refer to VPLUS.)

OPEN FORM
CLOSE FORM
CLEAR FORM
WRITE FORM
READ FORM
CURSOR
RPOS
CPOS
GET KEY
SAVE KEY
RESAVE KEY
SCRATCH KEY
ENABLE
DISABLE
PRESS KEY
CURKEY
ON KEY
OFF KEY
ENTER
LENTER

Partially Compatible Terminals

The following terminals are compatible with a subset of BASIC's terminal-specific features. If the configuration file says that the terminal is supported, but the terminal is not an HP terminal, the terminal is treated as a 2640.

125
2640
2641
2642
2644
2645
2647
2648

Valid Subset of Terminal-Specific Statements for Partially Compatible Terminals:

The following statements perform correctly when used on partially compatible terminals. The statement "Labels are ignored" means that the labels of the terminal's user-definable keys are not updated.

CURSOR
RPOS
CPOS
ENABLE
DISABLE
PRESS KEY
CURKEY
ON KEY (labels are ignored)
OFF KEY (labels are ignored)
ENTER
LENTER

Minimal Subset of Terminal-Specific Statements:

Other remaining terminals and batch jobs are less compatible with BASIC's terminal-specific statements. The valid subset of statements for these terminals is shown below:

ENABLE
DISABLE
PRESS KEY
CURKEY
ON KEY (labels are ignored)
OFF KEY (labels are ignored)

The following terminals are compatible with all of the JOINFORM statements listed in Appendix F:

150
2382
2392
2393A
2394
2397

Appendix F JOINFORM

JOINFORM Statements

JOINFORM is a FORMS/260 compatible forms package available in HP Business BASIC/XL. The JOINFORM package cannot be accessed by any other languages on the HP 3000. Use of JOINFORM is supported only on the HP150 and HP239X Terminals. It is intended to provide an easy-to-use alternative to VPLUS forms for HP260 users converting their applications.

OPEN FORM

OPEN FORM opens a form file. It tries to find *form_member_name* in *form_file_name* if a form file is specified. Otherwise it searches the currently open, default form file. If the specified form exists, it is displayed at the current cursor position. Form names are limited to eight characters.

If a form is already active when OPEN FORM is executed, it is deactivated and the new form is inserted at the cursor position.

The Keywords HOME, OVERLAY, APPEND, and FREEZE have no effect when a JOINFORM is opened.

Syntax

```
                        [HOME    ]
                        [OVERLAY]
OPEN FORM form_name  [;] [FREEZE ]
                        [APPEND ]
```

Parameters

form_name *Form_name* is a string expression with the following format: *form_member_name* [:*form_file_name*]

Form_member_name is the name of the form you are opening. *Form_file_name* is a quoted string literal that is the name of the file that contains the form.

HOME
OVERLAY
FREEZE
APPEND

The HOME, OVERLAY, FREEZE, and APPEND options are ignored if the form to be opened is a JOINFORM.

Examples

```
     130 OPEN FORM "Appl1"
     140 OPEN FORM Form2
     150 OPEN FORM Form$
     160 OPEN FORM "form1:joinfile"
```

CLEAR FORM

CLEAR FORM clears all input and output field entries on the form. The form is not drawn on the screen. The input, output, and cursor field pointers are reset to the first input and first output field. The cursor is placed in the first input field. If the form does not have input fields, the cursor is placed in the left upper corner.

The optional keyword [[WITH] DEFAULT[S]] has no effect for converted JOINFORM. It is ignored.

If there is no active form, CLEAR FORM returns an error.

Syntax

```
CLEAR FORM [[WITH] DEFAULT[S]]
```

Examples

```
     150 CLEAR FORM    !Clears all fields
```

CLOSE FORM

CLOSE FORM deactivates and erases the form that is currently active. If no option is specified, the form is erased by deleting all lines occupied by the form, so the lines following the form are moved up on the screen. Use the CLEARREST option to clear the form by clearing display memory from the first line of the form to the end of display memory. Use the CLEARALL option to clear the form by clearing all of display memory. Use the REMAIN option to deactivate a form without erasing it.

If the cursor is in the form when CLOSE FORM is called, it is positioned to the line that followed the form. If the cursor is outside of the form, it is positioned to the same line again after the form is deleted.

If no form is active, CLOSE FORM returns immediately without performing any action.

Syntax

```
           [{;}          ]
           [{,} CLEARREST]
CLOSE FORM [CLEARALL     ]
           [REMAIN       ]
```

Examples

```
     90 CLOSE FORM
     100 CLOSE FORM ;CLEARREST
     110 CLOSE FORM ;CLEARALL
     120 CLOSE FORM ;REMAIN
```

CURSOR

The CURSOR statement positions the terminal cursor within an active JOINFORM. When positioning the cursor while a JOINFORM is active two parameters must be supplied. The first parameter is either CFLD, IFLD, OFLD, SETCFLD, SETIFLD, or SETOFLD. This parameter specifies the type of field that the cursor is being moved into. The second argument is the number of the field of that type on the form. 'CURSOR OFLD (5)' means "position the cursor to the fifth output field within the defined output order of the active form". The SETCFLD, SETIFLD, and SETOFLD parameters set the internal field pointer, as do the CFLD, IFLD, and OFLD parameters, but they do not move the cursor. A subsequent INPUT, DISP, or PRINT statement will move the cursor to the desired field before the input or output operation takes place. There is a performance improvement because the cursor is not moved. CFLD stands for cursor field and IFLD stands for input field.

The IFLD, OFLD, CFLD, SETCFLD, SETIFLD, and SETOFLD options of the CURSOR statement cannot be executed unless a JOINFORM is active.

Syntax

```
       {IFLD   }
       {OFLD   }
       {CFLD   }
CURSOR {SETIFLD} (field_number )
       {SETOFLD}
       {SETCFLD}
```

Parameters

IFLD, OFLD, CFLD	A keyword that specifies the type of field the cursor moves into.
SETIFLD,	A keyword that sets the internal field pointer for the

SETOFLD, type of field indicated.
SETCFLD

field_number The number of the field that the cursor will move to.

Examples

```
     100 CURSOR OFLD (35)     !Moves cursor to output field 35.
     110 CURSOR SETIFLD (4)   !Sets the input field pointer to field 4.
```

TFLD

TFLD is a built-in numeric function that returns the field number of the last input field accessed in the form. The cursor pointer is moved either by a CURSOR IFLD(), CFLD(), or an INPUT statement.

NOTE The actual cursor position and fieldnum returned to TFLD are only identical when the fields were walked through using the RETURN key. The TAB key moves the cursor to the next field (or the previous field when BACKTAB is pressed) in screen order. This is not recognized by TFLD since TAB and BACKTAB are local to the terminal. TFLD also does not recognize moving the cursor using the cursor positioning keys.

TFLD returns zero if executed when no JOINFORM is active.

Syntax

TFLD

PRINT and DISP

PRINT and DISP are standard HP Business BASIC/XL statements. Their syntax is exactly the same for normal output and output to JOINFORM. However, if a form is active, HP Business BASIC/XL calls a special forms output routine that behaves like a PRINT or DISP statement on the HP260 does. If a ", " is used to separate the items, each item is displayed in a separate field. If a "; " is used to separate them, then the output is buffered and displayed when a ", " is found or the statement is completed. The first field that an item is to be displayed in is defined by the output field pointer. The output field pointer can be positioned with the CURSOR OFLD statement. After an item is displayed in a field, the output field pointer is incremented.

The syntax for the PRINT and DISP statements are in chapter 4.

LDISP

The result of an LDISP statement depends on whether a form is active.

When no JOINFORM is active, the current line is cleared from the current cursor position to the end of the line. Output of the values of the *output_item* begins at the current cursor position on the screen. If the output requires more than the number of characters remaining on the cleared line, additional lines on the screen are used. However, the additional lines are not cleared before character output begins.

If a JOINFORM is active, the form is then inactivated. The cursor is repositioned to the first column of the first line following the form. Output then proceeds as if no JOINFORM were active. Following output, the cursor does not return to its previous position in the now inactive form. If the cursor is already outside the form, LDISP behaves as if no JOINFORM were active.

Syntax

LDISP [*d_list*]

Parameters

```
                                      [{,...}              ]
d_list            [,]...output_item_list  [{;   } output_item ]...
```

output_item   One of the following:

num_expr

str_expr

array_name(*)   Array reference. See "Array References in the Output Item List" in chapter 6 for more information.

```
                                  {PAGE             }
                                  {{CTL}            }
                 output_function   {{LIN}            }
                                  {{SPA} (num_expr )}
                                  {{TAB}            }
```

See "Output Functions in the Display List" in chapter 6 for more information.

FOR_clause   (FOR *num_var =num_expr1* TO *num_expr2* [STEP *num_expr3*], *d_list*)

See "FOR Clause in Output Item List" in chapter 6 for more information.

Examples

Assume that the following program statements are executed while a form is active:

```
     10 V$="Hi there."
     20 DISP V$                   !Prints in form field
     30 LDISP V$                  !Prints outside form
```

INPUT

When an INPUT statement is executed while a JOINFORM is active, the cursor is placed in the current cursor field. You can input data until RETURN is pressed. If no input elements are specified, only the cursor field pointer is increased. Otherwise, the entered data is assigned to the variables in the input item list. Following the assignment, the cursor field pointer and the input field pointer are increased.

If the cursor field pointer already points to the last input field in the form, it is reset to the first input field of the form. In contrast, the input field pointer is not circularly reset to the first input field but left undefined. Any further assignments from fields to variables result in errors.

The cursor can be explicitly positioned within the currently active form by using a previously executed CURSOR CFLD, CURSOR SETCFLD, CURSOR IFLD, or CURSOR Setifld statement.

When an INPUT statement is executed and a JOINFORM is not active, INPUT behaves normally.

Prompts in the INPUT and LINPUT statements are not printed when a JOINFORM is active.

The syntax for the INPUT statement is in chapter 4.

LINPUT

When LINPUT is executed and a JOINFORM is active, the current cursor position in screen memory is determined. If the cursor is within the form, LINPUT moves it to the first unprotected line following the form. Otherwise, the cursor stays where it is (usually positioned by a previously executed CURSOR statement). Then LINPUT outputs a line-output prompt. When RETURN is pressed, only what has been typed in is assigned to the string variable in the LINPUT statement. Input and output field pointers are not affected.

When the LINPUT statement is executed and a JOINFORM is not active, LINPUT behaves normally. The syntax for the LINPUT statement is in chapter 4.

ENTER

When ENTER is executed and a JOINFORM is active, the content of the current input field is assigned to the first element in the variable list. If there are more input fields on the form, the input field pointer is incremented to point to the next JOINFORM field. If an additional element is present in the ENTER statement's variable list, the value of the field is assigned to that variable. The input is read directly from the JOINFORM field. Input from the user is not accepted. Assignment to the variables in the variable list continues until values have been assigned to each. If no more JOINFORM input fields are present on the form, but one or more variables exist on the ENTER statement's variable list, an error occurs.

The cursor can be positioned within the currently active form by using a previously executed CURSOR IFLD statement.

When the ENTER statement is executed and a JOINFORM is not active, ENTER behaves as described in chapter 4.

The syntax for the ENTER statement is in chapter 4.

LENTER

When LENTER is executed and a JOINFORM is active, the current cursor position in screen memory is determined. If the cursor is within the form, an error occurs immediately. Otherwise, the current line is input at once without waiting for a keystroke. The cursor can be positioned out of the currently active form by a previously executed CURSOR statement. Input and output field pointers are not affected.

When the LENTER statement is executed and a JOINFORM is not active, LENTER behaves as described in chapter 4.

The syntax for LENTER is in chapter 4.

ACCEPT

Input without an echo on the terminal is possible at any time, even if a JOINFORM is active. The ACCEPT statement has no specific interaction with JOINFORM. The ACCEPT statement is explained in chapter 4.

BB_BLOCK_READ

The routine BB_BLOCK_READ is an HP Business BASIC/XL run-time library routine that has been provided to improve application performance. JOINFORM requires a significant amount of terminal I/O, slowing down performance. The BB_BLOCK_READ routine does a full-screen block-mode read of the currently active JOINFORM, improving performance.

BB_BLOCK_READ resides in the HP Business BASIC/XL library segment in

XL.PUB.SYS.

Before using BB_BLOCK_READ, be aware of the following considerations:

* BB_BLOCK_READ can lead to hard-coding dependencies on form layout (such as field length and order).
* BB_BLOCK_READ reads all unprotected fields on the screen each time it is called; input and input/output JOINFORM fields are read.
* The characters from the fields are read into a single string. The string must be large enough to hold all data from all fields plus one byte per field as a record separator.
* The application must explicitly extract the fields and convert them into usable data from the string.

Syntax

To call BB_BLOCK_READ from HP Business BASIC/XL, declare it as an external routine and use the CALL statement to call it. The external declaration has the following syntax:

EXTERNAL PASCAL BB_block_read (*buffer* $, SHORT INTEGER *status*)

Parameters

buffer $ String buffer that will contain all the input field data. This string contains all the characters from all the fields. This string must be long enough for that data, plus one character per field as a record separator.

status A short integer that contains the status of the external call. After the call, status can has a code that indicates the result of the call. The codes and their meanings are:

0	Successful call.
284	String too short.
287	No input fields.
294	No JOINFORM active.

BB_BLOCK_READ reads the fields in *hardware order*, that is from left to right and top to bottom. Changing input order or tab order cannot alter how the fields are returned to BB_BLOCK_READ. Therefore, small changes to a form, such as shortening or moving a field, can impact applications using BB_BLOCK_READ.

