Eloquence Language Manual
B.06.32

Edition E1202

© Copyright 2002 Marxmeier Software AG.

Legal Notices

The information contained in this document is subject to change without notice.

MARXMEIER SOFTWARE AG 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. Marxmeier Software AG 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.

This document contains proprietary information which is protected by copyright. All rights reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Acknowledgments

© Copyright Marxmeier Software AG 2002. All Rights Reserved.

Marxmeier Software AG
Besenbruchstrasse 9
42285 Wuppertal
Germany

Eloquence is a trademark of Marxmeier Software AG in the US and other countries.

© Copyright Hewlett-Packard Company 1990-2002. All Rights Reserved.

This software and documentation are based in part on HP software and documentation under license from Hewlett-Packard Company. HP is a trademark of Hewlett-Packard Company.

Printing History

The manual printing date indicates its current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. New editions are complete revisions of the manual.The dates on the title page change only when a new edition or a new update is published.

Manual updates may be issued between editions to correct errors or document product changes. Manuals that are published on the Eloquence website (www.hp-eloquence.com/doc) may be updated more often, please visit this website periodically for the most recent versions. To ensure that you receive the updated or new editions, you should also subscribe to the appropriate product support service.

The software code printed alongside the date 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	Apr 1990	A.01.00
Second Edition	July 1991	A.03.00
Third Edition	January 1997	A.06.00
Fourth Edition	October 1997	A.06.00
Fifth Edition (E1202)	December 2002	B.06.32

Printed in the Federal Republic of Germany.

Printing History

Table of Contents

1 Things to Know Before You Start 13

Inside This Manual 14

Conventions 16

Related User Documentation 17

2 Starting Eloquence 19

Eloquence concepts 20

Starting the Run-Time Environment 22

3 Programming with Eloquence 25

Programming Guidelines 27

The character oriented development environment . . . 36

The Integrated Development Environment (IDE) . . . 68

4 Data Variables and Data handling 73

Types and Forms of Variables 75

Variable Names 77

String Variables 78

Numeric Variables 84

User defined Types 93
Declaring and Dimensioning Variables 100
Redimensioning an Array 106
Assigning Values to Variables 107
Eloquence keyboard handling 118
XPACK, XUNPACK statements 119
Memory Consumption 124

5 Operators and Functions 125
Operators and Expressions 126
Arithmetic Operators 127
Relational Operators 130
Logical Operators 132
Binary Operations 134
Operational Hierarchy 136
The Default ON/OFF Statements 137
Built-in Numeric Functions 138
Built-In String Functions 143
Defining a Function 148

Contents

6 Branching and Subroutines **151**

Unconditional Branching 154

Conditional Branching 156

Looping 158

Subroutines 162

Branching Using Softkeys 164

Error Testing and Recovery 167

The ON HALT Statement 169

The KEYBD function 170

Structured Programming 171

7 Subprograms **177**

Parameters 179

Multiple-Line Function Subprograms 184

Subroutine Subprograms 187

Subprogram Considerations 189

Busy Lines 193

8 File Storage **195**

Syntax Terms 198

Contents

File Structure 200
The Default Mass Storage Device 204
Cataloging Files (CAT) 205
Using message Catalogs 207
Identifying Volume Labels 208
Storing and Retrieving Programs 209
Storaging and Retrieving Data 216
Creating a Data File 217
Opening a Data File 218
Serial Access 222
Direct Access 228
Direct Word Access 232
Storing and Retrieving Arrays 236
Closing a File 237
Purging a File 238
File Storage Functions 239
Trapping EOR and EOF Conditions 243
Data Storage Requirements 245
Multi-User File Protection 246
Copying a File 247

Contents

Renaming a File 248

9 Output Operations 249

Restrictions on the Use of ASCII Control Characters 251

Selecting Output Devices 252

Printers 253

Audible Output (BEEP) 257

Displayed Output (DISP) 258

The LDISP Statement 260

POPUP BOX 262

The REFRESH Statement 264

Output Functions 265

Display Enhancements 268

The PRINT Statement 278

Formatted Output 280

Spool Files 292

Printer Control Functions 295

10 Matrix Operations 297

Redimensioning Arrays 299

Reading and Printing Arrays 300
Assigning Values to Arrays 303
Arithmetic Operations 305
Array Functions 307
Matrix Operations 308

11 System Clock 309
Returning the Current System Time and Date 310
Measuring Elapsed Time 311
Programmed Delays 312
Event Scheduling 314

12 Multiple Task Programming 315
Primary and Secondary Tasks 316
Configuration Requirements 317
Multi-Tasking Statements 318
Example Program Using TASK 321
Error Codes 321
HP-UX Background Processing 322
Programming Considerations 325

Contents

Performance Considerations 329

Functions for Task Control 331

13 Asynchronous Devices 335

TIO Statements 337

Eloquence Statements Used With TIO 341

Programming with TIO 344

14 Integrating C Functions (DLL) 353

Using DLL in Eloquence 355

Generating a DLL 359

Error Messages 370

15 Statement Flow Analyser 371

System Reset Conditions 384

TYP Function Return Values 385

ASSIGN Statement Return Variable 386

File Types 387

IMAGE Formatting Symbols 388

Storage Requirements 389

Display Enhancement Codes/Character Set Switching Codes 390

Contents

ASCII Character Codes 391

Introduction 398

Syntax List 399

Pack Errors 419

IMAGE Errors 420

PREDICATE Errors 421

SORT Errors 422

Report Writer Errors 423

FORMS Errors 425

TIMER Errors 426

TIO Errors 427

TASK Errors 428

User Defined Types Errors 430

HP-UX Errors 431

1

Things to Know Before You Start

Inside This Manual

This manual contains information on using the fourth generation program development environment Eloquence. Each chapter fully explains the features of Eloquence. Code examples are given to indicate the range of each command and provide tips for programming. Most example code can be tested; the remainder are illustrations taken from much larger programs and cannot be run alone. The manual is organized as follows:

Chapter 1 ”Things to Know Before You Start” contains an introduction to the use of this manual.

Chapter 2 ”Starting Eloquence” explains how to start both the development and run-time versions of Eloquence.

Chapter 3 ”Programming Guidelines” explains operations fundamental to the use of Eloquence (for example, developmental commands, error messages, entering, running and debugging a program).

Chapter 4 ”Data Variables and Data handling” covers all variable-related operations, including defining, dimensioning, and assigning values to both numeric and string variables.

Chapter 5 ”Operators and Functions” covers all of the built-in operators, mathematical functions, and string functions. Self-defined single-line functions are also covered.

Chapter 6 ”Branching and Subroutines” describes conditional and unconditional branching, looping, subroutines, error trapping, and ON HALT branching.

Chapter 7 ”Subprograms” shows how to define and access both multiple-line function subprograms and subroutine subprograms.

Chapter 8 ”File Storage” describes Eloquence statements and functions used to store and retrieve data and programs.

Chapter 9 ”Output Operations” covers audible, display, and printer output operations. It also covers formatting printed output.

Chapter 10 ”Matrix Operations” describes the operations available to handle arrays.

Chapter 11 ”The System Clock” describes the use of the system clock.

Chapter 12 ”Multiple Task Programming” describes the special program-

ming considerations which exist when tasks are added to a computer system.

Chapter 13 ”Asynchronous Devices” explains the terminal input/output (TIO) commands available to connect asynchronous devices to a computer system.

Appendix A ”Reference Tables” contains a number of helpful reference tables describing, for example, system reset conditions, file types, and ASCII character codes).

Appendix B ”Eloquence Syntax” alphabetically lists some of the statements, functions, and commands available with the Eloquence language.

Appendix C ”Error Messages” lists Eloquence error messages and gives a brief explanation after each message.

Appendix D ”Statement Flow Analyser” describes methods of analysing programs.

Glossary The glossary defines the common Eloquence terms.

Index The index contains page number references for the majority of Eloquence features.

Conventions

The following conventions are used throughout this manual:

- **Bold type** is used when a new term is introduced.
- `Computer font` indicates text to be input exactly as shown or text that is output from the system.
- *Italic type* is used for emphasis and titles of publications. It is also used to indicate parameters that are user defined.
- KEYCAP represents a key on the keyboard.
- shading represents the softkeys displayed on the computer screen.
- ... indicates that the previous variable can be repeated.
- [ ] indicates that information inside the brackets is optional. If there are brackets within brackets, the information within the inner bracket may only be specified if the information in the outer bracket is specified. Information may also be stacked in brackets. For example, A or B or neither may be selected when the following is shown:

$$\begin{bmatrix} A \\ B \end{bmatrix}$$

- { } indicates that one of the choices stacked within the braces must be selected. For example, A or B or C must be selected when the following is shown:

$$\begin{Bmatrix} A \\ B \\ C \end{Bmatrix}$$

NOTE: Notes contain important information and are set off from the text.

Related User Documentation

Additional information is included in the following manuals:

Eloquence Installation and Configuration Manual This contains detailed information about the installation and configuration on the HP-UX and NT platform.

Eloquence DBMS Manual This contains detailed information about the database commands briefly discussed in this manual.

Eloquence Forms Manual Contains complete instructions on drawing and using form images on the display screen.

Eloquence Report Writer Manual Contains report descriptions and details of execution and functions.

Eloquence Query Manual Describes the software used for accessing the Eloquence database.

HP-UX Operating System Manuals These manuals contain information on how to operate in the HP-UX environment.

Windows NT Manuals These manuals contain information on how to operate in the Windows NT environment.

2

Starting Eloquence

Eloquence supports the HP-UX and Windows NT platform. The runtime system is the same on both platforms, except the character I/O, which is available on HP-UX, only.

Eloquence concepts

Eloquence has been designed to operate in a distributed environment. It supports you in developing and maintaining your programs in a network environment. So again, Eloquence is keeping its promise of simple yet powerful program development and deployment.

Major features include:

- Supports development and debugging in a distributed environment
- State-of-the-art Development Environment
- Process attachment. Debug a process running on an arbitrary machine in the network. Even at the customer site.

How it works

Eloquence consists of several components, which can either run on a single system or spread across a network.

The Eloquence eloqd6 server

On each machine where Eloquence shall be used, an 'eloqd6' process has to be started. This process can be a 'master-' or a 'slave-server'. All 'slave servers' have to notify to the 'master server', so this server is the only server in the network who knows about all connected Eloquence systems. Which services a particular server provides, depends on the platform and the configuration. For example, an 'eloqd6' on a Windows 95 system can't have the same services as an 'eloqd6' on a HP-UX system, because the database server doesn't run on the Windows 95 platform.