Example

```
10  EXTERNAL PASCAL BB_block_read(Buf$, SHORT INTEGER Stat) !Declare BB_BLOCK_READ
 20  SHORT INTEGER Stat
 .

 .

 .
 150  CALL BB_block_read(Buf$,Stat)  !Call BB_BLOCK_READ
 160                                 !Buf$ contains the data, stat the status
 170  IF stat <> 0 Then GOSUB 800   !Goto an error subroutine if the
 180                                 !call was not successful
 .
 .

 .
```

The JOINFORM Editor

The JOINFORM Editor is a utility program used to work with HP 260 forms on an MPE XL computer. The JOINFORM Editor includes the following capabilities:

- Creating HP 260-type forms.

- Modifying HP 260-type forms.

- Copying and moving forms between files.

- Deleting HP 260-type forms.

- Displaying forms on a workstation screen.

- Printing forms.

To run the JOINFORM Editor, type

```
     RUN JOINEDIT.PUB.SYS
```

at the MPE XL prompt. The Main Menu screen will be displayed.

Each of the capabilities of the JOINFORM Editor is described in the following sections. The first section, "Creating New Forms" describes the procedure for several JOINFORM Editor capabilities. Most of these actions are the same for other capabilities, such as modifying existing forms. The procedure is described in detail for new forms. Later sections that use these procedures refer to the "Creating New Forms" section.

Creating New Forms

New forms can be created from a blank screen or from an existing form.

Creating New Forms From a Blank Screen. There are four operations involved in creating a form:

1. Creating the input, output, and input/output fields and defining their sizes, locations, and types.

2. Creating the form frame using the line drawing character set of the specific terminal you are using. (The terminal or workstation operating manual contains information about which line drawing characters correspond to the keys on the terminal or workstation keyboard.)

3. Defining the text you want to be displayed on the form.

4. Setting field order and individual field enhancements.

To start form creation, enter the JOINFORM Editor and press the CREATE FORM softkey. A menu containing a selection for each of the four operations is displayed. The operations can be done in any order. Each is described here.

Creating Fields. Use the cursor keys to move the cursor to the position on the screen where you want to create a field. Press the softkey that indicates a field type (Input, Output, Input/Output) to create the field. Each time you press the softkey, the field is extended by one character. The field is highlighted in inverse video with the currently defined default filler character for that field type.

Creating the Form Image. Use the line drawing character set to draw the frame of the form. The frame is optional, and can be anything that your line drawing set allows.

Defining Text. Use the cursor keys and the alphanumeric keys to input the text that will appear on the form each time it is displayed. The text can be anywhere on the screen, except inside fields.

Setting Field Order and Individual Field Enhancements. After creating the fields, the frame, and the text, press the ENTER FORM softkey to display the softkeys that control field order and enhancements. Your screen will be displayed. The softkeys at the bottom control enhancements and order. You can make changes to the form layout (frame, fields and text) by pressing the CHANGE LAYOUT softkey.

The cursor will be positioned at the bottom of the screen in the only active line on the screen. The input order, output order and current enhancement of the first field will be displayed on the bottom line of the screen. That first field blinks and displays the currently defined fill character for its type.

The following procedures change the order in which the fields are accessed:

Field Type	**Procedure**
Input	Type the input order number the field will have next to the "INPUT NO:" prompt on the bottom line.
Output	Type the output order number the field will have next to the "OUTPUT NO:" prompt on the bottom line.
Input/Output	Use both of the above procedures for input and output fields.

Table F-1 lists the available enhancements, and the character string that represents them.

Table F-1. Enhancements

Character String	**Enhancement**
H	Field appears half as bright as normal text.
I	Field appears in inverse video. (Dark characters on a light background).
U	Field appears underlined.
B	Field blinks.
None	Field appears like ordinary text.

To use any of these enhancements, enter the character string for that enhancement in the inverse video field next to the "ENHANCEMENT:" prompt on the bottom line.

Any combination of the character strings can be entered in any order in the "ENHANCEMENT:" field to give the current field that combination of enhancements. Press the SAVE VALUES softkey to store the new values. Use the NEXT FIELD and PREVIOUS FIELD softkeys to select different fields for enhancements and ordering.

The DEFAULT ENHANCEM and DEFAULT IO ORDER softkeys reset the display enhancements and access order respectively. Pressing either of these keys destroys any changes you have made to the enhancements or access order.

Storing the Form to Disk. When you are satisfied with the appearance of the form, press the EXIT softkey. The screen that is displayed prompts you for the name for the new form. After naming it, press the SAVE FORM softkey to store the form to disk. If you are not satisfied with the form, you can press the BACK TO EDITING softkey to return to the previous screen to further edit the form.

Creating a New Form From an Existing Form

To create a form from an existing form, enter the JOINFORM Editor, and fill in the name of the existing form in the inverse video field next to f1 CREATE A NEW FOR, COPY FORM FROM. In the field below that (the FILE IS field), type the name of the file that contains the existing form. Press the CREATE FORM softkey. The existing form will be displayed. You can then make changes to the enhancements and access order, using the same procedure explained for new forms in the previous section. If you want to change the frame, fields, or text, press the CHANGE LAYOUT softkey.

Changing Fields. You can add or delete fields, or alter the field length on after pressing the CHANGE LAYOUT softkey. Add a field using the same procedure used to create a field in a new form. (Refer to "Creating Fields" in this appendix for details). To delete a field, press the DELETE key on your keyboard until all the characters in that field have been deleted. To alter the length of a field, position the cursor one character beyond that field. Use the cursor keys to move the cursor. Press the softkeys that control field type to add characters to the field, or press the DELETE key to reduce the length of the field.

Changing the Form Image and Text. Modify the form frame by adding and deleting line drawing characters.

Modify text by moving the cursor to the text you want to modify. Input, delete or alter the text. Use the cursor keys to move the cursor.

Storing the New Form to Disk. After you have finished changing the form, press the ENTER FORM softkey to return to the previous screen. Once you are satisfied with the new form, press the EXIT softkey to store your form to disk. Follow the procedure described for new forms in the previous section to save your form.

If, while saving the form, you decide you don't want to store it, press the EXIT softkey. The JOINFORM Editor asks if you really want to return to the main menu without storing the form. Press the EXIT softkey again to delete the form and return to the main menu. Press the SAVE FORM softkey if the EXIT softkey was pressed by mistake. If you return to the main menu without storing the form, you cannot retrieve it.

Modifying Forms

You can modify existing forms with the JOINFORM Editor. You can make the following modifications:

* Create fields.
* Delete fields.
* Alter field lengths.
* Change the frame.
* Change text.
* Set field input and output order.
* Alter individual display enhancement.

To modify a form, enter the JOINFORM Editor. Type the name of the form in the field directly after f2 MODIFY FORM on the screen. Type the name of the file that contains the form in the field directly after FILE IS. Press the MODIFY FORM softkey to display your form. On that screen, you can change the field access order and field enhancements. Refer to

"Creating New Forms" for details.

To modify the form frame, length or number of fields, or text in the form, press the CHANGE LAYOUT softkey, and refer to "Creating New Forms from Existing Forms" in the previous section.

When you are adding, deleting, or moving fields while modifying forms, the JOINFORM Editor does not reorder them. The effect of these changes is as follows:

* The previous field order remains valid for each field that has not been moved from its original location.

* Order number of deleted fields become vacant (Other fields are not given that number).

* New fields are given the lowest unoccupied numbers. Numbers are occupied even if you've deleted the field assigned to that number.

* Fields that are moved to new locations are treated like new fields.

After modifying the form, press the ENTER FORM softkey. This allows you to store the form. You can then press the EXIT softkey, and follow the procedure explained in "Creating New Forms" to save the form.

Merging Forms

The JOINFORM Editor includes a form merging facility that allows you to copy forms from one file to another without changing the form in the original file and to move forms from one file to another.

Enter the JOINFORM Editor and press the MERGE FORMS softkey. This displays the Merge Facility screen. You can move the cursor while in the Merge Facility by using the TAB key and the SHIFT TAB key combination. You can exit the Merge Facility by pressing the EXIT softkey.

Use the following procedure to merge forms:

1. Type the name of the source file (the file that contains the forms that will be copied or moved) in the field labeled FILE IS: on the FROM side of the screen. You can specify multiple FROM files by using MPE wildcard characters (@,?, and #).

2. Type the name of the destination file (the file that the form will be moved or copied into) in the field labeled FILE IS: on the TO side of the screen.

3. Type the name of the first form to copied or moved in the field labeled 1: on the FROM side of the screen.

4. Specify whether the form is to be copied or moved by entering M (for move) in the field labeled C/M. You do not need to enter anything to copy the form, C (for copy) is the default.

5. Specify the name you want the copied or moved form to have by typing the new name in the field labeled 1: on the TO side of the screen. This field can be left blank if you are using the same name in the TO file. If the new form name is the same as an existing form, the Merge Facility will ask if you want to overwrite the existing form. Press the f1 key to overwrite the form, or f8 to cancel the current merge. If you cancel the merge, forms that have not yet been processed will be displayed, and you can change the names of any that will overwrite existing forms.

6. Up to six forms can be moved or copied using the above method.

7. Press the START MERGE softkey when you have finished specifying the forms to merge.

Once the merge has begin, the name of each form is removed from the display after it has been successfully merged.

Deleting Forms

Forms can be deleted one at a time or in groups, from a list. To start deleting forms, enter the JOINFORM Editor.

Use the following steps to delete forms:

1. To delete a single form, use the TAB key to position the cursor at the field just past f5 DELETE FORMS, FORM IS.

2. Type the name of the form to be deleted.

3. Move the cursor to the field labeled FILE IS directly below the FORM IS field. Type the name of the file that contains the form to be deleted.

4. Press the DELETE FORMS softkey.

5. To delete forms from a list, specify the file name, but not the form name. Press the DELETE FORMS softkey. The DELETE FORMS screen is then displayed. The screen has spaces for up to 24 forms to be deleted.

6. Type the name of each form you want to delete in the multicharacter fields, and type an x in the single character field next to the field for the name. If you need a list of the forms in the file, press the DISPLAY FORMS softkey. Type an x next to each form you want to delete.

7. After you have indicated all the forms that you want to delete, press the DELETE FORMS softkey. This deletes the forms. If you have more than 24 forms, you can use the NEXT FORMS and PREVIOUS FORMS softkeys to see all the forms. Select which forms to delete from these additional screens.

8. When you are finished deleting forms, press the EXIT softkey to leave the DELETE FORMS screen.

Printing and Showing Forms

The JOINFORM Editor has a printing and showing facility that prints either forms or a list of forms on your screen or on your printer.

To use the printing and showing facility, press the SHOW FORMS softkey from the main menu. Type the name of the form in the field labeled FORM NAME: and the name of the file containing that form in the field labeled FILE NAME:. Press the SHOW FORM softkey to display the form on the screen. Use the NEXT FORM NAME and PREVIOUS FORM NAME softkeys to display other forms in the file. You can also type the name of other forms over the current name in the FORM NAME: field.

You can print a form by pressing the PRINT FORM softkey instead of the DISPLAY FORM softkey.

The default option of the PRINT FORM facility includes a table that contains the following information about each field.

- Field number.

- Enhancement.

- Length.

- Input order.

- Output order.

- Row of screen in which field appears.

- Position of field in that row.

You can display a list of the forms in a particular file. Type the name of the file in the FILE NAME: field, but leave the FORM NAME: field blank. Press the SHOW DIRECTORY softkey. 90 forms are displayed at one time. Use the PREV PAGE and NEXT page softkeys to see more forms.

To print a list of forms to your printer, press the PRINT DIRECTORY softkey instead of the SHOW DIRECTORY softkey. The JOINFORM Editor uses the LP printer as the default printer on the HP 3000. Use a file equation to specify another device as the printer. Equate that device to the formal file designator JFOUT.

Selecting Default Enhancements and Fillers

You can change the default enhancement and fill characters for the fields of your form. From the JOINFORM Editor press the CHANGE DEFAULTS softkey.

Field enhancement is used to show clear differences between each of the three field types. The original default enhancements for each type is shown in the table below.

Table F-2. Field Enhancement Defaults

Field Type	Default Value
Input	Half-bright, inverse video.
Output	Underlined.
Input/Output	Half-bright, inverse video, underlined.

Fill characters for each field indicates the field while you are creating it. The default fill characters are shown in the table below.

Table F-3. Fill Character Defaults (by field type)

Field Type	Default Value
Input	I
Output	O
Input/Output	C

To change these values, use the TAB key to move the cursor to the parameter you are changing. To change fill characters, type the alphanumeric character that will become the fill character in the to field for that parameter. To change the enhancements, type in any of the enhancement symbols at the bottom of the screen in the to field for that parameter. Press the SAVE NEW VALUES softkey when you have completed changing the default parameters. If you press the EXIT softkey before saving the new values, those changed values will be lost.

Appendix G   ANYPARM External Call Feature

Introduction

The HP Business BASIC/XL ANYPARM external feature is used with programs that were originally written in BASIC/V. Although calls to externals are easy to code in an HP Business BASIC/XL program, understanding and writing the externals that use the ANYPARM interface are more difficult than for normal externals. Therefore, use the ANYPARM external only when the HP Business BASIC/XL normal external call interface is too restrictive. In fact, you should rarely have to use this feature. This appendix is a technical discussion of the ANYPARM feature. It explains how it works and how it can be used.

This appendix contains the following information:

Table G-1. Information in ANYPARM External Call Feature

Section	Information
Overview of Calling Externals	This section contains a brief general introduction to the process of calling externals.
An Overview of ANYPARM	This section contains an overview of the ANYPARM external. It explains general considerations when using the ANYPARM external.
ANYPARM Calls From HP Business BASIC/XL	This section explains the call syntax for two methods of calling ANYPARM externals.
Writing ANYPARM External Procedures	This section contains the requirements for writing the external procedure, as well as the requirements for HP Business BASIC/XL's data structures.
Example of a Simple Pascal ANYPARM Procedure	This section is a simple example external that has an ANYPARM formal parameter interface written in Pascal. The HP Business BASIC/XL program that calls the external is also included.
Example of a Simple C ANYPARM Procedure	This section is a simple example external that has an ANYPARM formal parameter interface written in C. The HP Business BASIC/XL program that calls that procedure is also included.
Pascal Data Structures for ANYPARM Calls	This section is a program that contains Pascal data structures required for writing ANYPARM externals. This provides all the constant declarations and type definitions that allow you to manipulate any of the actual parameters passed from an HP Business BASIC/XL program.