The Eloquence 'eloqd6' server provides a single point to access services on a particular system. On the server, it provides authorization services, remote execution and works as a supervisor process. In addition, it provides file sharing capabilities to the Development Environment. On client systems it implements remote execution services and communicates with the ’dlgsrv’ on the Windows platform.

The Integrated Development Environment (IDE)

The Eloquence Development Environment (IDE) is only available on the Windows platform.

It provides a graphical program development environment. In order to execute programs for debugging, it either spawns an eloqcore process locally or asks a remote 'eloqd6-server' to start a debug process.

The Eloquence Development Environment provides its own file handling in addition to the capabilities already provided in the operating system. This file handling is provided by the 'eloqd6' process. By providing its own file handling, Eloquence can achieve:

- uniform and consistent file handling accross operating systems, e.g. case sensitive file name handling.
- consistent and secure handling accross administrative domains. By providing its own authentification mechanism, Eloquence programs can be transparently executed and debugged on remote systems regardless of the platform and your access permissions.

The Eloquence Runtime (eloqcore)

'eloqcore' is the part of Eloquence which executes your program. In a character oriented environment (on UNIX) it controls the terminal and provides its own development and debugging environment.

In a graphical oriented environment (on NT) it runs in a kind of background. For User I/O the GUI server is used and accessed via network. The development and debugging is done by the development server independently. So different tasks are distributed to different servers. For more information see chapter , Starting the Run-Time Environment.

The Dialog Server

The Dialog Server (DLGSRV) is currently only available in the Windows environment. It provides the graphical user interface (GUI) in a networked environment. For more informationsee chapter 1, HP Eloquence Dialog System.

The Database Server

The Eloquence database server provides access to the database. All requests are received via network, even if the eloqcore process runs on the same machine.

Since all of the Eloquence components are able to communicate over the network, you can easily configure a system with all the components distributed to different machines. The 'master-eloqd' server integrates the components regardless of their real locations and provides a homogenous view of the entire system to the Development Environment. This works even if you dial-in to a 'master-eloqd6' at the customer site.

For more information see chapter 2, Introduction, in data base manual.

Starting the Run-Time Environment

The eloq or eloqcore command, used to start Eloquence, is executed from the operating system prompt. Syntax is as follows:

$$\left\{\begin{matrix}\text{eloq}\\\text{eloqcore}\end{matrix}\right\}\ \text{[options] [program name]}$$

NOTE: The eloq program is not available on the MS Windows NT platform, because multiple tasking functions are not available, either. On a graphical user interface a second program is started to provide the user several dialogs.

Notice that you can specify either eloq or eloqcore. eloq is an interface to eloqcore; therefore, using eloqcore is slightly faster. The advantage that eloq has over eloqcore is that it allows you to use the Eloquence tasking statements—REQUEST #, ATTACH #, DETACH, and RELEASE #.

Specifying a program name along with the eloq or eloqcore command (for example, **`eloq ABC`**) causes the program specified to be run in the Eloquence environment. Once the program has completed, control returns to the HP-UX environment.

Executing the eloq or eloqcore command by itself simply shows the usage screen and returns control to the HP-UX environment. The usage screen shows the syntax of the command along with the options available. In other words, it shows the basics of how to use the command. Note that this does the same thing as the -help option.

Here is an explanation of the options associated with eloq and eloqcore:

-help Causes the usage screen to be displayed.

-r[ecord] ***file name*** Records every keyboard action performed by the user, and stores it in the specified file.

-p[lay] ***file name*** Plays back the keyboard action in the specified file, previously recorded using **`-r`** ***`file name`***.

-b[ackground] Suppresses terminal output from a program. It is useful when running a program in background. Refer to page 322 for more information.

-n[otask] Only applicable for the eloq command. This option disables Eloquence programmatic task processing. In other words, the tasking statements REQUEST #, ATTACH #, DETACH, and RELEASE # are disabled. Note that this option does *not* disable HP-UX background processing.

-t[race][*level*] Causes a trace to be performed on the specified program. If no program name is specified, a trace is performed on any program run in the Eloquence environment. Levels available are 0 (default) = trace lines, 1 = trace explicit assigments, 2 = trace all assignments. For more informationsee chapter , External Tracing.

-taskid # Starts a specific HP-UX process as the primary task. Replace # with the desired taskid number.

3

Programming with Eloquence

Eloquence provide two development environments to write, modify and degug an application; a character oriented one, which is running on the HP-UX platform and a graphical one, which is running on Windows NT and Windows 95. The graphical one can communicate with the Eloquence daemon on the HP-UX platform, as on the NT platform.

This chapter covers those Eloquence operations most useful to the programmer. It is split into three parts:

- Programming guidelines, which are common to both development environments
- The character oriented development environment
- The interactive development environment (IDE).

Programming Guidelines

Entering Programs

Be sure to check the following points before entering your first program:

- Unique line labels can be used. This optional label must be from 1 to 15 characters long (alphanumeric and underscore allowed), begin with a capital letter, and be followed by a colon. For example:

```
50 Display_name:DISP "Name is, ";A$
```

- Branching instructions can refer to the optional label. So **GOTO Display_name** would branch to the above example line.
- All numeric variables are assumed to be of real (full) precision unless specifically declared as integer or short precision. The default maximum length of string variables is 18 characters. Strings with a length other than the default of 18 characters will have to be declared in a DIM (dimension) statement. Details on declaring variable names and sizes are in page 73 .
- The maximum program line length is 512 characters. This length includes the line number. After each line is typed in, check it carefully and then enter it into memory by pressing RETURN. The line is automatically checked for syntax errors before it is stored. If an error is detected, an error message appears.
- Use the exclamation point (!) comment delimiter and the REM (remark) statement to annotate your programs. Some examples are shown later.
- An END statement should be the last line in the main program. END stops program operation and resets program pointers.

NOTE: String constants and comments are limited to 255 characters. Comments cannot be positioned beyond column 255.

The following guidelines belong to the character oriented development environment, only:

- Each program line must be preceded by a unique line number. Although line numbers are stored in ascending order, you can enter them in any order since they are automatically sorted when stored. All integer numbers from 1 through 32767 are allowed.
- Branching instructions can refer to either the line number (not recommended) or the optional label. So either **GOTO 50** or **GOTO Display_name** would branch to the above example line.
- To edit a previously-stored line, just move the cursor up to the image of the line remaining in the display, edit it, and store it again with RETURN. If the old line is no longer

in the display, you can either recall it with the FETCH command or simply retype the line number and line correctly. Then press RETURN. This will overwrite any existing line with the same line number.

- You also can enter your program using any HP-UX editor or utility (for example, vi or awk). The ASCII text file you create using an editor or utility must then be syntax checked and converted to an Eloquence program file. This is done using the store command from the HP-UX prompt. Refer to page 32 for more information on this procedure.

Entering a Sample Program

Now you are ready to enter your first program. The next example shows some useful programming tips. The program is a simple guessing game which first computes an integer number between 0 and 9 and then gives the operator three chances to guess it. This example uses the character I/O (INPUT, DISP), so it cannot run on the NT platform.

```
10 REM
20 REM  THIS IS A GUESSING GAME
30 REM AN EXAMPLE PROGRAM DESIGNED AS AN INTRODUCTION TO Eloquence
40 REM
50 REM  DATE WRITTEN: dd/mm/yy
60 REM  AUTHOR      : John SMITH
70 REM  INSTALLATION: XYZ Computers
75 INTEGER Try,Guess,Number,You,Me
80 REPEAT
90    DISP  "~~"     ! Cursor home, clear display
100    PRINT "I'm thinking of a number between 0 and 9"
110    PRINT "You have three guesses"
120    Number = INT(RND*10)
130    FOR Try=1 TO 3
140        PRINT "Enter guess number, ";Try
150        INPUT "Now, ";Guess
160        IF Guess=Number THEN Win
170        IF Guess<Number THEN PRINT "Too Low"
180        IF Guess>Number THEN PRINT "Too High"
190    NEXT Try
200    !
210 Lose:  PRINT "Sorry, the number was: ";Number;"."
220           Me = Me+1
230           GOTO Tally
240 Win:   PRINT "That's right!"
250        You=You+1
260 Tally: PRINT SPA(30);"Game Score"
270        PRINT SPA(30);YOU      ME"
280        PRINT SPA(30);You;SPA(5);Me
290        WAIT 4000
300        INPUT "Do you want to play again? (YES/NO), ";Reply$
310 UNTIL Reply$="NO"
320 END
```

Here are some useful programming tips:

- Annotate your listings using REM and ! comments (see lines 10 through 70). Comments can be used to explain sections of code and to insert blank lines. Both uses enhance program readability. Comment delimiters can be placed anywhere within a line. Any items occurring after the comment will be ignored by the Eloquence interpreter. The comment ends at the end of the line.
- Use prompts in your input statements (see lines 150 and 300), which appear in place of the question mark (?) when the program requests data.
- Define variables at the start of a program. This makes program maintenance easier. In Eloquence, a variable can be defined anywhere within a program and only needs to be quoted to be defined. For example, there would be no error if line 75 was omitted. If it did not appear, then the variable Try would be defined as a REAL variable (the default numeric type) when it first appeared in the program on line 140.
- Use line labels, such as “Win:” in line 240, to allow for relative branching to a named part of the program (see line 160). A branch to a label is executed just as fast as if the line number was referenced.
- Use the alternate character sets within string variables, such as the DISP statement in line 90, which clears the display buffer. The "~" represents cursor-home and clear-display. The keys corresponding to these characters can vary from terminal to terminal.
- To minimize confusion, avoid using variable names, subprogram names, or line labels that are identical to Eloquence keywords.
- Always include an END or STOP statement as the last line of every main program.

This programming tips are concerning line numbers and so they are useful for the character oriented development environment, only.

- So that you can add new lines of code later, number the lines in increments greater than one. For example, it was possible to insert line 75 between lines 70 and 80, as a gap in the numbering had been left. If lines 70 and 80 had been consecutive (71, 72) then the program would have to be renumbered before another line could be inserted between them.

  There is another way to insert lines, without concern for line numbers. This process involves using the **`list`** and **`store`** commands from the HP-UX prompt. List the program without line numbers (option -n) to an ASCII text file, insert your lines using an HP-UX editor or utility, and then store the program. When the **`store`** command converts the ASCII text file back to an Eloquence program file, it automatically renumbers the program in increments of one, starting with one. For more information see chapter , The LIST and list Commands.
- Spacing between line numbers and statements is not important, so you can indent FOR-NEXT loops or any other structures for clarity. (For more information see chapter , The INDENT Command, for exceptions.)