A Pascal ANYPARM Procedure Designed to Process Any Parameter	This section is an example program and a memory display of a call from HP Business BASIC/XL to an external procedure. The example demonstrates how to call an ANYPARM external that is capable of processing any of HP Business BASIC/XL's data types.
Differences Relative to BASIC/V	This section explains the differences between HP Business BASIC/XL and BASIC/V that are relevant to the ANYPARM External.

Overview of Calling Externals

In general, programming languages provide an automatic interface for calling externals. That interface provides a correspondence between each parameter in the call to the external routine and each parameter in the called routine. The parameters are passed in a ordered list, and the programming language does most of the work required for *type checking* (ensuring that corresponding parameters are of compatible data types).

Another approach to parameter passing is to allow the calling program to make the external call with any list of actual parameters that the programmer writing the calling program chooses. The ordered list of actual parameters is passed to the called routine as a single table. This table of actual parameters contains information about each parameter. Although the table is ordered, the length of the table is not fixed. The called routine accepts the table as its formal parameter. That routine has the responsibility for performing any type checking required for correct execution of the routine.

This appendix explains the requirements for writing HP Business BASIC/XL programs to call externals that expect an actual parameter table as the external routine's formal parameter. Limit the use of this feature to those situations in which you perceive that the HP Business BASIC/XL standard external call feature is too restrictive. For example, you may want an external routine that prints values of HP Business BASIC/XL variables to a file. Rather than writing an external routine for each of the HP Business BASIC/XL data types, you can write a single routine to process variables that have any HP Business BASIC/XL data type passed as an actual parameter. In this case, the external must perform any necessary type checking, since HP Business BASIC/XL will pass variables of any type.

An Overview of ANYPARM

HP Business BASIC/XL's ANYPARM external call feature is designed to allow external calls with any number of actual parameters to a procedure in an Executable Library or in an object file that is linked into a compiled program file. Multiple calls to the same external procedure within an HP Business BASIC/XL program need not have the same number of actual parameters if the external is designed to process those parameters. Scalar and array variables of any HP Business BASIC/XL data type can be passed as actual parameters. String and numeric literals are legal as actual parameters. Also, both string and numeric functions that are evaluated prior to the external call are legal actual parameters.

Two methods are provided for calling ANYPARM external procedures. The first method utilizes an explicit ANYPARM EXTERNAL declaration and the CALL statement. The second method implements calls to the external by prefixing the name of the external to be called with an underscore. In the second method, a local implicit external declaration is made by HP Business BASIC/XL at the beginning of the execution of the subunit in which the call is made.

External procedures in the executable library that are to be called using the ANYPARM feature are written so that there are two formal parameters. The first is the number of actual parameters passed from HP Business BASIC/XL. The second formal parameter of the external procedure is a pointer to a formatted table of actual parameter information. The table contains the following information:

- The address of the value of each actual parameter stored in the format specified by that parameter's HP Business BASIC/XL data type.

- The type of the parameter at that address.

- A value indicating whether the parameter is a scalar or, if it is an array, the number of dimensions.

Data Structures in HP Business BASIC/XL

In order to correctly manipulate the actual parameters, it is important to have a thorough understanding of the data structures that HP Business BASIC/XL uses.

The method used to pass the actual parameters from the HP Business BASIC/XL program to the external procedure precludes the type checking of actual parameters. Therefore, HP Business BASIC/XL has no method of determining the number or the type of the parameters expected to be present in the table of actual parameters located at the address specified by the second formal parameter of the external procedure. Since only the addresses of the actual parameters are passed in the table, all HP Business BASIC/XL variables that are actual parameters are passed by reference. If a numeric or string constant or expression is an actual parameter, a temporary variable is created to store the value and the address of the temporary variable is passed.

On the return from the external procedure, HP Business BASIC/XL has no method for determining whether its internal data structures or data areas have been destructively altered. The programmer writing the external procedure needs to thoroughly understand the ramifications of the external procedure's interactions with all areas of memory. Direct heap management, which includes heap allocation in one external call and deallocation in a subsequent call, interferes with HP Business BASIC/XL's internal heap management and should be avoided.

Error Handling and Program Development

Error handling within the external procedure is the responsibility of the external procedure. HP Business BASIC/XL uses the XARITRAP intrinsic to replace MPE XL's arithmetic trap handler. HP Business BASIC/XL uses XLIBTRAP to enable an HP Business BASIC/XL library trap procedure. Use of either the XARITRAP or XLIBTRAP intrinsics will interfere with HP Business BASIC/XL's trap handling mechanism and should be avoided.

Programming errors encountered during development of the external procedure can be difficult to debug. Knowledge of the machine instruction set and the system debug facility prove to be invaluable tools in facilitating rapid program development. Relevant information is contained in the *Precision Architecture and Instruction Manual*, and the *MPE XL Debug Reference Manual*.

ANYPARM Calls From HP Business BASIC/XL

There are two methods of calling external procedures written to be called by the ANYPARM method. The first utilizes HP Business BASIC/XL's EXTERNAL and CALL statements. The second implements the underscore (_) to call the external. In both calling methods, the programmer has the responsibility for ensuring that the external being called is compatible with the formal parameter interface used by HP Business BASIC/XL's ANYPARM calling feature.

The external procedure can be included in any executable library. The order for resolving external procedure references for HP Business BASIC/XL programs executing in the interpreter is the same as that specified in the LIB = or XL = parameter when the interpreter is invoked. If the program is a compiled program, then the search order is the same as when the compiled program is starts executing.

Using ANYPARM EXTERNAL and CALL

The ANYPARM EXTERNAL statement is used to explicitly declare procedures that are to be called using the ANYPARM call feature. The CALL statement described in this section is used to transfer execution control to externals declared in an ANYPARM EXTERNAL statement.

Explicit declaration of procedures to be called by the ANYPARM method allows you to specify additional options concerning the scope and name of the external. External procedures can be declared in the main subunit of the program to be GLOBAL to the entire program. Otherwise, the external declaration is local to the subunit in which it is declared. A valid HP Business BASIC/XL identifier can be aliased to the names of externals which are not valid HP Business BASIC/XL identifiers, that is, procedure names which begin with an underscore.

The formal parameter list is not included in the ANYPARM EXTERNAL declaration since both the number and type of the formal parameters are not restricted.

Syntax

[GLOBAL] ANYPARM [EXTERNAL] *ap_name_clause_list*

Parameters

ap_name_clause_list

A list composed of *ap_name_clause* elements with the syntax:

```
                [{,}               ]
ap_name_clause  [{;} ap_name_clause ]
```

ap_name_clause

The identifier used to call the external from HP Business BASIC/XL together with an option that allows the name to be aliased to the actual name of the external. The syntax of *ap_name_clause* is:

ap_external_name [ALIAS "*alias_name* "]

ap_external_ name

The meaning is dependent on the presence or absence of the ALIAS option.

1. The ALIAS option is not present. *ap_external_name* is a valid HP Business BASIC/XL identifier in lower case that is the name of the external procedure in the executable library to be called from HP Business BASIC/XL. The maximum length of the name of the external is 60 characters.

2. The ALIAS option is present. *ap_external_name* is a valid HP Business BASIC/XL identifier used in the CALL statement in the HP Business BASIC/XL program to reference the *alias_name* external procedure in the executable library. The *alias_name* will be treated as the case-sensitive name of the procedure in the executable library.

In both cases, the *ap_external_name* is the identifier to be used with the CALL statement.

alias_name

The name of the external procedure. The *alias_name* is case-sensitive. The maximum length of the name of the external is 60 characters.

GLOBAL

Use of the GLOBAL option is restricted to the main program subunit. Use of the option specifies that the ANYPARM EXTERNAL declaration is accessible to all of the HP Business BASIC/XL procedures and functions in the program. This allows external calls to be made from the subunits without an additional ANYPARM EXTERNAL declaration.

The CALL Statement to Externals Declared Using ANYPARM EXTERNAL

The CALL statement for an ANYPARM EXTERNAL procedure is similar to that of other EXTERNALS.

Syntax

CALL *ap_external_name* [(*actual_param_list*)]

Parameters

ap_external_name An HP Business BASIC/XL identifier declared in an ANYPARM EXTERNAL or GLOBAL ANYPARM EXTERNAL declaration.

act_param_list The list of actual parameters to be passed to the external procedure. When more than two actual parameters are present in the list, each is separated from the next by a comma. Two consecutive commas are not valid. Each of the actual parameters can be a numeric or string identifier representing an HP Business BASIC/XL variable, or a literal, function, or expression that is evaluated prior to calling the external. Actual parameters that are HP Business BASIC/XL variables are passed by reference. To pass HP Business BASIC/XL variables by value, enclose the relevant identifier in a set of parentheses. All other actual parameters are evaluated, if required, and passed by value. Entire arrays passed as parameters must include the parentheses for the dimension information. An asterisk replaces each of the numbers that are required to reference an individual element of the array.

Examples

The following example shows the use of ANYPARM to call the externals ANYPARM_SUM and fileprint. Notice that the calls here are similar to calling any other external.

```
     100 GLOBAL ANYPARM EXTERNAL Fileprint
     110 ANYPARM EXTERNAL Sum ALIAS "ANYPARM_SUM"
     120 INTEGER Int1,Int2,Int3,Int4,Total
     130 Int1=1;Int2=2;Int3=3;Int4=4;Total=0
     140 CALL Fileprint("Beginning of Program.","Total is:",Total)
     150 CALL Sum(Total,Int1,Int2)
     160 CALL Sum(Total,Total,Int3,Int4)
     170 CALL Sum   ! No parameters are required for the call
     180 CALL Fileprint("New total is:",Total)
     190 CALL Suba(Total,10.50)
     200 CALL Fileprint("End of Program")
     210 END
     220 !
     230 SUB Suba(=INTEGER Substotal,REAL Price)
     235 REM Fileprint was declared as GLOBAL
     240    CALL Fileprint("Total Price is:",Substotal*Price)
     250 SUBEXIT
```

Using the Underscore to Call an ANYPARM External

The underscore is used to call external procedures in an executable library following an implicit local external declaration. By implicit, it is meant that no previous ANYPARM EXTERNAL statement in the HP Business BASIC/XL program is required to declare the external procedure name. The external to be called must be present in the executable library or program. Implicit declaration does not allow aliasing. Use of the underscore in a program subunit results in an implicit local external declaration. If the underscore is used in the main subunit, the implicit declaration is local to the main subunit. Refer to the following section, "Resolving Name Conflicts in Calls to ANYPARM Externals," for a description of HP Business BASIC/XL's method of determining which procedure is called when externals with the same names are declared both explicitly and implicitly within a program.

Syntax

_ap_external_name [*act_param_list*]

Parameters

ap_external_name A valid HP Business BASIC/XL identifier that is the name of the external procedure in the executable library to be called. The maximum length of the name of the external is 60 characters. The entry point name is *ap_external_name* in lower case unless the external is explicitly declared with an ALIAS clause.

act_param_list Same as the actual parameter list, *act_param_list*, in the CALL *ap_external_name* statement. Note that parentheses do not enclose the actual parameters when using the underscore to make a call to an external.

Examples

The following example shows the use of the underscore in a call to an ANYPARM External.

```
100 INTEGER Int1,Int2,Int3,Total
110 Int1=1;Int2=2;Int3=3;Total=0
120 _FILEPRINT "Beginning of Program","Total is:",Total
130 _ANYPARM_SUM Total,Int1,Int2
140 _ANYPARM_SUM Total,Int3,Int4
150 _ANYPARM_SUM  ! No actual parameters need be associated with a call
160 _FILEPRINT "New total is:",Total
170 CALL Suba(Total,10.50)
180 _FILEPRINT "End of Program."
190 END
200 !
210 SUB Suba(INTEGER Substotal,REAL Price)
220    _FILEPRINT "Total Price is:",Substotal * Price
230 SUBEND
```

Resolving Name Conflicts in Calls to ANYPARM Externals

When any of the GLOBAL explicitly, local explicitly, or local implicitly declared ANYPARM external procedures have the same name, HP Business BASIC/XL uses a hierarchy for determining which declaration is relevant to a specific call from within the program. The declarations are searched in the following order:

1. Local explicit ANYPARM declarations.
2. Local implicit ANYPARM declarations.
3. GLOBAL explicit ANYPARM declarations.

Since the names of all externals in the executable library must be unique, it is wise to give unique names to each of the externals referenced within your HP Business BASIC/XL program. Unique names for each external will avoid the mistake of calling non-ANYPARM externals when using the underscore. It will also ensure that you are calling the external that you intend to call.

The following examples are designed to clarify the actual external procedures called when conflicts arise between the various forms of ANYPARM external declarations. The ALIAS option has been used to allow distinction between calls to three ANYPARM EXTERNAL procedures, Test1, Test2, and Test3. In each example, "Call" (in the comments) refers to the procedure actually called.