Programming Aids

Program Annotations

The Eloquence language provides two ways to include non-executable text fields in program listings—the REM (remark) statement and the exclamation point (!) comment delimiter. All characters following either REM or ! are stored with the program but not executed. Any combination of text can follow each keyword, as shown in the next example. Notice that ! comment fields can be placed either on lines by themselves or after program statements. REM statements cannot.

```
10 ! You can say any **** thing you wish
20 ! in a REM statement
30 ! *********************************************
40 ! *                                           *
50 ! *      DOCUMENT YOUR PROGRAMS WELL!         *
60 ! *                                           *
70 ! *********************************************
80 PRINT "Enter Sales Code, ";Sales_Code
90 IF Sales_code < 1000 THEN Invalid_entry! Error_routine
100          ELSE
110             Enter_asale     !Code OK so user permitted entry
120                             !to Sales database
130          END IF
```

The Bit Bucket

When you wish to run programs that involve time-consuming output operations but do not want the output, you may assign program output to the **Bit Bucket**. This is an imaginary device where data is dumped and cannot be retrieved.

To assign output from the printer to the bit bucket, specify device address 9 in the appropriate printer assignment statement—PRINTER IS, SYSTEM PRINTER IS, or PRINT ALL IS. For example, to send all PRINT and PRINT USING output to the bit bucket, execute the following:

```
PRINTER IS 9
```

NOTE: Be sure to read page 252 before using these statements.

Creating programlines dynamically

With the COMMAND statement it is possible create programlines dynamically in the program.The COMMAND statement executes a statement contained within a string expression. The syntax is as follows:

COMMAND *string expression* [,*return variable*]

Any executable, non-declarative statement (*not* DIM, COM, etc.) can be executed via COMMAND. For example, the following program uses a COMMAND statement to display a user selected variable:

```
10   INPUT Variable$
30   COMMAND "DISP "&Variable$
60   END
```

Note that the string expression can contain any combination of string characters within quotes, string variable names, substrings and string functions. String operations are described in page 73 .

One COMMAND statement can be executed by another COMMAND statement. However one COMMAND statement cannot call itself, nor can it call itself via intermediate COMMANDs. (The COMMAND statement is not recursive.) For example, the following sequence is permitted:

```
10   A$="COMMAND B$"
20   B$="DISP C$"
30   C$="OK"
40   COMMAND A$
50   END
```

However, the following sequence causes **ERROR 156**:

```
10   A$="COMMAND A$"
20   COMMAND A$
30   END
```

Execution of COMMAND Statements

- There are three general forms of COMMAND statement execution available:
  1. Scan/parse Eloquence statement contained in a string expression and execute.
  2. If the first character of a string expression is an exclamation mark (comment), a system command is executed. Stdout and stderr are redirected to SYPR. You can redirect them with output redirection.
  3. If the first character of a string expression is an colon (:), a system command is executed. In this form it is possible to start a interactiv process, as an editor or a shell, except the 'ksh'.
- It is possible to execute HP-UX and Windows commands through the Eloquence COMMAND statement. The HP-UX command must be enclosed in quotation marks (" ") and begin with an exclamation mark (!).

  It also supports the specification of a result string. If present, the output of a HP-UX command will be returned in the result string instead being output to the **SYSTEM PRINTER**.

  Here is an example:

  ```
  COMMAND "!uname -i",Serial$
  ```

```
DISP "Your serial number is ";Serial$
```

This example reads the serial number of the system and copies the output in the variable **Serial$**.

If the HP-UX or Windows command fails, you will receive error number 170, or a more detailed error code if you specified the return variable.

- It is possible to execute interactive HP-UX processes, as an editor or a shell, except the 'ksh'.

  Here is an example:

  ```
  COMMAND ":elm"
  ```

  This example starts the 'elm' and after quiting this process the eloqcore refreshes the screen and continues.

Space Dependency

Each line entered is automatically checked by the Eloquence interpreter. This check not only shows syntax errors in the line, but also assists with line spacing. Two methods of assistance are provided.

The SPACE INDEPENDENT Statement

This mode is the default. Power up or SCRATCH A will set the SPACE INDEPENDENT mode. The SPACE INDEPENDENT mode may be entered without a full reset by typing:

SPACE INDEPENDENT

or

SI

Store the following line using the default (SPACE INDEPENDENT) mode.

```
10 IF Hours_worked>40 THEN GOTO Overtime
```

You must key in each word correctly. All Eloquence keywords (for example, IF, THEN, and GOTO) must be in uppercase, while variables (Hours_worked), line labels (Overtime) and subroutine names must be in initial caps (meaning the first letter of each word is in uppercase, while the rest of the word is in lowercase). However, the spacing between words is not important.

```
10IFHours_worked>40THENGOTOOvertime
```

```
10 IF Hours_worked> 40T HE NG OTOOv ertime
```

Both the above examples will enter the desired line correctly.

The SPACE DEPENDENT Statement

When the SPACE DEPENDENT mode is set, spaces between keywords and variables (or the lack of them) become significant. Keywords and variables must be separated from each other by at least one space. However, Eloquence variables, subprogram names, and labels can now be typed in any combination of uppercase and lowercase characters.

To set the SPACE DEPENDENT mode, execute:

SPACE DEPENDENT

or

SD

Now, as each line is stored, the computer automatically sets all Eloquence keywords to uppercase and sets other words to initial caps. Text in quotes, in REM lines, and after ! comment fields is not affected.

To store the example line using the SPACE DEPENDENT mode, the case of each character is not important, but intraline spacing (between keywords, variables, labels, and subprogram names) is essential. Using the SPACE DEPENDENT mode type in and store either of these lines:

```
10 IF Hours_worked > 40 THEN GOTO OVERTIME
```

```
10 if hours_worked > 40 then goto overtime
```

Notice that each word is separated by at least one space. Leaving any spaces out in this example will result either in a syntax error or in an unexpected line. Note also that the underscore character, "_", cannot be altered. (A minus sign, "–", is not a lowercase underscore!)

In SPACE DEPENDENT mode, trying to store the following program line gives an appropriate error message.

```
10FORI=1TO10
```

The computer interprets this as an assignment statement and encounters an error when trying to assign the value 1 to the variable FORI.

Another problem is encountered in SD mode when entering the following:

```
50WHILEA=50
```

This example would pass the Eloquence interpreter without error. Unfortunately, the interpreter would assign the value 50 to the short variable WHILEA, instead of performing the correct interpretation (that is, to begin a WHILE loop, governed by the test A=50).

Here are some rules to follow when entering programs in SPACE DEPENDENT mode:

- Any variable name that is the same as a secondary keyword (function, logical operator, THEN, etc.) cannot be entered. To minimize confusion, it is a good idea to use variable names that are not the same as any keyword (primary or secondary).
- A line label that is identical to an Eloquence keyword cannot be entered at the start of a line.
- The first variable in an implied LET statement cannot be entered if it is the same as a keyword. This is also the case if the implied LET follows THEN.
- If a program line is not accepted while using the SPACE DEPENDENT mode, try to enter the line by setting the space independent mode, changing all characters in the line to their correct upper and lower case forms, and re-entering the line.

Error Messages

There are three main types of errors.

Syntax Errors

Each line entered is automatically checked for syntax errors. The system will not accept an invalid Eloquence statement, and will help you by showing where the statement fails. However, a mistake in typing that accidentally forms another valid Eloquence statement cannot be detected.

How you will be informed about the error, depends on the development environment

Run-Time Errors

These errors only appear when a program is run. A run-time error will halt the program and display the line number where the error was found. Consider this fragment of a payroll program:

```
190 INPUT "Please Enter Number of Employees;",Emp_quantity
200 Emp_bonus = Total_bonus/Emp_quantity
210 PRINT Emp_bonus
```

If the operator enters an Emp_quantity of 0, an error 31 will occure. Error 31 indicates attempted division by 0. Note that this error can only be detected at run time, as a division by zero will only occur if an Emp_quantity of 0 is entered.

The line number displayed by a run-time error need not be the incorrect line. It merely shows the point at which the error was detected. These run-time errors can be handled (or “trapped”) by the program by using the ON ERROR statement, as described in page 151 .

Internal Errors

If an irretrievable system condition occurs while Eloquence is running, it stops and issues an error message. The error message contains the reason Eloquence stopped, will be different displayed, on HP-UX and Windows NT. Please make a note of this information, and report it to Marxmeier Software AG. To continue working, restart Eloquence.

The character oriented development environment

Starting the character oriented Development Environment

The eloq or eloqcore command used to start Eloquence is executed from the HP-UX prompt. Syntax is as follows:

{ eloq | eloqcore } [options] [program name]

NOTE: The character oriented development environment is not available on NT platform

Here is an explanation of the options associated with eloq and eloqcore:

-help Causes the usage screen to be displayed. The usage screen shows the syntax of the command along with the options available. In other words, it shows the basics of how to use the command.

-r[ecord] ***file name*** Records every keyboard action performed by the user and stores it in the specified file.

-p[lay] ***file name*** Plays back the keyboard action in the specified file, previously recorded using `-r file name`.

-n[otask] Only applicable for the eloq command. This option disables Eloquence programmatic task processing. In other words, the tasking statements REQUEST #, ATTACH #, DETACH, and RELEASE # are disabled. Note that this option does *not* disable HP-UX background processing.

-t[race][*level*] Causes a trace to be performed on the specified program. If no program name is specified, a trace is performed on any program run in the Eloquence environment. Levels available are 0 (default)= trace lines, 1 = trace explicit assignments, and 2 = trace all assignments. Refer to page 282 for more information.

-sfa Activates the statement flow analyzer (SFA). The SFA records which statements have been executed and the execution time for each statement within a program.

-taskid # Starts a specific HP-UX process as the primary task. Replace #

with the desired taskid number.

Notice that you can specify either eloq or eloqcore. eloq is an interface to eloqcore; therefore, using eloqcore is slightly faster. The advantage that eloq has over eloqcore is that it allows you to use the Eloquence tasking statements—REQUEST #, ATTACH #, ATTACH, DETACH, and RELEASE #.

Executing the eloq or eloqcore command by itself starts the Eloquence program and causes a blank screen to appear. At this point, you are in the Eloquence development environment. You may create, edit, or run applications. To exit Eloquence and return to the HP-UX environment, type QUIT and press RETURN.

Specifying a program name along with the eloq or eloqcore command (for example, **`eloq ABC`**) causes the program specified to be run in the Eloquence environment. Once the program has completed, control returns to the HP-UX environment.

Input Redirection

You can use input redirection, as follows:

Example:

`eloqcore` *`program name input file name`*

and you can “press” a softkey by entering a line such as

```
:KEY#4
```

in the input data file.

Quitting Eloquence

The QUIT command terminates Eloquence, but gives a warning if a program has been modified but not stored, and gives you the alternative of cancelling.