Examples

The first example demonstrates the effect of aliasing the external procedure named Test1 to the HP Business BASIC/XL identifier, Test2.

```
10 ANYPARM EXTERNAL Test2 ALIAS "Test1"  ! Explicit local declaration
20 CALL Test2                            ! Call is made to Test1
```

In the following example, the explicit local declaration takes precedence over the implicit local declaration.

```
10 ANYPARM EXTERNAL Test2 ALIAS "Test1"  ! Explicit local declaration
```

```
     20 CALL Test2                            ! Call is made to Test1
     30 _Test2    ! Implicit local declaration  Call is made to Test1
```

In the following example, the explicit local declaration takes precedence over the explicit global declaration.

```
     10 GLOBAL ANYPARM EXTERNAL Test2 ALIAS "Test3"  ! Explicit global declaration
     20 ANYPARM EXTERNAL Test2 ALIAS "Test1"         ! Explicit local declaration
     30 CALL Test2                                   ! Call is made to Test1
```

In the following example, the implicit local declaration takes precedence over the explicit global declaration in the main subunit. However, in the Suba subunit, the explicit global declaration is used to determine which external to call.

```
     10 GLOBAL ANYPARM EXTERNAL Test2 ALIAS "Test1"  ! Explicit global declaration
     30 _Test2           ! Implicit local declaration  Call is made to Test2
     40 CALL Suba
     50 END
     60 SUB Suba
     70    CALL Test2    ! Call is made to Test1 as specified in GLOBAL declaration
     80 SUBEND
```

An explicit local external declaration also takes precedence over implicit local ANYPARM declarations. In the following example, a call is made to the Pascal external, Test4, using the ANYPARM underscore. Avoid calls to non-ANYPARM externals using the ANYPARM underscore.

```
     10 EXTERNAL PASCAL Test4  ! Explicit local external Pascal declaration
     20 _Test4                 ! Call is made to the external Pascal procedure Test4
```

Writing ANYPARM External Procedures

Writing an ANYPARM external procedure requires a thorough understanding of the method that HP Business BASIC/XL uses to implement ANYPARM calls. This section is divided into two subsections. The first subsection describes the requirements for formal parameters to be included in the procedure header and the actual parameter table passed to the ANYPARM external procedure. The second subsection describes the internal data structures that HP Business BASIC/XL uses to store the values of variables in memory.

Requirements for the External Procedure

The external procedure must have two formal parameters. The first is a value parameter to which is passed the number of parameters in the call's actual parameter list. The second is a value parameter to which is passed the address of the actual parameter table. In the MPE XL operating system environment, the first parameter type must be a 4 byte integer, and the second parameter type must be a 4 byte pointer.

The first formal parameter (the number of actual parameters) must be checked prior to using the address of the formal parameter table. If the number of actual parameters is zero, the address is set the value of Pascal's NIL pointer constant. On the MPE XL based HP 3000, this value is the four byte integer, 0.

The Actual Parameter Table

HP Business BASIC/XL prepares for the call to the external by building the actual parameter table. First, it must be determined whether the actual parameter is an HP Business BASIC/XL variable, an expression, or a literal. Expressions are evaluated and the result is assigned to a temporary variable. Literals are assigned to temporary variables of the appropriate type. If the value is a temporary variable, the address of the temporary variable is entered into the actual parameter table. Otherwise, the actual parameter is an HP Business BASIC/XL variable, the address of which is entered into the actual parameter table. The second entry to the actual parameter table is the type of value present. The third entry is the dimensionality of the value at the specified address. If the value is a scalar, then the dimensionality is zero. Otherwise, the dimensionality is the number of dimensions of the HP Business

BASIC/XL array. The data type and dimensionality information are contained in the flag word that immediately follows the address of the actual parameter. The structure of the resulting actual parameter table in memory is:

```
+----------------------------------------+
|        address of parameter 1          |
|----------------------------------------|
|              flag word 1               |
|----------------------------------------|
|        address of parameter 2          |
|----------------------------------------|
|              flag word 2               |
|----------------------------------------|
|                   .                    |
|                   .                    |
|                   .                    |
|----------------------------------------|
|        address of parameter n          |
|----------------------------------------|
|              flag word n               |
+----------------------------------------+
```

Flag Words - Data Type and Dimensionality Information

In the MPE XL operating system environment, each of the 4 byte flag words is divided into a left, high-order 2 bytes and a right, low-order 2 bytes. The left 2 bytes contains the data type of the associated actual parameter. The values that HP Business BASIC/XL uses to designate the corresponding HP Business BASIC/XL data types are the same as those returned by HP Business BASIC/XL's TYP and BUFTYP functions:

1	DECIMAL
2	STRING
5	SHORT INTEGER
6	SHORT DECIMAL
11	INTEGER
12	SHORT REAL
13	REAL

The right (low-order) 2 bytes will contain zero if the actual parameter is a scalar. A string is considered to be a scalar. If the actual parameter is either a string or numeric array, the right 2 bytes will contain the number of dimensions of the array.

HP Business BASIC/XL's Internal Data Structures

The address of the HP Business BASIC/XL variable entered into the actual parameter table is that of either the data value itself or the HP Business BASIC/XL data structure information that is stored together with values of that type.

Scalar Numeric Values

For numeric expressions that are evaluated and stored in a temporary variable, scalar numeric variables and individual elements of a numeric array, the address is that of the actual value stored in memory. The amount of memory used by each of these values is dependent on the data type as illustrated in the following table:

DECIMAL	8 bytes
SHORT INTEGER	2 bytes
SHORT DECIMAL	4 bytes
INTEGER	4 bytes
SHORT REAL	4 bytes
REAL	8 bytes

The Pascal data types used to declare HP Business BASIC/XL's DECIMAL and SHORT DECIMAL data types are explained in the section, "Pascal Data Structures for ANYPARM Calls," later in this appendix.

Scalar Strings

The data structure that HP Business BASIC/XL uses to store strings consists of two parts:

1. A dope vector that (in the MPE XL environment) consists of one 4 byte word to indicate the maximum number of characters allowed in the string (the declared length) and one 4 byte word to indicate the actual number of characters currently in the string.

2. The characters in the string.

The address that is passed to an ANYPARM EXTERNAL is the address of the dope vector, not the address of the first character. The structure of the string in memory is:

```
     Address in the
     actual parameter table
     references
        |      +---------------------------------------+
        -----> |             maximum length            |
               |---------------------------------------|
               |             actual length             |
               |---------------------------------------|
     char-     |    first| second  | third  | fourth   |
      acters:  |---------------------------------------|
               |   fifth  | sixth  | seventh| eighth   |
               |---------------------------------------|
               |                   .                   |
               |                   .                   |
               |                   .                   |
               +---------------------------------------+
```

NOTE HP Business BASIC/XL always reserves an extra byte at the end of all strings, including each element of string arrays. When computing the size of an element, this extra byte must be taken into account. For example, in a string array dimensioned with eight characters per string, each element will take up 20 bytes.

The actual number of bytes used to store a string can easily be calculated by the following formula:

```
     bytes_required = 8 + maximum_length +( 4 - ((maximum_length + 4) MOD 4))
```

Arrays

All arrays are preceded by a dope vector that describes pertinent information concerning the number of elements in the array and the number of dimensions. The address of the array in the actual parameter table passed to the external procedure is that of the first word in the array's dope vector.

Array dope vectors contain the following information:

1. The address of the first word of the data portion of the array.

2. The total number of elements (not words or bytes) in the array.

3. For each dimension:

   a. The total number of elements in the dimension.

   b. The lower bound for the dimension.

There can be up to six dimensions in an array.

The array dope vector has the following structure:

```
     Address in the
     actual parameter table
     references
        |
        |      +---------------------------------------+
        -----> |   address of beginning of data area   |-----
               |---------------------------------------|    |
               |       total number of elements        |    |
               |---------------------------------------|    |
               |      number of elements in dim 1      |    |
               |---------------------------------------|    |
               |          lower bound of dim 1         |    |
               |---------------------------------------|    |
               |                   .                   |    |
               |                   .                   |    |
               |                   .                   |    |
               |---------------------------------------|    |
               |      number of elements in dim n      |    |
               |---------------------------------------|    |
               |          lower bound of dim n         |    |
               |---------------------------------------|    |
               |              start of data            |<---
               |                   .                   |
               |                   .                   |
               |                   .                   |
               +---------------------------------------+
```

String Arrays

A string array is just an array of scalar strings. The address for the string array in the actual parameter table is actually that of the first word of information in the string array's array dope vector. The structure of a string array is on the next page.