The QQUIT command terminates Eloquence without giving such a warning.

QUIT and QQUIT do *not* check for “secondary” tasks.

Program Execution and Edit Commands

The following commands help you enter, run, and edit programs. It is important to note that these commands are only available when running the character oriented development environment. This is not intended to be a complete list of all development environment commands and statements. These are simply the specific commands to run and edit a program. For more information on storing and retrieving files, refer to page 195 and page 36 later in this chapter.

All these commands except `list` and `store` (lower case) are executed from within Eloquence. The `list` and `store` commands are executed from the HP-UX prompt. To execute a command, type it on the terminal and press RETURN.

The development commands are:

RUN Runs an Eloquence program.

CONTINUE Continues running a program.

AUTO Automatically numbers lines.

REN Renumbers program lines.

DEL Deletes one or more program lines.

FETCH Displays one program line for editing.

HOP 'Advanced step' functionality

LIST and `list` Displays or prints a list of program lines.

[RE-]STORE and store Syntax checks the current program or an ASCII text file and converts it to an Eloquence program file.

INDENT Indents all program lines and structured constructs. Refer to page 161 for a full description of this command.

All development commands except DEL can be executed from the keyboard but not from a program. For the most part, other Eloquence operations are stored in a program line or executed from the keyboard.

Parameter Values

The syntax rules used in this manual are listed under page 16 , and a complete list of terms and definitions are in the glossary. However, the following three terms are so often used as parameters that they are given here as well:

line number An integer from 1 through 32767.

line label A unique name assigned to a program line. It can contain up to 15 alphanumeric characters including the underscore. The first character must be a capital letter. The line label is separated from the line number by one or more spaces and must be followed by a colon.

line id A program line can be referenced by either its line number (GOTO 150) or line label (GOTO Routine). Using line labels allows your code to be moved and is a useful program documentation tool.

The RUN Command

The RUN command is used to load and run a specified program. Syntax for this command is as follows:

RUN [file specifier [,line id]]

The *file specifier* is a string expression containing the file name and, optionally, the volume specifier as described in page 195 . The optional *line id* specifies a starting line number. Program files (files with the extension .PROG) can be loaded and run in this way. For example, **`RUN "START"`** loads and runs a program named START.PROG. **`RUN "START",50`** loads and runs the program named START.PROG beginning at line 50. For more details on file operations, refer to page 195 .

If the program to be run is already in memory, it is not necessary to specify the *file specifier*. In this case the following syntax is appropriate:

RUN [line id]

RUN clears all the variables in a pre-run initialization and then begins execution of the program lines. The optional *line id* references a line in the main program and specifies that execution is to begin at that line. Omitting the *line id* causes execution to begin with the first line in memory. For example, executing **`RUN 150`** causes execution to begin with line 150.

A complete list of the pre-run default conditions can be found in page 383 .

The CONTINUE Command

Program execution, if suspended, may be resumed with the CONTINUE command. (Program suspension occurs if a PAUSE command is encountered or if BREAK or CTRL Y is pressed.)

$$\begin{Bmatrix} \text{CONTINUE} \\ \text{CONT} \end{Bmatrix} \quad [\textit{line id}]$$

If the optional *line id* is not specified, execution resumes with the next program line to be executed. No pre-run initialization occurs with CONTINUE.

The AUTO Command

The AUTO command allows lines to be numbered automatically as they are entered and stored.

AUTO [beginning line number [,incremental value]]

If neither parameter is specified, executing AUTO causes line numbering to begin with the last line number in memory plus 10 and is incremented by 10 as lines are stored. The optional *beginning line number* and the *incremental value* must be positive integers from 1 to 32767. For example, executing **`AUTO 5,5`** causes numbering to begin with 5 and increment by 5.

Automatic line numbering remains in effect until DELETE LINE is pressed or a program line is executed.

The REN Command

The REN (renumber) command causes program lines in memory to be renumbered.

REN [beginning line number[, incremental value]]

If no parameters are specified, all lines in memory are numbered in intervals of 10, beginning with 10. For example, **`REN 100,5`** causes program lines to be renumbered in intervals of 5, beginning with line 100 (resulting in 100, 105, 110, and so on). All line references in the program are automatically adjusted as the lines are renumbered.

The DEL Command

The DEL (delete) command is used to delete a line or section of a program.

DEL first line id [,last line id]

NOTE: DEL is allowed as a program statement; however, references to line labels must be local to the current environment (main program or subprogram).

Specifying one line identifier causes only that line to be deleted. Specifying two line identifiers causes that block of lines to be deleted. For example, to delete line 40 and lines 100 through 150 from a program, execute **`DEL 40`** and **`DEL 100,150`**.

The DEL SUB statement is available for deleting subprograms, as described in page 177 .

A program statement also can be deleted by re-entering only the program line number.

The FETCH Command

Use the FETCH command to bring to the display for editing any program line in memory.

FETCH [line id]

When a *line id* is not specified, the current line is displayed. If the specified line does not exist, the next-highest numbered line in memory is displayed. If there are no lines beyond the specified line, the highest numbered line in memory is displayed. For example, to fetch line number 300, execute **`FETCH 300`**.

To fetch the highest line in memory, execute **`FETCH 32767`**.

If no program line is active and no line id is specified, the first line is displayed.

FETCH, like LIST, outputs as default to the system printer, unless system printer 8 (display) has been specified.

HOP key

The HOP command provides the same functionality as the ’advanced step’ softkey on system softkeyset. In addition of entering the HOP statement in order to execute a 'step over' or 'advanced step' operation during program debugging, you may press ^G (ctrl-G).

LASTLINE

The LASTLINE keyword does return the last line number used in current program.

This makes it more convenient, to dynamically load subprograms or functions below the end of the program.

For example:

```
LOAD SUB "SUBX",LASTLINE+1,1
```

The LIST and `list` Commands

Two “list” commands are available in the development version. One is used in the Eloquence environment (LIST), the other in the HP-UX environment (`list`).

Both commands can produce a listing of a program; however the following tasks can only be done by one of the commands:

- Produce a partial listing (only LIST).
- List lines that contain a certain string (only LIST)
- Cross-reference check a program (only list).
- Create an ASCII text file (only `list`).
- Produce a listing without line numbers (only `list`).
- Produce a cross-reference printout (only `list`).

NOTE: Both commands are usable only on Eloquence program files (*not* .DATA, .FORM, or .ROOT files).

From Within the Eloquence Environment The LIST command (upper case) is used to obtain a listing of the program or a section of the program currently in memory. The listing is output on the device specified as the system printer (the display is set to be the system printer when Eloquence is started). Syntax for the LIST command is as follows:

```
LIST { [beginning line id [ , ending line id ]]                  }
     { string [ ; beginning line id [ , ending line id ]]        }
```

If no parameters are specified, the entire program in memory is listed. If one line identifier is specified, the program is listed from that line to the end. If two line identifiers are specified, that segment of the program, including beginning and ending lines, is listed. If a string is specified, the program lines containing that string are listed.

For Example:

```
LIST
```

Lists the entire program.

```
LIST 50
```

Lists the program beginning with line 50.

```
LIST 200,250
```

Lists lines 200 through 250.

```
LIST "ABC"
```

Lists all lines containing the string ABC.

```
LIST "ABC";50
```

Lists all lines beginning with line 50 that contain the string ABC.

```
LIST "ABC";200,250
```

Lists all lines from 200 through 250 that contain the string ABC.

```
LIST 50;20
```

Lists the 20 lines starting at line 50.

To output LIST to a device other than the display, first specify the device via the SYSTEM PRINTER IS statement. Syntax for this statement is as follows:

$$\text{SYSTEM PRINTER IS} \begin{Bmatrix} \textit{printer number} \\ \text{"}\textit{file name}\text{"} \end{Bmatrix}$$

All successive system printer output is now directed to the *device* specified rather than the display (printer number 8). Execute LIST to output a listing of the program. For example:

```
SYSTEM PRINTER IS 0
LIST
```

To redefine the display as the system printer, execute **`SYSTEM PRINTER IS 8`**. Executing **`SCRATCH ALL`** also returns output to the display, as does exiting and reentering Eloquence or the HP-UX operating system. See page 249 for more details on SYSTEM PRINTER IS and other printer options.

Within the HP-UX Environment

The **list** command (lower case letters) is used to obtain a listing of a program, to cross-reference check a program, or to convert a program file (.PROG) to an ASCII text file. The “list” command is executed from the HP-UX prompt. Syntax for the command is as follows:

list [-l | -n | -x [*l* | *c* | *v* | *s*]] *program name* [password] [>*user-defined file name*]

The following options are available:

-l Lists the specified program to the standard output device. This is the default unless the **-x** option is specified.

-n This option causes no line numbers to be listed, unless referenced.

-x[lcvs] This option does a cross-reference check on the specified file. Options for -x are l (labels and line numbers), c (constants), v (variables), and s (subprograms and functions). The default is for all the -x options to be set. Refer to page 284 for more information.

program name Replace *program name* with the name of a program file.

password If the program was stored in protected mode, then the program can be listed only, if the correct password is supplied.

user-defined file name The *user-defined file name* can be any name that conforms to HP-UX naming conventions. This file will contain ASCII text.

With the “list” command it is possible to convert an Eloquence program file to an ASCII text file that can then be changed using an editing program (for example, vi). This is done by making use of the >*user-defined file name* option. Once edited, the “store” command can convert the ASCII text file back to an Eloquence program file.

For example, the following “list” command converts the Eloquence program file ABC.PROG to an ASCII text file named abc.txt:

```
list ABC >abc.txt
```

At this point, the ASCII text file abc.txt can be edited using the vi editor. Once editing is completed, abc.txt is converted back to an Eloquence program file using the “store” command:

```
store -o ABC abc.txt
```

The [RE-]STORE and `store` Commands

The purpose of the **`store`** command is to do an Eloquence syntax check on an ASCII text file and convert it to an Eloquence program file. The **`store`** command is executed from the HP-UX prompt. Syntax for the command is as follows:

store [-help | -o *file name* | -n | -v | -s | -t width | -f line no | -i inc | -e] *input file* [,password]

Here is an explanation of the available options:

-help This option causes the usage screen to be displayed. The usage screen shows the syntax of the command along with the options available. In other words, it shows the basics of how to use the command.

-o *file name* This option is used to specify an output file name. If not specified, the input file name will be used. Whether the output file name is specified or not, the extension .PROG is automatically added to the file name. (Note that this option *must* be specified if reading from stdin.)

-n This option specifies that an output file *not* be created; however, the syntax of the input file is still checked.

-v This option causes the lines of the file being stored to be output

	to the standard output device. Normally this is the display.
-s	This option activates SPACE DEPENDENT mode. Refer to page 51 for more information.
-t wdth	tab width (default is 8, 0 = off) When editing your programs in an HP-UX editor (like vi) rather than the integrated Eloquence editor, tab characters provide a convenient way to format your source code. Tab characters in the source code caused a syntax error. When specifying a tab width, all tab characters read from input file are replaced by the appropriate number of spaces. A zero tab width will disable tab expansion. This may also be controlled by the $TAB directive.
-f lno	first line number (default is 1). When no line numbers are included in source code, store will provide its own, starting with 1.This commandline switch makes it possible to define the first line number to use for automatically generated line numbers. This may also be controlled by the $LINE directive.
-i inc	line number increment (default is 1) This commandline switch makes it possible to define the increment, used for automatically generated line numbers. This may also be controlled by the $LINE directive.
-e	If the -e commandline argument is present, store will output error messages is a format more suitable for automatic processing.
input file	Replace *input file* with the name of the ASCII text file to be converted to an Eloquence program file. It is also possible to replace *input file* with a hyphen (for example, `store -o ABC -`). Specifying a hyphen (-) as the input file will force reading from the standard input device (stdin). If this is done, -o *file name* must also be specified; otherwise, the store command will not know where to put the data it reads from the standard input device.
password	The program can be secured against listing and re-storeing by supplying a password

If the -e commandline argument is present, the error message will be of the following format:

```
store: "file.in", line 1: IMPROPER STATEMENT
^      ^          ^       ^
|      |          |       The error message
|      |          The line number of the input file
|      The name of the input file or stdin if read from stdin
To indicate it's a message from the store utility
```

For example:

Put the invalid statement below in a file named "file.in":

```
STOP HERE
```

Without the -e commandline argument, store will output the following error message:

```
STOP HERE
     ^
IMPROPER STATEMENT
store: "file.in", line 37
```

With the -e commandline argument, store will output the following error message:

```
store: "file.in", line 1: IMPROPER STATEMENT
```

To explain the **store** command, suppose you create an ASCII text file using the vi editor, and it contains the sample program shown in this chapter under page 44 . Also suppose the name of the ASCII text file is games.txt. The next step is to check the syntax of the ASCII text file and convert it to an Eloquence program file. This is done by issuing the following command:

```
store games.txt
```

Notice that no output file is specified; therefore, the input file name (games) is used to create the Eloquence program file (games.PROG). This program file is now ready to be run.

NOTE: If a variable contains the *linefeed* character, the **store** command cannot proceed.

The following “store” command also could be used:

```
store -o GAMES games.txt
```

The resulting Eloquence program file is GAMES.PROG. Notice that the .PROG extension is automatically added. If GAMES.PROG had been specified as the output file, the result would have been GAMES.PROG.PROG, so *do not* include the .PROG extension when specifying an output file.

Directives

The following directives, if included in the source code, make it possible to control the behavior of some store options within the code.

```
$TAB wdth
```

This will define a different tab character handling.

For example:

```
$TAB 8
```

Sets the tab expansion to 8 for subsequent source lines.

```
$LINE fline,inc
```

Defines a different automatically generated line numbers.

For example:

```
$LINE 1000,10
```

Presets the next automatically generated line number to 1000 and the line increment to 10.

```
$LINE 1000
```

Presets the next automatically generated line number to 1000 and the line increment to 10.

Using store with the C preprocessor

The C preprocessor is a macro processor which provides the following functionality:

- include files
- define and replace macros

Using the C preprocessor with Eloquence makes it possible to use the preprocessor functionality.

For example:

```
! This is a sample program Eloquence program using the
! C preprocessor

#define MAXLOOP 10

SUB Sample(INTEGER Partno)

! include the common block
#include "common"

! The subprogram body
   DBASE IS Db$
   IN DATA SET "PARTS" USE REMOTE LISTS Part_1
   DBGET(Db$,"PARTS",7,S(*),"@",Buf$,Partno)
   ...
   FOR I=1 TO MAXLOOP

      ...
   NEXT I
   ...

! include the definitions for PARTS
#include "PARTS"

SUBEND
```

To make this a Eloquence PROG file, you could use the following command line:

```
cc -E sample | store -o SAMPLE -
```

The C preprocessor replaces the #include directives by the referenced file content and the macros by the definitions. When a syntax error in an included file is detected, store is able to understand the preprocessor location information and will report the file name and position.

Please refer to the appropriate HP-UX documentation for more information.

Running a Program

Once the program lines are entered, you can run them immediately. For example, run the guessing game program on the character oriented development environment:

```
RUN "GAMES" ENTER

I'm thinking of a number between 0 and 9
You have three guesses
Enter guess number, 1
Now, 5
Too low
Enter guess number, 2
Now, 8
Too high
Enter Guess number, 3
Now, 7
Sorry, the number was: 6.

                                Game  Score
                                You     Me
                                 0       1

Do you want to play again?  (YES/NO)
```

This program will run until you enter **"NO"** (in capitals, with or without quotes) to the INPUT prompt **Do you want to play again (YES/NO)**. You may halt the program at any other time by pressing BREAK or CTRL Y. When a program is halted, the line to have been executed next is displayed.

After halting a program, you can restart it from the line displayed by using the CONT (continue) command or restart from the beginning by using RUN.

Program Termination

Five statements are available for halting program execution—STOP, END, PAUSE, WAIT and SLEEP. The STOP statement may appear anywhere in the program; it halts execution and resets all file and return pointers.

The END statement halts program execution, like STOP, but is intended to be the last line in the main program. Any lines beyond END can still be executed (via branching or subroutines). END cannot be executed from the keyboard.

The PAUSE statement suspends execution, but does not reset program or file pointers. This allows you to do such things as check program variables and modify lines. Execution resumes by executing the CONT (continue) command. PAUSE cannot be executed from the keyboard.

Here is a typical use for STOP and END:

```
120 INPUT "Enter your Sales file access code,";Salescode
130 IF Salescode  <  1000 THEN Access_error
140 ! User code invalid for all Sales file access
150 IF Salescode  <> 9999 THEN Unauth_error
160 ! User code invalid for file update, read only allowed
170 ! Pass above tests then OK to update Sales file
180 Sales_update! Start of Sales file update program
 .
 .
 .
560 Access_error:!
570 DISP "Sorry, you are not allowed to enter the Sales file"
580 STOP
590 Unauth_error:!
600 DISP "Sorry, You can't enter the Sales file update program"
610 DISP "You have read access only"
620 STOP
630 Endit:!
640 END
```

Note the use of STOP when there are multiple error conditions. Each error thus can display the correct reason for denying access, and the program will halt. No other error condition will be executed unnecessarily. Also note the use of the line label Endit, so the last line can be accessed without executing any error routines. STOP, of course, can be used anywhere within a program.

The WAIT statement delays program execution a specified number of milliseconds before continuing. The syntax is as follows:

WAIT [numeric expression]

The numeric expression can range from -2^{31} through $2^{31}-1$ (about 33 seconds); a negative number defaults to 0. The wait can be interrupted by pressing CTRL Y, in the character oriented runtime environment or a user-defined softkey. Examples are given later in this chapter and also in page 151 . More information on SLEEP is available in chapter 13.

Error Messages

When an error occurs, the terminal beeps and displays either an error number or a warning message. The number references a description that will help find the cause of the error. There are three main types of errors.

Syntax Errors

Each line entered is automatically checked for syntax errors. The system will not accept an invalid Eloquence statement, and will help you by showing where the statement fails. However, a mistake in typing that accidentally forms another valid Eloquence statement cannot be detected. Note the following example:

```
3*(5/7 RETURN
```

This example causes the message **IMPROPER EXPRESSION** to appear on the following line and the cursor to flash below the omitted closing parenthesis. If you had entered both parentheses correctly, but typed the minus operator "–" instead of the division sign "/", no error message would be displayed.

If you are using the **store** command and an error occurs during the syntax check, no program (.PROG) file is created.

Run-Time Errors

These errors only appear when a program is run. A run-time error will halt the program and display the line number where the error was found. Consider this fragment of a payroll program:

```
190 INPUT "Please Enter Number of Employees;",Emp_quantity
200 Emp_bonus = Total_bonus/Emp_quantity
210 PRINT Emp_bonus
```

If the operator enters an Emp_quantity of 0, the terminal will display **ERROR 31 IN LINE 200**. Error 31 indicates attempted division by 0. Note that this error can only be detected at run time, as a division by zero will only occur if an Emp_quantity of 0 is entered.

The line number displayed by a run-time error need not be the incorrect line. It merely shows the point at which the error was detected. These run-time errors can be handled (or “trapped”) by the program by using the ON ERROR statement, as described in page 151 .

Internal Errors

If an irretrievable system condition occurs while Eloquence is running, it stops and issues an error message. The error message contains the reason Eloquence stopped, the source-code file name, and the source-code line number. Please make

a note of this information, and report it to Marxmeier Software AG. To continue working, restart Eloquence. The following internal error message serves as an example of this type of error:

```
Internal error processing line 4420
Assertion failed: (ssp->ofs < seg->symsz)
file prerun.c, line 227
```

Storing a Program

Once the program lines are in memory, you can make a permanent copy on a disk file by executing the STORE statement. The syntax is as follows:

STORE "file name"

For example, to create a program file named GAME and store the guessing game program in it, type in and execute the following:

```
STORE "GAME"
```

This statement assumes that there is no other file on the disk already named GAME. Later, you may wish to store another version of the program in the same file by using the RE-STORE statement.

To see what files are now on the disk, execute the CAT (catalog) statement:

```
CAT
total 4
-rw-rw-rw-   1 john    tstctr      1070 Oct 12 10:59 GAME.PROG
```

The CAT listing shows you that GAME is a program file (.PROG) and requires 1070 bytes of disk space. The program was saved on October 12th at 10:59 (24 hour clock). The program belongs to the group *tstctr* and to the user *john*. The user, group, and others have read and write access to the program.

Later, use the LOAD statement to copy a previously-stored program back into the computer memory.

For more details on CAT, STORE, RE-STORE and the other storage operations, refer to page 195 .

NOTE: There is a difference between the STORE statement (upper case) and the **store** command (lower case). The STORE statement is executed from the Eloquence prompt or from within a program, while the **store** command is executed from the HP-UX prompt. Refer to page 29 and page 32 for more information on the **store** command.

NOTE: The STORE statement, like the **store** command, is only available in the development version of Eloquence (*not* the run-time version).

Listing a Program

To see a copy of the program, use the LIST command. If you have many corrections to make in a program, it is better to list the program on the display and then edit and store each line from the listing, rather than FETCHing or retyping individual lines.