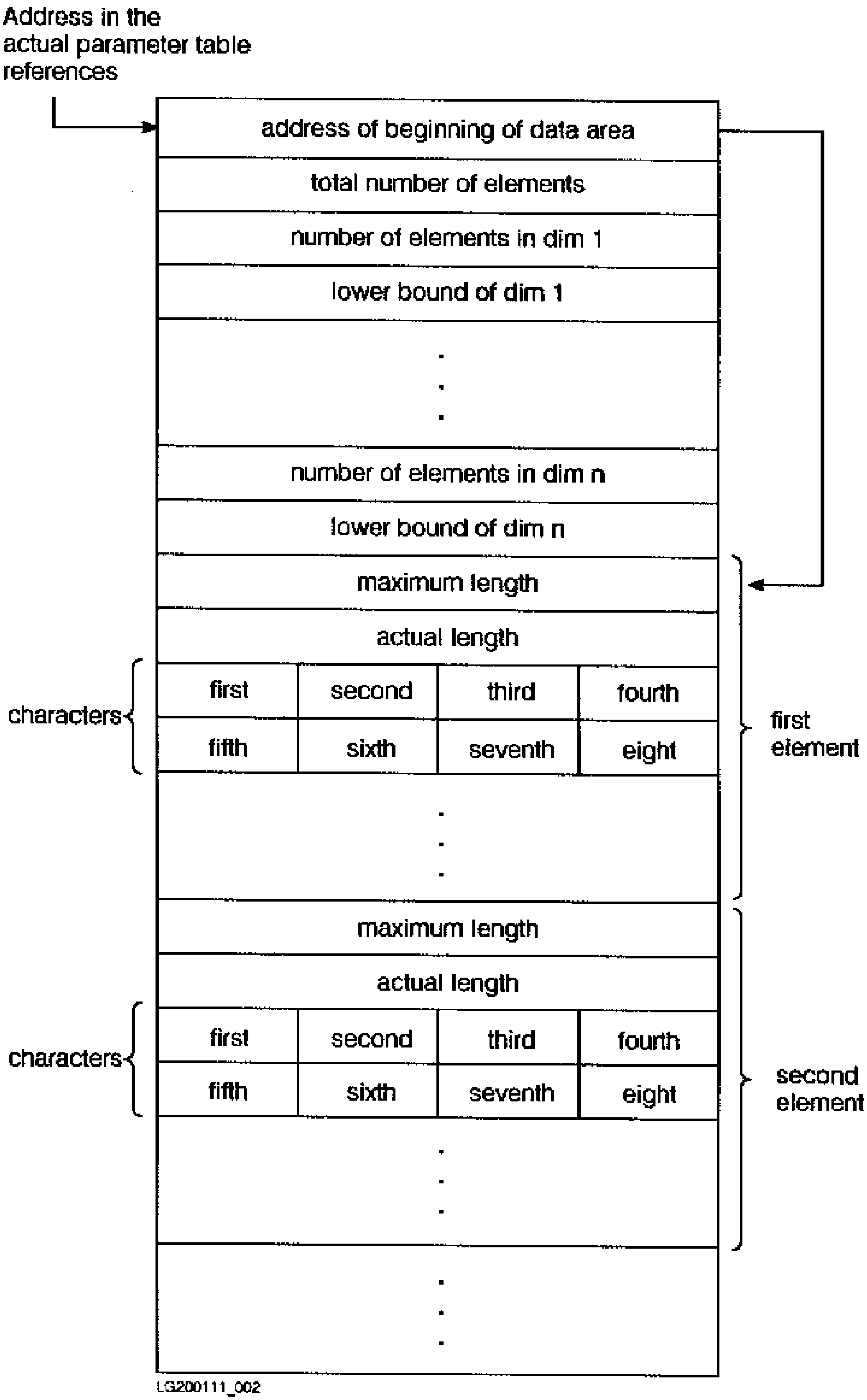


Figure G-1. String Array Structure

Example of a Simple Pascal ANYPARM Procedure

This section contains a Pascal procedure that can be called from HP Business BASIC/XL using the ANYPARM call interface. This procedure shows how to define the actual parameter table that the ANYPARM call requires. It also contains an example procedure that accepts the actual parameter table as a formal parameter.

```
$title 'SIMPLE_ANYPARM_PROGRAM / SIMPLE_EXAMPLE with
   INTEGER and SHORT INTEGER'$
$subprogram$
$tables on$
$code_offsets on$
$range off$

{*******************************************************************}
{*                                                                 *}
{*                         SIMPLE_ANYPARM_PROGRAM                  *}
{*                                                                 *}
{*Definition of the actual parameter table and the                 *}
{*constants and types required to process                          *}
{*Business BASIC/XL's SHORT INTEGER                                *}
{*and INTEGER data types. The addresses of the SHORT INTEGER       *}
{*and INTEGER values are passed in the actual parameter table      *}
{*to the procedure, SIMPLE_EXAMPLE.  SIMPLE_EXAMPLE prints the     *}
{*values of SHORT INTEGER and INTEGER values.                      *}
{*                                                                 *}
{*******************************************************************}
program simple_anyparm_program;

{-------------------------------------------------------------------}
{Machine constants and types specific for the MPE XL based HP3000.  }
{-------------------------------------------------------------------}
const
   c_min_mchn_wrd_int = minint;
   c_max_mchn_wrd_int = maxint;

type
   t_mchn_wrd_int = integer;
   t_half_mchn_wrd_int = shortint;

{-------------------------------------------------------------------}
{Constants representing Actual Parameter Types                      }
{The values in the actual parameter table that define the           }
{type of the parameter.                                             }
{-------------------------------------------------------------------}
const
   c_sinteger_type        = 5;
   c_integer_type         = 11;

   {---------------------------------------------------------------}
   {Scalar_value                                                   }
   {The pointer and associated variant record defining the HP      }
   {Business BASIC/XL value's storage format in memory.            }
   {---------------------------------------------------------------}
type
   tp_scalar_value = ^t_scalar_value;
   t_scalar_value = record
                       case integer of
                          1: ( sinteger_value : shortint);
                          2: ( integer_value  : integer     );
                       end;

{-------------------------------------------------------------------}
{The Actual Parameter Table                                         }
{An array of records describing the address, type and dimensionality}
{of each of the actual parameters.                                  }
{-------------------------------------------------------------------}
```

```
const
   c_max_num_parameters = 50;

type
   t_parameter_record = packed record
                        param_address       : tp_scalar_value;
                        param_type          : shortint;
                        number_of_dimensions: shortint;
                        end;

   t_actual_parameter_array = array [1..c_max_num_parameters] of
                                 t_parameter_record;

   tp_actual_parameter_array = ^t_actual_parameter_array;

{********************************************************************}
{*                                                                   }
{*                           SIMPLE_EXAMPLE                          }
{*                                                                   }
{*SIMPLE_EXAMPLE is a procedure written to accept an actual          }
{*parameter table as the formal parameter to the procedure.  The     }
{*purpose of the procedure is to write to a file the values of       }
{*all scalar actual parameters that have either an INTEGER or        }
{*SHORT INTEGER BASIC data type format.                              }
{*Actual parameters are processed in a for loop in which             }
{*the value of each valid parameter is written to $STDLIST.          }
{********************************************************************}
procedure simple_example(
   num_params           : integer;
   p_actual_param_table : tp_actual_parameter_array
                        );

var
   param_index : integer;
           {references entry in actual parameter table }
   tstfil      : text;
           {text file to which output is to be written }

begin  {procedure_example }

{--------------------------------------------------------------------}
{TESTFILE is opened in append mode so that information written to    }
{ the file by previous calls is not overwritten.                     }
{--------------------------------------------------------------------}
append( tstfil, '$STDLIST' );

writeln( tstfil
       , 'Number of parameters passed to SIMPLE EXAMPLE is: '
       , num_params:2
       );

{--------------------------------------------------------------------}
{Check to ensure that the number of actual parameters passed can be }
{processed by the external.                                          }
{--------------------------------------------------------------------}
if num_params > c_max_num_parameters then
   begin  {too many parameters to process }
   writeln( tstfil, '   Too many actual parameters passed to SIMPLE EXAMPLE.' );
   writeln( tstfil, '   Maximum number is: ',
                        c_max_num_parameters:1 )
   end    {too many parameters to process }
else

   begin  {simple_example's parameter array is large enough }

{------------------------------------------------------------}
{Process each of the entries in the actual parameter table       }
{referenced by the formal parameter, p_actual_parameter_array.}
```

```
     {------------------------------------------------------------}
        for param_index := 1 to num_params do
           begin  {for loop processing of the actual parameters }

           write( tstfil, param_index:3, '   ' );
           if p_actual_param_table^[param_index].number_of_dimensions
      = 0 then
           begin  {process scalar actual parameters }
           with p_actual_param_table^[param_index].param_address^ do
                                               {sinteger_value}
     {integer_value }
         case p_actual_param_table^[param_index].param_type of
            c_sinteger_type:
               writeln( tstfil, 'SHORT INTEGER ', sinteger_value:1 );
            c_integer_type:
               writeln( tstfil, 'INTEGER       ', integer_value:1 );
            otherwise
               write( tstfil, 'Actual parameter to SIMPLE EXAMPLE has an');
               writeln( tstfil, 'invalid data type.');
         end    {case }

        end    {process scalar actual parameters }

     else
        begin  {process actual parameters that are arrays }
        write( tstfil, 'Actual parameter to SIMPLE EXAMPLE must ');
        writeln( tstfil, 'be a scalar.')
        end    {process actual parameters that are arrays }

        end    {for loop processing of the actual parameters }

        end    {simple_example's parameter array is large enough }

     end;   {procedure simple_example }

     begin  {simple_anyparm_program }
     end.   {simple_anyparm_program }
```

Example of a Simple ANYPARM Call

Assume that the Pascal program presented above is in the file, PASPROG. To add the SIMPLE_EXAMPLE procedure to the local executable library named XL, do the following:

```
     :pasxl pasprog
     :linkeditor
     linked>buildxl xl
     linked>addxl from=$oldpass; to=xl
     linked>exit
     :
```

Consult the *HPLink Editor/XL Reference Manual* for more information.

Enter the HP Business BASIC/XL interpreter, specifying your group executable library. (Refer to "The Interpreter" in chapter 2). Within the interpreter, enter and execute the following program:

```
     >list
      !  testany
         10 ANYPARM EXTERNAL Example ALIAS "simple_example"
         20 INTEGER Int1,Int2          ! variable declarations
         30 SHORT INTEGER Sint1,Sint2
         40 REAL Real1
         50 DIM INTEGER Int_arr(2,2)
         60 CALL Example               ! a call with no parameters
         70 Int1=-2147483648
         80 Int2=2147483647
         90 CALL Example(Int1,Int2)    ! a call with two integer parameters
```

```
    100 Sint1=-32768
    110 Sint2=32767
    120 CALL Example(Sint1,Sint2)
    121                      ! a call with two short integer parameters
    130 CALL Example(Real1,Int_arr(*,*))
    131                      ! invalid real and array parameters
    140 Int_arr(2,2)=100000
    150 CALL Example(Int_arr(2,2))
    151                      ! a call with an array element parameter
    160 CALL Example(Sint1,Int_arr(1,1),Int2,Sint2,Int1,&
         (Sint1),(Sint1+Sint2),&
        Int_arr(2,2),(Int1+Sint2),"Beginning of invalid parameters",&
        Str$,Real1,Int_arr(*,*))

>run
Number of parameters passed to SIMPLE EXAMPLE is:  0
Number of parameters passed to SIMPLE EXAMPLE is:  2
1   INTEGER       -2147483648
2   INTEGER       2147483647
Number of parameters passed to SIMPLE EXAMPLE is:  2
1   SHORT INTEGER -32768
2   SHORT INTEGER 32767
Number of parameters passed to SIMPLE EXAMPLE is:  2
1   Actual parameter to SIMPLE EXAMPLE has an invalid data type.
2   Actual parameter to SIMPLE EXAMPLE must be a scalar.
Number of parameters passed to SIMPLE EXAMPLE is:  1
1   INTEGER       1000000
Number of parameters passed to SIMPLE EXAMPLE is: 13
1   SHORT INTEGER -32768
2   INTEGER       0
3   INTEGER       2147483647
4   SHORT INTEGER 32767
5   INTEGER       -2147483648
6   SHORT INTEGER -32768
7   INTEGER       -1
8   INTEGER       1000000
9   INTEGER       -2147450881
10  Actual parameter to SIMPLE EXAMPLE has an invalid data type.
11  Actual parameter to SIMPLE EXAMPLE has an invalid data type.
12  Actual parameter to SIMPLE EXAMPLE has an invalid data type.
13  Actual parameter to SIMPLE EXAMPLE must be a scalar.
>
```

Example of a Simple C ANYPARM Procedure

This section contains a C procedure that can be called from HP Business BASIC/XL using the ANYPARM call interface. This procedure shows how to define the actual parameter table that the ANYPARM call requires. It also contains an example procedure that accepts the actual parameter table as a formal parameter.

```
#define C_MAX_NUM_PARAMETERS  50
#define C_SINTEGER_TYPE        5  /* identifies BASIC SHORT INTEGER type */
#define C_INTEGER_TYPE        11  /* identifies BASIC INTEGER type       */
union u_scalar_value{             /* union to access parameter appropriately */
   short sinteger_value;
   int   integer_value;
} scalar_value;

struct parameter_record{          /* entry in the actual parameter array     */
   union u_scalar_value *param_address;
   short param_type;
   short number_of_dimensions;
};

/* simple_example
   simple_example is a procedure written to be called by the BASIC ANYPARM
   call mechanism.  A loop prints the values of scalar 16 and 32 bit integers
   and prints error messages for all other entries in the actual parameter
   table.
*/
simple_example(num_params, p_actual_param_table)
int num_params;
struct parameter_record p_actual_param_table[];
{
   int param_index;
   printf("Number of parameters passed to SIMPLE EXAMPLE is:%3d\n",
           num_params);
   if (num_params > C_MAX_NUM_PARAMETERS) {
      printf("Too many actual parameters passed to SIMPLE EXAMPLE.\n");
      printf("Maximum number is: %d\n", C_MAX_NUM_PARAMETERS);
      exit(0);
   }
   for (param_index = 0; param_index < num_params; param_index++){
      printf("%3d   ", (param_index+1));
      if (p_actual_param_table[param_index].number_of_dimensions == 0){
         switch (p_actual_param_table[param_index].param_type){
            case C_SINTEGER_TYPE:
              printf("SHORT INTEGER %d\n", (*p_actual_param_table[param_index].
                  param_address).sinteger_value);
               break;
            case C_INTEGER_TYPE:
              printf("INTEGER       %d\n", (*p_actual_param_table[param_index].
                  param_address).integer_value
                 );
               break;
            default:
               printf("Actual parameter to SIMPLE EXAMPLE has an invalid");
               printf(" data type.\n");
         }
      } else {
         printf("Actual parameter to SIMPLE EXAMPLE must be a scalar.\n");
      }
   }
}
```

Calling the C External SIMPLE_EXAMPLE

Assume that the C program presented in the previous section is in the file, CPROG. To add the SIMPLE_EXAMPLE procedure to the local executable library named XL, do the following:

```
:ccxl cprog
:linkeditor
linked>buildxl xl
linked>addxl from=$oldpass; to=xl
linked>exit
:
```

The output from the C procedure is the same as that from the Pascal procedure in the previous section.