To get a printed listing, first specify the printer's address by using the SYSTEM PRINTER IS statement. (The system printer is usually set to address 8.) Then execute LIST again. For more information on printers, see page 249 .

For example, to obtain a printed listing of the guessing game example, the program should be retrieved from the disk, the system printer set to 0, and the LIST command executed as follows:

```
LOAD "GAME"
SYSTEM PRINTER IS 0
LIST
```

NOTE: The command to obtain a printed listing of the guessing game example from the HP-UX prompt is as follows:

```
list GAME | lp
```

NOTE: This command produces a listing and pipes it to the lp command. Refer to the HP-UX documentation for more information on piping and the lp command.

The INDENT Command

The INDENT command is used to change all program line indentation. The syntax is as follows:

INDENT starting column, increment

All program lines are re-positioned with the first character of each keyword starting in the specified column. Keywords are not placed in that column when numbers or labels already occupy the column. Comment (!) lines are not moved. Comments following statements are not moved unless they would overlap the statement.

Structured constructs (IFTHENELSE, REPEAT, SELECT, LOOP, WHILE) are indented an additional incremental value. Intermediate keywords in a construct (CASE, CASE ELSE, etc.) are not indented. If structured constructs do not match properly, either the indentation does not occur at the starting column at the end of the program segment or the listing returns to the starting column too soon. In either case the indentation is reset to the starting column at the beginning of each program segment (main program, multiple line function, subprogram, etc.).

The INDENT command also re-positions the FOR/NEXT, DEFFN/FNEND, and SUB/SUBEND system keyword pairs and certain Report Writer constructs. A sample program indented using INDENT 10,2 is shown on the next page.

NOTE: The maximum line length is 512 characters.

Listed below is the structure of a program *before* executing **`INDENT`**:

```
10 !
20 !  INDENT does not change the position of comment lines.
30 !
40    INPUT " Enter a number:";X
50    IF X>10 THEN
60    PRINT "X>10."
70    ELSE
80    PRINT "X<=10";
90    IF X>0 THEN
100   PRINT ", but>0."
110   END IF
120   FOR I=1 TO X
130   Sum=Sum+X
                ! Trailing comments are not
140   NEXT I
             ! moved unless they get in
150   DISP "Sum from 1 - ";VAL$(X);" is";Sum  ! the way.
160   END IF
170   CALL Y
180   END
190   SUB Y
200   DISP "(ALMOST DONE)"
210   LOOP
220   READ A$
230   EXIT IF A$=" "
240   DISP A$
250   WAIT 1000
260   END LOOP
270   DATA 5 seconds,4 seconds,3 seconds,2 second,1 second,DONE,"
```

```
"
280   SUBEND
```

Listed below is the structure of the previous program *after* executing **INDENT 10,2**:

```
10 !
20 ! INDENT does not change the position of comment lines.
30 !
40        INPUT "  Enter a number:";X
50        IF X>10 THEN
60          PRINT "X>10."
70        ELSE
80          PRINT "X<=10";
90          IF X>0 THEN
100           PRINT ", but>0."
110         END IF
120         FOR I=1 TO X
130           Sum=Sum+X                       ! Trailing comments a
re not
140         NEXT I                            ! moved unless they g
et
150         DISP "Sum from 1 - ";VAL$(X);" is";Sum   ! in the way.
160       END IF
170       CALL Y
180       END
190       SUB Y
200         DISP "(ALMOST DONE)"
210         LOOP
220           READ A$
230           EXIT IF A$=" "
240           DISP A$
250           WAIT 1000
260         END LOOP
270
DATA 5 seconds,4 seconds,3 seconds,2 second,1 second,DONE," "
280       SUBEND
```

Interrupting a Program

The execution of a program can be stopped by pressing CTRL Y. This is known as interrupting a program. When CTRL Y is pressed in the *run-time* version of Eloquence, the program currently loaded is stopped, Eloquence is stopped, and control returns to the HP-UX operating system. In the *developmental* version of Eloquence, what happens when CTRL Y is pressed depends on where the program was started. If started from inside Eloquence, the program is stopped and control returns to Eloquence. If started from the HP-UX prompt, the program is stopped, Eloquence is stopped, and control returns to the HP-UX operating system.

Other interruptions are made when an error is encountered, the end of a file is reached, or a softkey (SFK) is pressed. These interruptions can be handled using the appropriate ON statement. (In other words, you may define action to be taken

when CTRL Y is pressed using the ON HALT statement.) ON interrupts may be disabled using the OFF statement. Executing SCRATCH ALL or starting Eloquence again also disables ON interrupts.

Program debugging

Once a program is stored in user memory, it can be executed, one line at a time, by using the single-step mode. The easiest way to enter the single-step mode is by pressing the BREAK key while the computer is at a ready state. The computer then displays the first line to be executed. Pressing the BREAK key successively executes each line, displays any result, and displays the next line to be executed. The single-step mode is automatically cancelled by RUN or CONT[INUE].

Another way to enter the single-step mode is to execute a GOTO statement from the keyboard. From this point, continue (as mentioned in the above paragraph) by pressing the BREAK key to execute each program line.

HOP (Debugging Aid)

Syntax:

HOP [line id]

HOP will resume execution until reaching either the next line if *line id* is not specified or the line specified. Execution will stop before executing line.

This is an advantage in the debugging process, because functions, subprograms and loops may be executed, and execution stops after processing.

Example:

```
10 FOR I=1 TO 10
20   X=X+I
30 NEXT I
40 DISP X
50 END
```

HOP 30 will resume execution until reaching line 30. Line 30 will be displayed. If you enter HOP execution will continue until line 40 is to be executed.

Tracing Program Operations

A convenient method of debugging program operation is to trace the logical flow and variable assignments. There are two tracing methods available—statement tracing and external tracing. These two methods can be used separately or together; however, in most instances they are used separately.

Statement Tracing

Seven tracing statements are available. Executing each statement sets a corresponding trace mode which outputs all related information to the device currently set as the system printer (SYSTEM PRINTER IS). The tracing statements available are as follows:

TRACE	Monitors all executed lines for specified program segments.
TRACE WAIT	Causes a WAIT instruction to occur after each TRACE output.
TRACE PAUSE	Causes a PAUSE instruction to occur at specified program lines.
TRACE VARIABLES	Lists the values of specified variables changed during the specified program segment.
TRACE ALL VARIABLES	Monitors the value of all variables within a program.
TRACE ALL	Enables both TRACE and TRACE ALL VARIABLES mode at the same time.
NORMAL	Cancels any previous TRACE modes.
XTRACE	Turn external trace within a program on or off.

NOTE: In the development mode of Eloquence tracing statements can be either programmed or executed from the keyboard. In the run-time mode the statements must already be present in the program to be run. If not, use the external tracing method.

The TRACE Statement

The TRACE statement is used to trace program logic flow in all or part of a program. When any branching occurs in a program, both the line number where the branch is from and the line number where the branch is to are output. Syntax for this statement is as follows:

TRACE [beginning line id [,ending line id]]

When a branch occurs, the output is in the following format:

TRACE--FROM line number TO line number

If no line ids are specified, all branches in the program are monitored. When one line id is specified, tracing does not begin until that line is executed. If a second line id is specified, tracing is switched off when that line is executed. Since TRACE operates dynamically, tracing may be switched on and off many times throughout the execution of a program. The beginning line id must be present in memory or tracing will never occur; the ending line id turns off the TRACE only if that line number is encountered.

The TRACE WAIT Statement

Use the TRACE WAIT statement in conjunction with any other TRACE statement to cause a specified delay after each statement that causes a trace output. It is used to monitor and examine trace output as it occurs. Syntax for this statement is as follows:

TRACE WAIT number of milliseconds

The delay is specified by a numeric expression in the range -32768 through 32767, which indicates the number of milliseconds after each trace printout. A negative number defaults to 0.

The TRACE PAUSE Statement

To check whether or not a line in a program is reached, or to monitor the number of times a specified line is executed, use the TRACE PAUSE statement. Syntax for this statement is as follows:

TRACE PAUSE line id [,numeric expression]

When only the *line id* is specified, the running program stops just before the specified line is executed. When the *numeric expression* is specified, it is rounded to an integer—call it N. The program will then stop when the specified line is reached for the Nth time; the line is not executed. Execution is resumed with that line by executing the CONT[INUE] command. Every subsequent execution of that line

causes execution to pause. This type of tracing can be disabled by letting the *line id* be one that is not in memory. The most efficient way is to let it be lower than the lowest-numbered line. Also see page 282 , later in this chapter.

The TRACE VARIABLES Statement

To trace changes in values of variables without using an output statement, use the TRACE VARIABLES statement. Syntax for this statement is as follows:

TRACE VARIABLES variable list

The *variable list* can contain simple numerics, strings, and array specifiers. There can be from one to five items separated by commas. The value of any variable which changes is printed. The output is in the following format:

TRACE--LINE line number variable name [(subscripts)] = value

The *line number* is the line in which the change occurred. If the change comes from a keyboard operation, the *line number* is replaced by KEYBOARD. The new value of the variable is indicated. In the case of an array, the values of the subscript at the time will be printed following the name.

When an entire array changes value, the printout is in the following format:

TRACE--LINE lne number array name (*) CHANGED VALUE

Tracing variables also detects changes in subprograms or variables passed by reference. For example, say `TRACE VARIABLES A,B` is executed and the value of A is passed by reference to a subprogram. If the corresponding variable in the subprogram is changed, a trace message for variable A occurs.

The TRACE ALL VARIABLES Statement

To trace all variables, use the TRACE ALL VARIABLES statement. Syntax for this statement is as follows:

TRACE ALL VARIABLES [beginning line id [,ending line id]]

When no line ids are specified, all variables are traced throughout the program. When one line id is specified, tracing begins after that line is executed. The ending line causes tracing to stop after that line is executed. TRACE ALL VARIABLES cancels and is cancelled by TRACE VARIABLES.

This method of tracing can be turned off by letting the first line id be one not in memory, such as an undefined label or line number lower than the lowest line number in memory. Also see page 282 , later in this chapter.

The TRACE ALL Statement

Use the TR

logic and variables. This statement is equivalent to executing both TRACE and TRACE ALL VARIABLES.

TRACE ALL

Although the volume of printout is high, TRACE ALL is useful if a logic problem in a program has not been isolated with selective tracing.

The NORMAL Statement All tracing modes are cancelled by either executing any SCRATCH statement or by using the NORMAL statement:

NORMAL

External Tracing

External tracing is initiated when -t is specified with the eloq command, or with XTRACE inside a program. Note that the eloq command is executed from the HP-UX prompt. The syntax is as follows:

eloq -t [race] [level] [program name] [2>trace file]

If no program name is specified, the trace is performed on every program run. If a program name is specified, the trace will occur on only that program.

Three options are available for the *level* parameter—0 (default), 1, and 2.

Level 0 monitors all executed lines.

Level 1 monitors explicit assignments (for example, **`A=B*5`**).

Level 2 monitors implicit assignments like dbget and unpack.

The optional 2>*trace file* parameter is used to put the output of the trace into a file or display the output on another terminal. If 2>*trace file* is not specified, the trace output is displayed on the terminal, prior to the program output.

To send trace output to a file, replace *trace file* with the desired file name. You can then send this file to the printer or display it on the terminal. Here is an example:

```
eloq -t ABC 2>abctrace
```

This command (1) executes the program ABC, (2) performs a trace, (3) routes the trace output to the file abctrace, and (4) displays the program output on the terminal.

To display trace output on another terminal, replace *trace file* with the terminal address. Here is an example:

```
eloq -t ABC 2>/dev/tty1p5
```

This command (1) executes the program ABC, (2) performs a trace, (3) displays trace output on terminal tty1p5, and (4) displays the program output on the terminal that the command was executed from.

To turn on External tracing *within a program*, use

```
XTRACE [tracing level]
                      ......
                      ......
```

Setting external trace level is identical to the *-t* option when starting Eloquence. The trace will be directed to stderr in both cases.

NOTE:

External tracing is available in either the development or run-time mode of Eloquence. Note that XTRACE or XTRACE -1 will disable external tracing. XTRACE 2 will have the same effect as eloqcore -t2 command.

Program Cross-Referencing

The purpose of program cross-referencing is to examine a program file and list where constants, line numbers, line labels, variables, functions, and subprograms appear. Cross-referencing of a program is done by using the list command with the -x option plus the options associated with -x. Note that the list command is executed from the HP-UX prompt. The syntax is as follows:

list -x [options]

Options associated with -x are l (labels and line numbers), c (constants), v (variables), and s (subprograms and functions). If no options are specified with -x, all of the options (l, c, v, and s) are set. In other words, the default is that a cross-reference be performed on all line labels, line numbers, constants, variables, subprograms, and functions.

Here is a short program named ADVERT. Following the program is the complete cross-reference table obtained by executing **list -x ADVERT**.

```
10       OPTION BASE 1
20       INTEGER Not_used        ! Variable declared, but not used
30       DIM Sign$(7)[15]        ! Constants used in declaratives
40       Sign$(1)="           "
50       Sign$(2)="           "
60       Sign$(3)="           "
70       Sign$(4)="           "
80       Sign$(5)="           "
90       Sign$(6)="           "
100      Sign$(7)="           "
110      DISP "  ",LIN(10)
120 Put_up_sign:                 ! Label not referenced
130      FOR Line=1 TO 7         ! Line not declared
140         DISP SPA(30);Sign$(Line)
150      NEXT Line
160      RESTORE
170 Again:                       !
180      READ Message$           ! Message$ not declared
190      CURSOR (31,13)
200      DISP "  "&Message$&"  "
210      WAIT 900
220      IF Message$[1,1]="D" THEN 240
230      GOTO Again
240 Last_sign:                   ! Another unreferenced label
250      WAIT 2000
260      CURSOR (29,13)
270      DISP "  "&"Buy Wonder-Shave Cream"&"  "
280      END
290  !
300     DATA "  Dina  ","  doesn't   ", "  treat him ","  right ..."
310      DATA "  but if  ","  he'd shave  ","Dina might!"
```

The cross reference listing is as follows:

```
segment: main
Symbol          Type  References
--------------- ----- ---------------------------------------------
```

```
----
1               CONST  220  220   40  130
10              CONST  110
13              CONST  190   260
2               CONST   50
2000            CONST  250
29              CONST  260
3               CONST   60
30              CONST  140
31              CONST  190
4               CONST   70
5               CONST   80
6               CONST   90
7               CONST  100   130
900             CONST  210
240             LINE   220
Again           LABEL  170   230
Last_sign       LABEL  240
Put_up_sign     LABEL  120
Message$        X      200   180   220
Sign$          X(1)    60   100   50   40   30   80   70   90   140
Line            R      140   150   130
Not_used        I       20
```

Integer numbers are listed first in the cross reference listing. These include numeric constants in declaratives, functions, etc. and referenced line numbers. The right-hand column lists all Eloquence links where each constant or line-number reference appears.

Next, all names are listed. The second column identifies each name type. Possible name types are listed in the table below.

Table 1 **Summary of Image Symbols**

Type	Description	Type	Description
LABEL	Line Labels	R	Real Precision Variable
SUBP	Subprogram Labels	S	Short Precision Variable
N FUN	Numeric Functions	I	Integer Precision Variable
$FUN	String Functions	X	String Variable
LINE	Line Numbers	D	Double Integer Precision Variable
CONST	Constants		

A number in parentheses following variable types R, S, I or X indicates the array's number of dimensions. The right-hand column lists all line numbers where each label appears.

The SCRATCH Statement

All or part of the user work area can be erased using a SCRATCH statement.

$$\text{SCRATCH}\begin{bmatrix} A \\ C \\ P \\ V \end{bmatrix}$$

Here is a summary of the SCRATCH statements:

SCRATCH Erases programs and variables.

SCRATCH A[LL] Erases the entire user area.

SCRATCH C Erases the values of all variables, including those in common (COM).

SCRATCH P Erases programs and variables.

SCRATCH V Erases the values of all variables except those in common. Do not use this statement in subprograms.

NOTE: If SCRATCH, SCRATCH A, or SCRATCH P is executed in a program,which run in run-time mode, the program and Eloquence will stop, thus returning control to the HP-UX environment.

The Reset Table in page 384 lists all conditions reset by SCRATCH statements.

Display Function Characters

The are two forms of special characters. They are obtained by selecting the appropriate special character set. (Consult your terminal manual for the method of switching character sets.)

- The first type of special character provides terminal display enhancements (inverse video, half bright, underline, and flashing). These will be covered fully in page 249 .
- The second type of special character is reached via the “DISPLAY FUNCTIONS” mode. When in this mode, control characters, such as cursor home or clear display, can be stored. (The keys corresponding to these characters can vary from terminal to terminal; refer to your terminal manual for further information.) If you list your program, all control characters are displayed as a tilde (~).

The Integrated Development Environment (IDE)

Program Development

The new Eloquence development Environment makes developing with Eloquence even easier and more convenient.

Major benefits are:

- You can have any number of text or program files active at one time, resticted only by memory limitations.
- The new Browse Toolwindow provides an entry for each editor window. For Eloquence programs it also provides the list of active segments. Double clicking on an entry brings the associated window on top and positions the cursor.
- Splittable editor windows and additional views make it convinient to operate at several places of the same document at one time without having to scroll around.
- Language sensitive editor.
- Integrated context sensitive online help.
- Includes its own client/server file sharing capabilities. This will make you independend of the availability of specific network file systems (NFS/SMB) and overcomes network topology and filesystem limitations.
- Consistent authorization scheme in a heterogeneous environment
- Supports the HP Roman-8 and ISO 8859-1 character sets.
- The editor is sensitive about line termination sequences.
- Integrated debugger

The Browser

All files loaded into an editor window have a corresponding entry in the Browse Toolwindow - it's symbolized by a folder symbol. When an editor window holds a Eloquence Program, you can open the folder (this is symbolized by the leading + symbol) and the list of program segments becomes visible.

Double clicking on the folder symbol will bring the associated client window on top. Double clicking on a segment name will additionally position the cursor at the beginning of that section. (This will even work if a program has been edited). Recompiling a program will re-createe its section list.

This is a real live saver if you ever had to jump between different segments (in even different files) while editing a program.

Compiling a program

When a program is compiled, all lines are checked for syntax errors (which will be displayed in the Output Toolwindow) and translated to the internal Eloquence code. Additionally, the segment list in the Browser Toolwindow is updated. If you double click on the error message in the Output Window, the cursor is automatically positioned on the offensive line.

Please Note: The compiler will reject to compile a program containing references to line numbers. Since the Development Environment does not deal with line numbers those programs would become useless.

Storing and loading program files

When you store a program file, it is compiled automatically. Only programs with no syntax erros can be stored (as a binary program file). To store a program file as text, simply change the file type in the save dialog. (Plase take care, that the file extension is something different than .PROG).

Loading a text file and converting it into a program document is easy. Just load the text file, open the Document Properties dialog and change the file type.

Debugging

The Eloquence Development Environment contains a simple yet powerfull debugger, you are probably already familiar with: The editor. When debugging a program, an arrow will indicate the current line in the source window, which is about to be executed. When the source code is not already present in a window, it will be requested from the eloqd or the eloqcore process as a last resort.

Eloquence makes it possible to debug your program in different environ ments:

- Execute your program on your local system
- Execute your program in a remote system

When initiating a local debugging session, Eloquence starts a local eloqcore process which is controlled and monitored by the Develop environment.

When initiating a remote debugging session, the Eloquence contacts the eloqd server at the remote system. The remote eloqd process will check your authorization, start the debug session and connect it to the Development Environment.

You normally don't want to execute the program in the editor window directly. It is usually part of a larger project and not self contained. Instead you create a Debug configuration which tells Eloquence how to invoke the start program of your project.

- The system where the debug process should be executed on
- The arguments to pass to eloqcore
- The environment to provide to your program

Please Note: You can only have one debugging session active at a time. Starting a debug session requires configuration of the Debug settings in the Application Properties.

NOTE: You can only have one debugging session active at a time. Starting a debug session requires configuration of the Debug settings in the Application Properties.

Initiating a debug session

The debug process is started by selecting the RUN/CONTINUE menu item from the Program menu pane, the associated accelerator key F5 or the button in the Program Toolbar window.

This way, the debug process will be executed. If any breakpoints are defined, it will be halted at the first breakpoint. If not, it will run until finished.

Alternatively, the debug process can be started by selecting one of the Step commands (Step into, Step over, Step out). In this case, execution of the debug process will be halted before executing the first line.

The "Run to cursor" command can also be used to start the debug process. In this case, a temporary breakpoint is set at the cursor location and then the debug process will be executed.

Whenever the debug process is suspended, the source file containing the next line to be executed will be automatically be loaded, unless it's already opened in a window. The current line is marked by a yellow arrow.

Stepping through a program

After the debug process has been suspended, you can continue it by one of the fol lowing actions:

- Continue. This will continue execution until the program exists or is halted (e.g. if a breakpoint is reached).
- Step into. This will continue the execution for one line. (This is similar to the former Halt key.)