Pascal Data Structures for ANYPARM Calls

This section contains a Pascal program that illustrates type and constant definitions required for ANYPARM externals.

```
$title 'ANYPARM.DECLS.BASIC/ANYPARM Data Declarations',page$
{-----------------------------------------------------------------------------}
{ ANYPARM EXTERNAL DATA DECLARATIONS                                          }
{-----------------------------------------------------------------------------}

{-----------------------------------------------------------------------------}
{ Constants related to the machine and operating system.                      }
{-----------------------------------------------------------------------------}
const
```

```
   c_bytes_per_pointer     = 4;
   c_bytes_per_integer     = 4;
   c_bytes_per_32_bit_word = 4;
   c_bytes_per_16_bits     = 2;

{----------------------------------------------------------------------------}
{ t_basic_data_types                                                         }
{ An enumerated type that associates a data type with a value.  Used as a    }
{ field selector for variant records to associate the relevant variant with  }
{ the data type.                                                             }
{----------------------------------------------------------------------------}

type
   t_basic_data_types = { 0 } ( basic_sinteger_type,
                        { 1 }   basic_integer_type,
                        { 2 }   basic_short_decimal_type,
                        { 3 }   basic_decimal_type,
                        { 4 }   basic_short_type,
                        { 5 }   basic_real_type,
                        { 6 }   basic_string_type
                              );

{----------------------------------------------------------------------------}
{ Data types that have corresponding Pascal data types.                      }
{----------------------------------------------------------------------------}
type
   t_short_integer_type = shortint;
   t_integer_type       = integer;
   t_short_real_type    = real;
   t_real_type          = longreal;

$page$
{----------------------------------------------------------------------------}
{ DECIMAL data type                                                          }
{----------------------------------------------------------------------------}
const
   c_dec_positive_mantissa = 12;
   c_dec_negative_mantissa = 13;
type
   t_shortint_rep_decimal = array [1..4] of shortint;
   t_dec_digit_pack       = packed array [-2..12] of 0..9;
   t_decimal_exponent_mantissa_sign_rep =
      packed record
         exponent           : -511..511;                 { 10 bits      }
         alignment_1_filler : 0..63;                     {  6 bits      }
         alignment_2_filler : shortint;                  { 16 bits      }
         alignment_3_filler : shortint;                  { 16 bits      }
         alignment_4_filler : -2048..2047;               { 12 bits      }
         mantissa_sign      : c_dec_positive_mantissa..
                              c_dec_negative_mantissa    {  4 bits      }
      end;

{----------------------------------------------------------------------------}
{ DECIMAL TYPE                                                               }
{ The first variant of the record is designed to serve as a record overlay   }
{ to quickly access the exponent and mantissa sign fields of the DECIMAL     }
{ representation of a number.                                                }
{                                                    1   1   1   1   1   1   }
{     bits:  0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   }
{ shortint |===============================================================| }
{    [1]   |<-------------- exponent ------------->|<- alignment_1_filler->| }
{          |===============================================================| }
{    [2]   |<-------------------- alignment_2_filler --------------------->| }
{          |===============================================================| }
{    [3]   |<-------------------- alignment_3_filler --------------------->| }
{          |===============================================================| }
{    [4]   |<-------------- alignment_4_filler ----------->|<mantissa sign>| }
{          |===============================================================| }
{                                                                            }
{ The second variant of the record is designed to serve as a record          }
{ overlay to access each of the decimal digits of the DECIMAL value.  The    }
{ digits are stored in elements 1 to 12 of the array.                        }
{                                                    1   1   1   1   1   1   }
{     bits:  0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   }
{ shortint |===============================================================| }
{    [1]   |<-digits[-2] ->|<-digits[-1] ->|<- digits[0] ->|<- digits[1] ->| }
{          |===============================================================| }
{    [2]   |<- digits[2] ->|<- digits[3] ->|<- digits[4] ->|<- digits[5] ->| }
{          |===============================================================| }
{    [3]   |<- digits[6] ->|<- digits[7] ->|<- digits[8] ->|<- digits[9] ->| }
{          |===============================================================| }
{    [4]   |<- digits[10]->|<- digits[11]->|<- digits[12]->|               | }
{          |===============================================================| }
{                                                                            }

{ NOTE: By definition, if shortint_rep[1] = 0 then the value of the DECIMAL  }
{       number stored at that location is zero.                              }
{----------------------------------------------------------------------------}
type
   t_decimal_type = packed record
      case integer of
         0:  ( decimal_rep  : t_decimal_exponent_mantissa_sign_rep );
         1:  ( digits       : t_dec_digit_pack );
         2:  ( shortint_rep : t_shortint_rep_decimal );
         3:  ( longint_rep  : longint );
      end;

$page$
{----------------------------------------------------------------------------}
{ SHORT DECIMAL data type                                                    }
{----------------------------------------------------------------------------}
const
   c_sdec_positive_mantissa = 0;
   c_sdec_negative_mantissa = 1;

type
   t_shortint_rep_short_decimal = array [1..2] of shortint;
   t_sdec_digit_pack            = packed array [-1..6] of 0..9;
   t_sdecimal_exponent_mantissa_sign_rep =
      packed record
         exponent      : -64..63;
         mantissa_sign : c_sdec_positive_mantissa..c_sdec_negative_mantissa;
         fill_16_bits  : shortint
      end;

{----------------------------------------------------------------------------}
{ SHORT DECIMAL                                                              }
{ The first variant of the record is designed to serve as a record overlay   }
{ to quickly access the exponent and mantissa sign fields of the SHORT       }
{ DECIMAL representation of a number.                                        }
{                                                    1   1   1   1   1   1   }
{     bits:  0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   }
{ shortint |===============================================================| }
{    [1]   |<------- exponent -------->| * |                               | }
{          |===============================================================| }
{    [2]   |<----------------------- fill_16_bits ----------------------->| }
{          |===============================================================| }
{                                                                            }
{ where the * is the bit used to represent the mantissa sign.                }
{                                                                            }
{ The second variant of the record is designed to serve as a record          }
{ overlay to access each of the decimal digits of the SHORT DECIMAL.  The    }
{ digits are stored in elements 1 to 6 of the array.                         }
{                                                    1   1   1   1   1   1   }
{     bits:  0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   }
{ shortint |===============================================================| }
{    [1]   |<-digits[-1] ->|<- digits[0] ->|<- digits[1] ->|<- digits[2] ->| }
{          |===============================================================| }
{    [2]   |<- digits[3] ->|<- digits[4] ->|<- digits[5] ->|<- digits[6] ->| }
{          |===============================================================| }
{                                                                            }
{ NOTE: By definition, if shortint_rep[1] = 0 then the value of the SHORT    }
{       DECIMAL number stored at that location is zero.                      }
{----------------------------------------------------------------------------}
type
   t_short_decimal_type = record
         case integer of
           0:  ( sdecimal_rep : t_sdecimal_exponent_mantissa_sign_rep );
           1:  ( digits       : t_sdec_digit_pack );
           2:  ( shortint_rep : t_shortint_rep_short_decimal );
           3:  ( integer_rep  : integer );
         end;

$page$
{----------------------------------------------------------------------------}
{ STRING data types                                                          }
{  An even length string declared as DIM A$[4] is stored in consecutive      }
{  32 bit words as:                                                          }
{    +-------------------------------------------+                           }
{  1 |              maximum_length               |                           }
{    +-------------------------------------------+                           }
{  2 |              logical_length               |                           }
{    +-------------------------------------------+                           }
{  3 |  char1   |  char2   |  char3   |  char4   |                           }
{    +-------------------------------------------+                           }
{  4 |  extra   | not used |                     |                           }
{    +-------------------------------------------+                           }
{  An odd length string declared as DIM Str$[3] is stored as:                 }
{    +-------------------------------------------+                           }
{  1 |              maximum_length               |                           }
{    +-------------------------------------------+                           }
{  2 |              logical_length               |                           }
{    +-------------------------------------------+                           }
{  3 |  char1   |  char2   |  char3   |  extra   |                           }
{    +-------------------------------------------+                           }
{----------------------------------------------------------------------------}
const
   c_max_str_len = 32767;
type
   t_string_length = integer;

   t_basic_string_type =
      record
         max_len : t_string_length;
         case integer of
            0:  ( actual_len : t_string_length;
                  bytes : packed array [1..c_max_str_len] of char
                );
            1:  ( pascal_string_view : string[c_max_str_len] );
      end; { record t_basic_string_type }

$page$
{----------------------------------------------------------------------------}
{ The constants that represent the amount of memory allocated for each       }
{ of the BASIC data types.                                                   }
{----------------------------------------------------------------------------}
const
   c_sizeof_short_integer = 2;   { number of bytes in a SHORT INTEGER }
   c_sizeof_integer       = 4;   { number of bytes in a INTEGER       }
   c_sizeof_short_real    = 4;   { number of bytes in a SHORT REAL    }
   c_sizeof_real          = 8;   { number of bytes in a REAL          }
   c_sizeof_short_decimal = 4;   { number of bytes in a SHORT DECIMAL }
   c_sizeof_decimal       = 8;   { number of bytes in a DECIMAL       }
```

```
$page$
{----------------------------------------------------------------------------}
{ t_basic_scalar_type                                                        }
{ Definition of a variant record for which the representation of the data    }
{ can be selected when the data type of the value is known.                 }
{----------------------------------------------------------------------------}
type
   t_basic_scalar_type =
      record
         case t_basic_data_types of
            basic_sinteger_type :
               ( sinteger_value      : t_short_integer_type );
            basic_integer_type :
               ( integer_value       : t_integer_type       );
            basic_short_type :
               ( short_value         : t_short_real_type    );
            basic_real_type :
               ( real_value          : t_real_type          );
            basic_short_decimal_type :
               ( short_decimal_value : t_short_decimal_type );
            basic_decimal_type :
               ( decimal_value       : t_decimal_type       );
            basic_string_type :
               ( string_value        : t_basic_string_type  );
      end; { record t_basic_scalar_type }

$page$
{----------------------------------------------------------------------------}
{ Array constant and type definitions.                                       }
{----------------------------------------------------------------------------}

{----------------------------------------------------------------------------}
{ Constants describing array bounds and limits.                              }
{----------------------------------------------------------------------------}
const
   c_max_array_bound    = 32767;
   c_min_array_bound    = -32768;
   c_max_array_elements = 32767;
   c_max_array_size     = 32767;  { bytes }
   c_max_array_dim      =  6;

{----------------------------------------------------------------------------}
{ Definition of the array descriptor that precedes the area used to store    }
{ the array data.                                                            }
{----------------------------------------------------------------------------}
type
   t_dimension_subrange = integer;

   t_array_single_dimension_descriptor =
      record
        dim_size    : t_dimension_subrange; { number of elements in dimension }
         lower_bound : t_dimension_subrange; { lower bound for dimension }
      end;  { record t_array_single_dimension_descriptor }

   t_array_dimension_descriptor =
      array [1..c_max_array_dim] of t_array_single_dimension_descriptor;

   t_array_descriptor =
      record
         total_elements : integer;
         bounds_info    : t_array_dimension_descriptor;
      end;

{----------------------------------------------------------------------------}
{ Definition of the DATA area of the array.                                  }
{----------------------------------------------------------------------------}
```

```
{----------------------------------------------------------------------------}
{ Definition of the maximum size and dimensions of each array type.          }
{----------------------------------------------------------------------------}
const
   c_sizeof_single_dimension_descriptor = 2 * c_bytes_per_integer;  { bytes }
   c_max_array_bytes_unavail =
      c_bytes_per_pointer +   { pointer to the data area }
      c_bytes_per_integer +   { stores total number of elements in array }
      c_max_array_dim * c_sizeof_single_dimension_descriptor; { bytes }

   {-------------------------------------------------------------------------}
   { c_max_array_bytes defines the maximum space that an array of any type   }
   { may use.                                                                }
   {-------------------------------------------------------------------------}
   c_max_array_bytes = c_max_array_size - c_max_array_bytes_unavail;

   {-------------------------------------------------------------------------}
   { Calculate the maximum index for each of the arrays.  Subtract one       }
   { element when calculating because the array indexing is zero based.      }
   {-------------------------------------------------------------------------}
   c_max_sinteger_array_index =
     ( c_max_array_bytes - c_sizeof_short_integer ) div c_sizeof_short_integer;
   c_max_integer_array_index =
      ( c_max_array_bytes - c_sizeof_integer ) div c_sizeof_integer;
   c_max_short_array_index =
      ( c_max_array_bytes - c_sizeof_short_real ) div c_sizeof_short_real;
   c_max_real_array_index =
      ( c_max_array_bytes - c_sizeof_real ) div c_sizeof_real;
   c_max_short_decimal_array_index =
     ( c_max_array_bytes - c_sizeof_short_decimal ) div c_sizeof_short_decimal;
   c_max_decimal_array_index =
      ( c_max_array_bytes - c_sizeof_decimal ) div c_sizeof_decimal;

   {-------------------------------------------------------------------------}
   { String arrays are contained in a "word_view", so max index is word, not }
   { element, related.  Individual array elements are always 4 byte aligned  }
   { because the t_basic_string_type record requires 4 byte alignment.       }
   {-------------------------------------------------------------------------}
   c_max_string_array_index      = c_max_array_bytes;
   c_max_string_array_word_index = c_max_array_bytes div
                                        c_bytes_per_32_bit_word;

{----------------------------------------------------------------------------}
{ Definition of the types that describe each array that is used to store     }
{ data of that type.                                                         }
{----------------------------------------------------------------------------}
type
   t_bas_sinteger_array =
      array [0..c_max_sinteger_array_index] of t_short_integer_type;
   t_bas_integer_array =
      array [0..c_max_integer_array_index] of t_integer_type;
   t_bas_short_array =
      array [0..c_max_short_array_index] of t_short_real_type;
   t_bas_real_array =
      array [0..c_max_real_array_index] of t_real_type;
   t_bas_short_decimal_array =
      array [0..c_max_short_decimal_array_index] of t_short_decimal_type;
   t_bas_decimal_array =
      array [0..c_max_decimal_array_index] of t_decimal_type;
   t_string_word_view =
      array [0..c_max_string_array_index div 4] of integer;

{----------------------------------------------------------------------------}
{ t_basic_array_type                                                         }
{ Definition of an array data type that has a variant for each of the data   }
{ types.                                                                     }
{----------------------------------------------------------------------------}
type
```

```
   t_basic_array_type =
      record
         case t_basic_data_types of
         basic_sinteger_type      : ( sinteger_array : t_bas_sinteger_array  );
         basic_integer_type       : ( integer_array  : t_bas_integer_array   );
         basic_short_decimal_type : ( short_decimal_array
                                                : t_bas_short_decimal_array );
         basic_decimal_type       : ( decimal_array  : t_bas_decimal_array   );
         basic_short_type         : ( short_array    : t_bas_short_array     );
         basic_real_type          : ( real_array     : t_bas_real_array      );
         basic_string_type        : ( word_view      : t_string_word_view    );
      end; { record t_basic_array_type }

$page$
{----------------------------------------------------------------------------}
{ t_basic_data_type                                                          }
{ The value referenced by the parameter address passed in the ANYPARM        }
{ actual parameter table has this type.  The correct representation of the   }
{ parameter is determined by the dimensionality and data type of the         }
{ parameter.                                                                 }
{----------------------------------------------------------------------------}
type
   t_dimension_range = 0..6; { a scalar has 0 dimensions, max array is 6 }

   t_basic_data_type =
      record
         case t_dimension_range of
            0 : ( scalar_value : t_basic_scalar_type );
            1..6 :
                (
                  {----------------------------------------------------------}
                  { Pointer to the beginning of the actual data area of the  }
                  { array.  The pointer is always used to reference the      }
                  { actual data.                                             }
                  {----------------------------------------------------------}
                  p_array_data : ^t_basic_array_type;

                  {----------------------------------------------------------}
                  { The area storing the total number of elements and the    }
                  { descriptor of each dimension - there are two words of    }
                  { information for each dimension.  The data area of the    }
                  { array will overwrite unused dimension information.       }
                  {----------------------------------------------------------}
                  array_descriptor : t_array_descriptor;

                  {----------------------------------------------------------}
                  { A field that defines the beginning of the actual data    }
                  { area - not to be used to reference the data.             }
                  {----------------------------------------------------------}
                  array_value : t_basic_array_type;
               );
      end; { record t_basic_data_type   }

   tp_basic_data_type = ^t_basic_data_type;

$page$
{----------------------------------------------------------------------------}
{ ANYPARM Parameter Type Field Values                                        }
{ The parameter type flag passed to the external for a parameter has the     }
{ same value as that which is returned by the TYP function.                  }
{----------------------------------------------------------------------------}
const
   c_decimal_type         = 1;
   c_whole_string_type    = 2;
   c_short_integer_type   = 5;
   c_short_decimal_type   = 6;
   c_integer_type         = 11;
   c_short_real_type      = 12;
   c_real_type            = 13;

$page$
{----------------------------------------------------------------------------}
{ The Actual Parameter Table                                                 }
{ An array of records describing the address, type and dimensionality of     }
{ each of the actual parameters.  t_parameter_record, a record which         }
{ contains fields for the address, type and dimensionality of a single       }
{ actual parameter in the actual parameter table, is defined.                }
{ t_short_basic_string_type is defined to allow processing of strings.       }
{ External declarations are made for the functions which process decimal     }
{ values.                                                                    }
{----------------------------------------------------------------------------}
const
   c_max_num_parameters            = 50;
   c_short_basic_string_max_length = 400;  { bytes }

type
   t_parameter_record = packed record
                             param_address       : tp_basic_data_type;
                             param_type          : shortint;
                             number_of_dimensions: shortint;
                          end;

   t_actual_parameter_array = array [1..c_max_num_parameters] of
                                 t_parameter_record;

   tp_actual_parameter_array = ^t_actual_parameter_array;

   t_short_basic_string_type =
        record
           max_len : integer;
           case integer of
              0:  (actual_len : integer;
                   case integer of
                     0:  ( bytes: packed
                          array [1..c_short_basic_string_max_length] of char );
                     1:  ( words:
                           array [1..c_short_basic_string_max_length div
                                     c_bytes_per_32_bit_word] of integer )
                  );
             1:  (pascal_string_view: string[c_short_basic_string_max_length]);
           end;
```

A Pascal ANYPARM Procedure Designed to Process Any Parameter

This section contains an example procedure that can process any of the Business BASIC/XL data types. The procedure uses the file of definitions shown in the previous section as an include file. The procedure is followed by the HP Business BASIC/XL program that calls this procedure. The section also contains a display that shows a logical representation of memory during the ANYPARM call to the Pascal procedure.

```
$standard_level 'os_features', os 'MPE/XL'$
$partial_eval on, literal_alias on$
$tables on, code_offsets on$
$diagnostic 'mapinfo_on'$
$optimize 'level2'$
$subprogram$

program pascal_example( input, output );
$include 'anyparm.decls.basic'$

$title 'ANYPARM_EXAMPLE/ANYPARM external testing all valid BASIC types',page$
{----------------------------------------------------------------------------}
{ ANYPARM_EXAMPLE                                                            }
{                                                                            }
{ This procedure is written to accept a pointer to an actual parameter table }
{ as the formal parameter to the procedure.  The actual parameter table      }
{ contains addresses referencing any of the data types.  The referenced      }
{ values can be either scalars or arrays.  The procedure will print the data }
{ type of the value and the value itself to a file named TESTFILE.  TESTFILE }
{ must be created before calling this procedure.  Testfile should be created }
{ as an ASCII file with a fixed record length of 80 bytes.                   }
{                                                                            }
{ ANYPARM_EXAMPLE contains the following second level procedures:            }
{                                                                            }
{   write_header         - writes a header for the call to the file.         }
{   process_string_array - writes the value of individual elements of a      }
{                          string array to the file.                         }
{   process_array        - writes the value of individual elements of        }
{                          numeric arrays to the file.                       }
{   process_scalar       - writes the value of all scalar types to the       }
{                           file.                                            }
{                                                                            }
{----------------------------------------------------------------------------}
procedure anyparm_example( num_params           : integer
                         ; p_actual_param_table : tp_actual_parameter_array
                         );
var
   param_index : integer;       { references entry in actual parameter table }
   tstfil      : text;          { text file to which output is written       }

$title 'EXTERNAL DECLARATIONS FOR THE FUNCTIONS TO CONVERT DECIMAL TYPES',page$
{----------------------------------------------------------------------------}
{ External declarations used to convert decimals to reals and short decimals }
{ to reals.  It is the caller's responsibility to check the values of        }
{ parameters passed to these procedures to ensure that no overflow occurs    }
{ during the conversion.                                                     }
{----------------------------------------------------------------------------}
const
   c_convert_short_decimal_to_real = 3;
   c_convert_decimal_to_real       = 1;

procedure bb_sdtor $alias 'bb_fp_decimal_convert'$(
     conversion_type : integer;   { c_convert_short_decimal_to_real }
 var short_dec_param : t_short_decimal_type;
 var longreal_param  : longreal
                                                  ); external;
procedure bb_dtor $alias 'bb_fp_decimal_convert'$(
     conversion_type : integer;   { c_convert_decimal_to_real }
 var decimal_param   : t_decimal_type;
 var longreal_param  : longreal
                                                 ); external;

$title 'ANYPARM_EXAMPLE/Example of ANYPARM external testing all BASIC types'$
$page$
{----------------------------------------------------------------------------}
{ procedure write_header of anyparm_example                                  }
{----------------------------------------------------------------------------}
procedure write_header(
     num_parms : integer;
 var tstfil    : text
                      );

begin  { procedure write_header}
writeln( tstfil, ' ' );
writeln( tstfil, 'enter the external anyparm_example' );
writeln( tstfil
       , 'the total number of parameters passed to anyparm_example is: '
       , num_params:2
       );
writeln( tstfil, 'param type' );
writeln( tstfil, '----- -------------' )
end;   { procedure write_header}
```

```
$title 'PROCESS_STRING_ARRAY of ANYPARM_EXAMPLE',page$
{-----------------------------------------------------------------------------}
{ procedure process_string_array of anyparm_example                           }
{-----------------------------------------------------------------------------}
procedure process_string_array(
     p_actual_param_table : tp_actual_parameter_array;
     param_index          : integer;
 var tstfil               : text
                              );
const
   c_2_spaces = '  ';

type
   t_pascal_string  = string[c_max_str_len];
   tp_pascal_string = ^t_pascal_string;

var
   array_element_num : integer;  { element number in the array of strings  }
   word_view_index   : integer;  { index for the word view of p_array_data }
   p_pascal_string   : tp_pascal_string; { pointer to string in the array  }
   array_element_word_length : integer;  { maximum length of the string    }

begin  { procedure process_string_array}
writeln( tstfil, 'STRING Array' );
with p_actual_param_table^[param_index].param_address^.p_array_data^,
                                                            { word_view      }
     p_actual_param_table^[param_index].param_address^.array_descriptor do
                                                            { total_elements }
   begin  { with}
   {--------------------------------------------------------------------------}
   { The maximum length of each string in the array is identical and can be  }
   { set to a constant for processing of the array.  Since the information   }
   { in word_view[0] is in units of bytes and an extra byte is always        }
   { reserved at the end of the string, a simple calculation is performed    }
   { to convert the 8 bit byte units to the 32 bit word units.               }
   {--------------------------------------------------------------------------}
   array_element_word_length :=
     ( ( word_view[0] + c_bytes_per_32_bit_word ) div c_bytes_per_32_bit_word )
     + 1 { for maximum length field } + 1 { for actual length field  };

   {--------------------------------------------------------------------------}
   { The array of strings is stored as an array of 32 bit words.             }
   { word_view_index is used to reference each of these words.               }
   {--------------------------------------------------------------------------}
   word_view_index := 1;

   for array_element_num := 0 to ( total_elements - 1 ) do
      begin  { processing individual strings }
      {-----------------------------------------------------------------------}
      { Move that part of the word_view array that contains the actual       }
      { characters of the string into the temp_string.                       }
      {-----------------------------------------------------------------------}
      $push, type_coercion 'conversion'$
      p_pascal_string := addr( word_view[word_view_index] );
      $pop$

      writeln( tstfil
             , c_2_spaces
             , array_element_num:3
             , c_2_spaces
             , p_pascal_string^
             );

      {-----------------------------------------------------------------------}
      { Increment to the index to the next element in the string array.      }
      {-----------------------------------------------------------------------}
      word_view_index := word_view_index + array_element_word_length;
      end    { processing individual strings }

   end    { with }

end;   { procedure process_string_array }

$title 'PROCESS_ARRAY of ANYPARM_EXAMPLE',page$
{----------------------------------------------------------------------------}
{ procedure process_array of anyparm_example                                 }
{----------------------------------------------------------------------------}
procedure process_array(
     p_actual_param_table : tp_actual_parameter_array;
     param_index          : integer;
 var tstfil               : text
                       );
const
   c_2_spaces = '  ';

var
   array_element_num      : integer;      { element number in the array of   }
                                          { appropriate type                 }
   temp_real              : longreal;             { used for conversion from }
                                                  {   decimal and short dec  }

begin
{----------------------------------------------------------------------------}
{ First de-reference the two pointers for the fields specified:              }
{----------------------------------------------------------------------------}
with p_actual_param_table^[param_index].param_address^.p_array_data^,
                                                      { short_decimal_array  }
                                                      { decimal_array        }
                                                      { sinteger_array       }
                                                      { integer_array        }
                                                      { short_array          }
                                                      { real_array           }
     p_actual_param_table^[param_index].param_address^.array_descriptor do
                                                      { total_elements       }
   begin  { with }
   {-------------------------------------------------------------------------}
   { Process the actual parameter by selecting the processing appropriate    }
   { for that type.                                                          }

   {-------------------------------------------------------------------------}
   case p_actual_param_table^[param_index].param_type of
      c_short_decimal_type:
         begin
         writeln( tstfil, 'SHORT DECIMAL Array' );
         for array_element_num := 0 to ( total_elements - 1 ) do
            begin  { short decimal element }
            bb_sdtor( c_convert_short_decimal_to_real
                    , short_decimal_array[array_element_num]
                    , temp_real
                    );
            writeln( tstfil
                   , c_2_spaces
                   , array_element_num:3
                   , c_2_spaces
                   , temp_real
                   );
            end;   { short decimal element }
         end;
      c_decimal_type:
         begin
         writeln( tstfil, 'DECIMAL Array' );
         for array_element_num := 0 to ( total_elements - 1 ) do
            begin
            write( tstfil
                 , c_2_spaces
```

```
           , array_element_num:3
           , c_2_spaces
           );
      {----------------------------------------------------------------}
      { Check to ensure that there will not be a numeric overflow when }
      { the decimal value is converted to a real.                      }
      {----------------------------------------------------------------}
      if
     ( decimal_array[array_element_num].decimal_rep.exponent > -308 ) and
     ( decimal_array[array_element_num].decimal_rep.exponent < 308 ) then
         begin  { decimal element }
         bb_dtor( c_convert_decimal_to_real
                , decimal_array[array_element_num]
                , temp_real
                );
         writeln( tstfil, temp_real );
         end    { decimal element }
      else
         writeln( tstfil, 'Decimal value is too large to convert' )
      end
   end;
c_short_integer_type:
   begin
   writeln( tstfil, 'SHORT INTEGER Array' );
   for array_element_num := 0 to ( total_elements - 1 ) do
      writeln( tstfil
             , c_2_spaces
             , array_element_num:3
             , c_2_spaces
             , sinteger_array[array_element_num]:1
             )
   end;
c_integer_type:
   begin
   writeln( tstfil, 'INTEGER Array' );
   for array_element_num := 0 to ( total_elements - 1 ) do
      writeln( tstfil
             , c_2_spaces
             , array_element_num:3
             , c_2_spaces
             , integer_array[array_element_num]:1
             )
   end;
c_short_real_type:
   begin
   writeln( tstfil, 'SHORT REAL Array' );
   for array_element_num := 0 to ( total_elements - 1 ) do
      writeln( tstfil
             , c_2_spaces
             , array_element_num:3
             , c_2_spaces
             , short_array[array_element_num]
             )
   end;
c_real_type:
   begin
   writeln( tstfil, 'REAL Array' );
   for array_element_num := 0 to ( total_elements - 1 ) do
      writeln( tstfil
             , c_2_spaces
             , array_element_num:3
             , c_2_spaces
             , real_array[array_element_num]
             )
   end;
c_whole_string_type:
   process_string_array( p_actual_param_table, param_index, tstfil );
```

```
      otherwise
         writeln( tstfil,'error in passed type')

   end    { case }

   end    { with }

end;   { procedure process_array }

$title 'PROCESS_SCALAR of ANYPARM_EXAMPLE',page$
{----------------------------------------------------------------------------}
{ procedure process_scalar of anyparm_example                               }
{----------------------------------------------------------------------------}
procedure process_scalar(
     p_actual_param_table : tp_actual_parameter_array;
     param_index          : integer;
 var tstfil               : text
                        );
var
   temp_real    : longreal;  { used for conversion from dec and short dec  }
   temp_integer : integer;

begin  { procedure process_scalar }
{----------------------------------------------------------------------------}
{ First de-reference the pointer for the associated fields specified.        }
{----------------------------------------------------------------------------}
with p_actual_param_table^[param_index].param_address^.scalar_value do
                                           { short_decimal_value             }
                                           { decimal_value                   }
                                           { sinteger_value                  }
                                           { integer_value                   }
                                           { short_value                     }
                                           { real_value                      }
                                           { string_value.pascal_string_view }
   begin  { with }
   {-------------------------------------------------------------------------}
   { Process the actual parameter by selecting the processing appropriate    }
   { for that type.                                                          }
   {-------------------------------------------------------------------------}
   case p_actual_param_table^[param_index].param_type of
      c_short_decimal_type:
         begin  { short decimal value }
         bb_sdtor( c_convert_short_decimal_to_real
                 , short_decimal_value
                 , temp_real
                 );
         writeln( tstfil, 'SHORT DECIMAL ', temp_real );
         end;   { short decimal value }
      c_decimal_type:
         begin
         {-------------------------------------------------------------------}
         { Check to ensure that there will not be a numeric overflow when    }
         { the decimal value is converted to a real.                         }
         {-------------------------------------------------------------------}
         if ( decimal_value.decimal_rep.exponent > -308 ) and
            ( decimal_value.decimal_rep.exponent < 308 ) then
            begin  { decimal value }
            bb_dtor( c_convert_decimal_to_real, decimal_value, temp_real );
            writeln( tstfil, 'DECIMAL               ', temp_real );
            end    { decimal value }
         else
            writeln( tstfil
                   , 'DECIMAL       '
                   , 'Decimal value is too large to convert'
                   )
         end;
      c_short_integer_type:
         begin  { short integer }
         temp_integer := sinteger_value;
         writeln( tstfil, 'SHORT INTEGER ', temp_integer:1 );
         end;   { short integer }
      c_integer_type:
         writeln( tstfil, 'INTEGER       ', integer_value:1 );
      c_short_real_type:
         writeln( tstfil, 'SHORT REAL    ', short_value );
      c_real_type:
         writeln( tstfil, 'REAL          ', real_value );
      c_whole_string_type:
         writeln( tstfil, 'STRING        ', string_value.pascal_string_view );

      otherwise
         writeln( tstfil,'error in passed type');

   end    { case }

   end    { with }

end;   { procedure process_scalar }

$title 'ANYPARM_EXAMPLE/Example of ANYPARM external testing all BASIC types'$
$page$
{----------------------------------------------------------------------------}
{                          main of ANYPARM_EXAMPLE                           }
{----------------------------------------------------------------------------}
begin  { anyparm_example }

{----------------------------------------------------------------------------}
{ TESTFILE is opened in append mode so that information written to the file  }
{ by previous calls is not overwritten.                                      }
{----------------------------------------------------------------------------}
append( tstfil, 'testfile' );

write_header( num_params, tstfil );

{----------------------------------------------------------------------------}
{ Check to ensure that the number of actual parameters passed can be         }
{ processed by the external.                                                 }
{----------------------------------------------------------------------------}
if num_params > c_max_num_parameters then
   begin  { too many parameters to process }
   writeln( tstfil, '   Too many actual parameters passed to ANYPARM_EXAMPLE' );
   writeln( tstfil, '   Maximum number is: ', c_max_num_parameters:1 )
   end    { too many parameters to process }

else

   begin  { anyparm_example's actual parameter array is large enough }

   {-------------------------------------------------------------------------}
   { Process each of the entries in the actual parameter table referenced by }
   { the formal parameter, p_actual_parameter_array.                         }
   {-------------------------------------------------------------------------}
   for param_index := 1 to num_params do
      begin  { for loop processing of the parameters }

      {----------------------------------------------------------------------}
      { Write the number of the parameter, the value(s) of which are about   }
      { to be written.                                                       }
      {----------------------------------------------------------------------}
      write( tstfil, param_index:3, '   ' );

      {----------------------------------------------------------------------}
      { Do the appropriate processing dependent upon the dimensionality of   }
      { the parameter in the actual parameter array currently being          }
      { processed.                                                           }
      {----------------------------------------------------------------------}
      if p_actual_param_table^[param_index].number_of_dimensions > 0 then
         process_array( p_actual_param_table, param_index, tstfil )
      else
         process_scalar( p_actual_param_table, param_index, tstfil )

      end    { for loop processing of the parameters }

   end;   { anyparm_example's actual parameter array is large enough }

writeln( tstfil, 'exiting anyparm_example' );
end;   { anyparm_example }

begin
end.
```

The ANYPARM Call

Assume that the Pascal program presented in the previous section is in the file, ANYPROG. To add the EXAMPLE procedure to the local executable library named XL, do the following:

```
:pasxl anyprog
:linkeditor
 linked>buildxl xl
 linked>addxl from= $oldpass; to=xl
 linked>exit
:
```

Consult the *HPLink Editor/XL Reference Manual* for more information.

Enter HP Business BASIC/XL and type the following program:

```
100 ! --- purge the file to which the &
    external writes the information --
110 PURGE "TESTFILE";STATUS=Status
120 !
130 ! --------- create the file to which &
    the external will write -------
140 CREATE ASCII "TESTFILE",RECSIZE=-80
150 !
160 ! ---------- declare and initialize variables ------------
170 REAL Real1
180 DIM Str8$[8]
190 DIM SHORT INTEGER Sint_array(1,1) &
   ! Assumes the OPTION BASE is zero
200 Real1=1.23E+45
210 Str8$="ANYPARM"
220 Sint_array(0,0)=1
230 Sint_array(0,1)=2
240 Sint_array(1,0)=3
250 Sint_array(1,1)=4
260 !
270 ! ---------------- call the external --------------------
280 _EXAMPLE Real1,Str8$,Sint_array(*,*)
290 !
300 ! ---------- print the contents of testfile -------------
310 COPYFILE "testfile"
320 END
```

Display of Memory during an ANYPARM Procedure Call

When the program is executed, the following is the layout of memory just as execution of the external, EXAMPLE, is beginning:

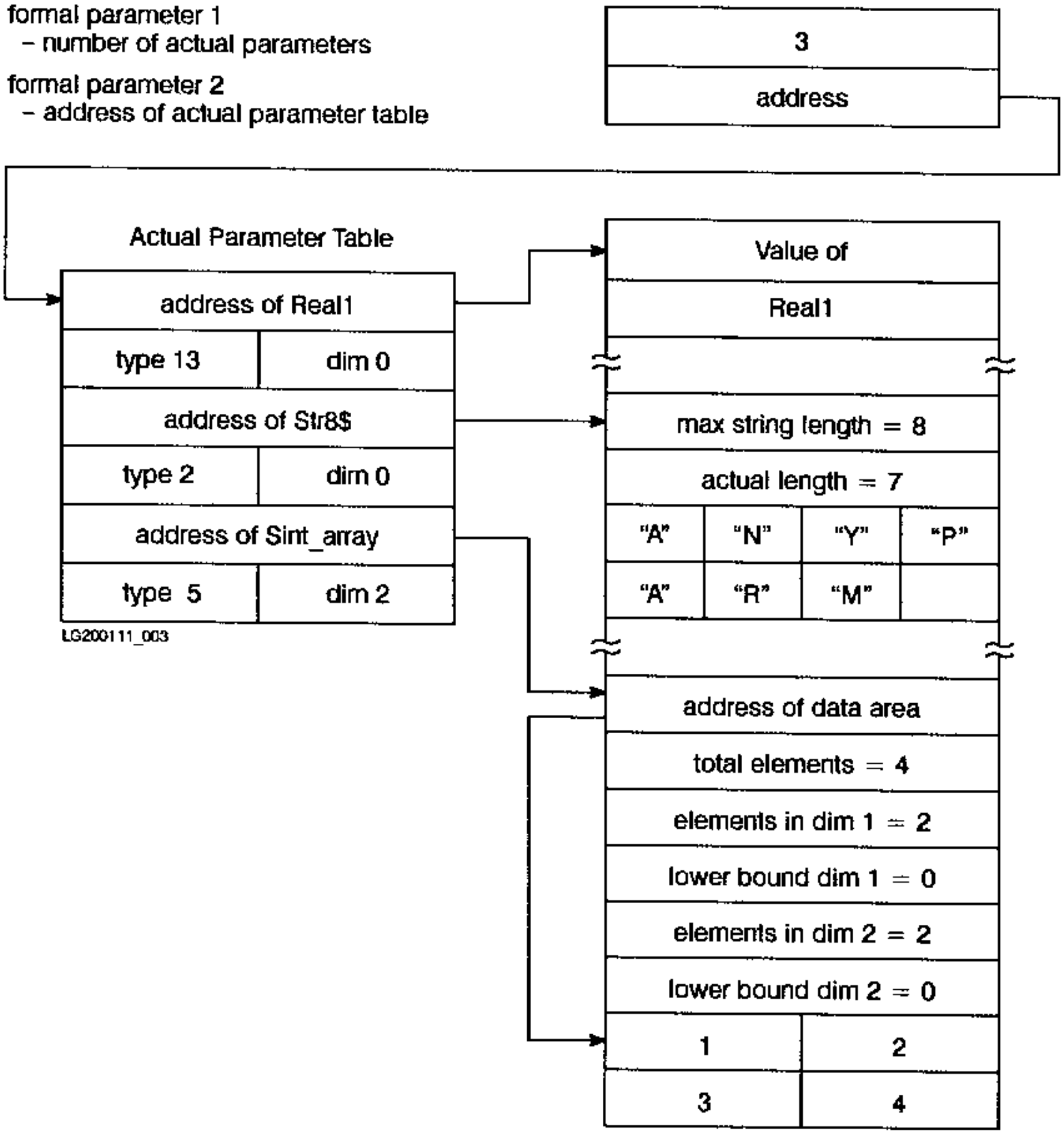


Figure G-2. Memory Layout

The Results of Program Execution

The first call to the external from within the interpreter will require substantially more time than subsequent calls. The reason is that the external procedure must be dynamically loaded before it can be called. Subsequent calls do not need to reload the external. The amount of time required to do the initial load is dependent on the size of the external being loaded. Externals called from compiled HP Business BASIC/XL programs are loaded when program execution starts.

The following is the result of program execution in the interpreter.

```
>run

hello from the external example
the total number of parameters passed to example is:  3
param type
----- ------------
1   REAL           1.2300000000000L+45
2   STRING        ANYPARM
3   SHORT INTEGER Array
  0  1
  1  2
  2  3
  3  4
exiting example
>
```

Differences Relative to BASIC/V

For those users familiar with BASIC/V's external procedure call feature, this section describes the differences between that feature and HP Business BASIC/XL's ANYPARM feature, and explains some of the reasons for the differences. Although the ANYPARM feature is designed to provide the same functionality as the BASIC/V feature, it is also designed to be consistent with other aspects of HP Business BASIC/XL. It is not meant to be identical with BASIC/V. An MPE/V machine word in this section refers to the 2 byte machine word of the HP 3000 running with the MPE V operating system.

An MPE XL machine word is a 4 byte machine word of the HP 3000 running under the MPE XL operating system.

The View from the External Procedure

In the BASIC/V feature, the field containing the number of parameters is located at Q+1 of the calling procedure, and the addresses and flag words immediately follow it on the stack. The HP Business BASIC/XL ANYPARM external procedure must declare two formal parameters: one for the number of parameters, and one for the address of the actual parameter table. This was done both to enable the external procedures to be written in Pascal, and to make it easier to migrate the external procedures to future HP computers.

Each flag word on MPE XL takes up an entire word and immediately follows the address of the parameter, instead of being packed three to a word and residing together in a block. The change makes it easier to obtain the required information and to port the feature to future computers.

The Flag Words

Data Types.   The values in the flag words that indicate the data types are not the same as those used by BASIC/V. The change was necessary to allow the use of the Business BASIC XL data types that don't exist in BASIC/V. The values are now consistent with the values returned by the HP Business BASIC XL TYP and BUFTYP built-in functions.

Sizes.   The size field (dimensionality) for a scalar string contains a zero, rather than a one as it did in the BASIC/V feature; the size field for a one-dimensional string array contains a one, rather than a two. The change is required to ensure that strings are handled consistently with the method used in HP Business BASIC/XL. Remember that HP Business BASIC/XL allows string arrays of up to six dimensions, whereas in BASIC/V strings arrays are limited to one dimension.

The Addresses

For arrays and strings, the address passed to the ANYPARM external references the first byte of the dope vector, rather than the beginning of the data area. All addresses are now byte addresses.

HP Business BASIC/XL Reference Manual

HP Business BASIC/XL
Reference Manual

Printed in U.S.A.

900 Series HP 3000 Computer Systems
HP Part No. 32715-90001
Edition First Edition
Printed Oct 1989
E1089

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Print History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the data indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

First Edition	October 1989	32715.00.00

Additional Documentation

Refer to the following manuals for further information on the MPE XL operating system, HP Business BASIC/XL and the IMAGE Database Management System:

- *MPE XL Commands Reference Manual* (32650-9003).
- *MPE XL Intrinsics Reference Manual* (32650-90028).
- *HPLink Editor/XL Reference Manual* (32650-90029).
- *Accessing Files Programmer's Guide* (32650-90017).
- *TurboIMAGE/XL Database Management System* (30391-90001).
- *SORT-MERGE/XL General User's Guide* (32650-90082).
- *System Debug Reference Manual* (32650-90013).
- *HP Pascal Reference Manual* (31502-90001).
- *Native Language Programmer's Guide* (32650-90022).
- *Data Entry and Forms Management System VPLUS/3000* (32209-90001).
- *HP Business BASIC/XL Migration Guide* (32715-90003).

Preface

This reference manual for the Hewlett-Packard HP Business BASIC/XL programming language provides programmers with information about the specific use of HP Business BASIC/XL as they prepare their applications. The manual is intended for reference only, to review the syntax and functions of HP Business BASIC/XL. It is not intended to teach the inexperienced programmer HP Business BASIC/XL. Information about migrating to HP Business BASIC/XL is contained in the HP Business BASIC/XL Migration Guide (PN 32715-90003).

The HP Business BASIC/XL language is for programming on 900 Series HP 3000 Computers, under the MPE XL operating system.

This manual contains the following chapters and appendixes:

Chapter 1 Provides an introduction to the HP Business BASIC/XL programming language.

Chapter 2 Explains the program development environment in which programs are created, modified, debugged, stored, and retrieved.

Chapter 3 Describes the elements of the HP Business BASIC/XL language.

Chapter 4 Describes all the statements available for creating a HP Business BASIC/XL program. They are arranged alphabetically for quick reference.

Chapter 5 Describes all the functions available within HP Business BASIC/XL. They are arranged alphabetically for quick reference.

Chapter 6 Explains input and output with HP Business BASIC/XL, including using the Native Language Support features.

Chapter 7 Describes the Report Writer.

Chapter 8 Explains the user-definable keys.

Chapter 9 Explains the HP Business BASIC/XL compiler. Lists statements that the compiler ignores and statements that cause compiler errors.

Appendix A Explains the errors that occur in HP Business BASIC/XL. They are listed by number.

Appendix B Lists the statements available to the user grouped by functionality.

Appendix C Explains the HP Business BASIC/XL Configuration Utility, which establishes default values for HP Business BASIC/XL.

Appendix D Gives the decimal and hexadecimal codes for the ASCII characters.

Appendix E Describes the HP terminals and language features.

Appendix F Explains JOINFORM, the FORMS/260 compatible forms package.

Appendix G Contains a technical discussion of the ANYPARM External Call Feature.

Conventions Used In This Manual

Notation **Description**

COMMAND Commands are shown in CAPITAL LETTERS. The names must contain no blanks and be delimited by a non-alphabetic character (usually a blank).

KEYWORDS Literal keywords, which are entered exactly as specified, appear in CAPITAL LETTERS.

parameter Parameters, for which you may substitute a value, appear in *italics*.

[ ] An element inside brackets is optional. Several elements stacked inside a pair of brackets means the user may select any one or none of these elements. Example:

[A]
 [B] user may select A or B or neither.

When brackets are nested, parameters in inner brackets can only be specified if parameters in outer brackets or comma place-holders are specified.

Example: [*parm1* [,*parm2* [,*parm3*]]]

may be entered as

parm1,parm2,parm3 or
parm1,,parm3 or
,,parm3 ,etc.

{ } When several elements are stacked within braces the user *must* select one of these elements. Example:

{A}
 {B}
 {C}

You must select A or B or C.

... An ellipsis in a syntax statement indicates that a previous bracketed element may be repeated. Within an example, vertical and horizontal ellipses show where portions of the example have been omitted.

<u>User Input</u> In examples of interactive dialog, user input is <u>underlined</u>.
Example: NEW NAME? <u>ALPHA1</u>

CONTROL Control characters are indicated by CONTROL. Example: CONTROL Y. (Press the CNTL key and Y simultaneously.)

RETURN RETURN indicates the carriage return key.