- Step over. This will continue the execution until the current line has been finished. (This is similar to the former Advanced Step.) Please note, that recursive calls are taken care of. Execution will not be suspended in a different recursion level.
- Step out. This will continue execution until the end of the current segment has been reached.
- Run to cursor. This will set a temporary breakpoint at the cursor location and continue execution until the program exits or is halted. If another breakpoint is reached before the cursor line is executed, the debug process will stop at that breakpoint and remove the temporary breakpoint from the cursor location.

You can use the Halt operation to suspend the execution of a running debug pro cess after the end of the current line. When a Halt operation is pending, the cor responding button is displayed deperessed as a visible acknowledgement.

Breakpoint

A breakpoint causes the debug process to suspend execution, before the marked program line is executed. Breakpoints can be defined any program file, at any time. They are even remembered, when you exit the Development Environment.

Breakpoints are associated with the relative line in a program segment (when the file is loaded or stored), so they may be garbaged, when you edit a particular program file outside the Development Environment. You cannot set a breakpoint unless the program has been previously compiled.

A breakpoint becomes submitted to the debug process, when the program execution has been suspended. If you add or delete a breakpoint, while the debug process is currently executing, you need to suspend execution (for example, by selecting Halt) to activate the changes.

The Breakpoints dialog (available in the Program menu pane or by pressing the accelerator key Shift-F9) can be used to review or remove breakpints without having the associated file in an editor window. When double-clicking the first column, the corresponding file is loaded automatically and the cursor is positioned in the corresponding line.

The Output Toolwindow

Currently by default, each variable modification causes a trace message sent to the Debug area of the Output Window. This can be disabled by selecting the Trace menu item in the Program menu pane.

The Call Stack Toolwindow

The Call Stack Toolwindow visualizes the call chain, how the current line has been invoked. Double-clicking an entry positions the cursor on the corresponding source line.

The Variables Toolwindow

The Variables Toolwindow contains the active variables for this segment along with the current value. Array variables do not display the value of individual elements by default. They are marked by a leading + sign. To display array elements, you can expand the array by clicking on the leading + sign. Please note, that values are truncated at a certain length. This is indicated by a trailing "...". In order to change a variable value, just double-click it, type the new variable value in the edit box and press return.

Debugging a character oriented program

You can use the Eloquence Development Environement to debug a character oriented program (a program using FORMs and softkeys). This requires a UNIX server as character oriented programs are only supported in a UNIX environment.

In addition to a server, you need either a terminal or a terminal emulator (such as Reflection). After the debug process is started, the input and output is redirected to the terminal.

In order to debug a character oriented program, you must configure the TTY device in the Application Properties accordingly. To obtain the tty devicefile name, you should follow the procedure below:

- If you use a terminal device connected to a serial device, simply enter the tty device file (for example /dev/tty0p1). You should not run a getty on this port and you must configure it properly before using it (setting baud rate, data format etc.).
- \- or -
  1. Login to your server system
  2. Obtain the tty device using the tty command. The device file is typically similar to /dev/ttyp4.
  3. Disable the shell input (for example by executing a sleep 10000)
  4. Enter the device file in the configuration.

Please be aware, that you must define a TERM environment variable, describing the type of terminal used. After that, start the debug process as usual.

(Yes, I know - this is complicated. We are looking for a way to make this easier and even automatic.)

4

Data Variables and Data handling

Eloquence programs are used to process data for a great variety of applications, such as company payrolls, order processing, budgeting, and accounting. Each application has its own requirements for the data it uses. Some programs work with numeric data while others use only alphabetic. Some programs perform calculations requiring many digits of precision while others need less accuracy.

Using the versatile Eloquence programming language, you can specify both the data types and the required precision (or you may simply let the program default to the standard data types).

The following data-handling statements are introduced in this chapter:

STANDARD, FIXED and FLOAT Set the format for displayed and printed numbers.
LET Assigns values to variables.
COM and DIM Reserve memory space for variables to be used and specify bounds for arrays.
REDIM Changes the working bounds for an existing array.
DINTEGER, INTEGER, SHORT and REAL Reserve memory space, specify bounds for arrays and specify precision for selected variables.
TYPE Specify user defined types.
OPTION BASE Sets the default lower bound for arrays.
DATA Specify a list of data values (DATA) and
READ assign the values to variables in a program (READ).
RESTORE RESTORE resets a data pointer to the start of the Data list.
INPUT Requests and accepts data input from the keyboard.
LINPUT Requests and assigns data to a string variable from the keyboard.
ENTER and LENTER Input data from the display and assign values to variables without suspending execution.
ACCEPT Accepts data input from the keyboard without displaying it on the screen.
EDIT Allows changing the current value of a string variable from the keyboard.
XPACK/XUNPACK This statements provide a convenient way to transfer string and numeric data to and from a string variable.

Types and Forms of Variables

A variable describes a location in memory in which values can be stored. Computer languages use variable names to represent these locations. Then each time that variable name is quoted, the computer looks up the corresponding memory location and finds the value. The value contained within a variable may be altered, hence the name “variable”. Each variable is of one type and holds a value of that type. For example:

```
A = 2 * B
```

Here A and B are variables. The number 2, of course, cannot be altered.

There are two main types of variables available with Eloquence—numeric and string. Numeric variables hold numbers, both positive and negative, integer or fractional. Numeric variables are themselves split into three types—INTEGER, SHORT and REAL. Numeric variables will be fully covered later in the chapter, but briefly:

- INTEGER & DINTEGER variables hold integers; that is numbers without fractional parts.
- REAL variables hold numbers, with or without fractional parts, to the maximum precision.
- SHORT variables also hold numbers with or without fractional parts, but the precision is less exact.
- TYPE is the keyword to define user defined type.

A string variable can hold any sequence of ASCII characters. ASCII is the acronym for American Standard Code for Information Interchange. It is a standard way of representing characters and printing commands within a computer. A full list of ASCII characters is given in see chapter , ASCII Character Codes, in Appendix A. Note that an ASCII string is capable of holding all the keyboard type characters (including the blank character) and ASCII non-printing characters.

A feature of Eloquence is that strings may also include alternate character sets and display enhancements. These include such useful tools as line drawing characters and inverse video displays and are described fully in see chapter , Display Enhancement Codes/Character Set Switching Codes, in Appendix A. Use of alternate character sets will entail some added space overhead, as control bytes are added to the string.

A string may hold the characters 0 through 9. *These are characters, not numbers*. Arithmetic calculations cannot be performed on digits that are part of strings. Typical strings that contain digits not used in arithmetic calculations are strings holding addresses, part numbers, dates, and department codes.

Each type of variable, except the User Defined Types which can be of type simple only, can be declared in one of two forms—simple (non-subscripted) or array (subscripted). Each simple variable holds either one number (simple numeric variable) or a string of characters (simple string variable). An array variable is a collection of data items of the same type having from one to six dimensions. It is a convenient tool for handling large groups of data within a program.

****

The Eloquence language supports user defined data types. A user defined data type consists of a list of variables, called member variables. When a type is derived it inherits all properties (in this case, the member variables) of the base type. When you derive a type from a base type, you can use the derived type to call functions or subprograms which accept the base type.

Type definitions are inclosed in the TYPE .. END TYPE keywords. All variable declarations between will be part of the new data type.

TYPE TypeName [EXTENDS BaseTypeName]   END TYPE

The example below defines the data type Tphone, containing the member variables Id, Name$ and Phone$.

TYPE Tphone   INTEGER Id   DIM Name$[30],Phone$[20]   END TYPE

The example below defines the data type Tphone2. Because it is derived from the type Tphone, it includes all member variables of the base type. It defines the new member variable Comment$.

TYPE Tphone2 EXTENDS Tphone   DIM Comment$[40]   END TYPE

There are different scopes (lifetimes) for type definitions:

If a type is defined globally (in the main program), it is available to all subprograms and functions. In addition, it will be passed to a program which is LOADed from the initial program. This is similar to COM variables. If defined in a SUB/FN program segment, a type definition is only known inside the subprogram. It will be deleted when the segment returns.

****

Variable Names

Every variable name must abide by the following rules. They apply equally to both simple and array variables.

- A variable name is from 1 to 15 characters (see string variables below).
- The first character must be an uppercase (capital) letter.
- The remaining characters must be lowercase letters, digits, or the underscore character (_).
- Each string variable name must end with a dollar sign ($). Note that this $ is not counted as one of the possible 15 characters; therefore, string variable names may be up to 16 characters long, if the compulsory $ suffix is included.
- Variable names must be unique.

Some examples of legal numeric, string and user defined variable names are as follows:

Numeric variable names
X
Data1
Order_no

String variable names
A$
Password$
Name_address$
City_state_zip$

User defined variable names
Customer.name

As a general rule, a variable name should be indicative of its contents. Use of the underscore character (_) will make the name more comprehensible.

Keyword names are always in upper-case. Thus a legal variable could have the same combination of letters as a keyword, as long as only the first letter of the variable name was uppercase. This feature is affected by Space Dependency; therefore, to ease confusion, it is suggested that variable names and keywords *not* be spelled identically.

String Variables

A string is a series of ASCII characters which can be stored in a string variable. (In Eloquence a string may also hold display enhancement and line drawing characters.) A string variable can be declared in a DIM or COM statement, which specifies the maximum length of the string. The maximum length of a string is the maximum number of characters it can hold. If a string variable is used without first being specified in a DIM or COM statement, it is implicitly dimensioned to be 18 characters maximum. The current length of a string refers to the number of characters currently contained within it.

Each string variable name is terminated with a dollar sign ($). For example:

```
X$

Home_address$

Part_no1$
```

Characters can be assigned to a string (or substring) variable using the LET statement:

[LET] string (or substring) variable [=string variable ...]=string expression

For example:

```
LET Title$="Chapter 1"

Asterisks$="*********"

Married_name$=Husbands_name$="Smith"
```

There are many other ways to assign values, as shown later in this chapter.

String Arrays

A string array is equivalent to a numeric array except that its elements are strings. String array names follow the rules for string variable names. String arrays are dimensioned in a DIM or COM statement. Every string in the array has the same maximum length. Like a numeric array, a string array can be implicitly dimensioned. In all string operations, an element of a string array can be used like a simple string variable. The element is accessed in the same way as a numeric array—by quoting the array name and the subscript(s).

For example, the following statement prints the string contained in the quoted element of the three-dimensional string array Name_array$:

```
PRINT Name_array$(1,3,4)
```

String Array Defaults

Explicitly dimensioning a string array conserves memory if the amount of space required is less than the default (10 elements per subscript). As each default element contains an 18 character string, a lot of space may be wasted. Always dimension string arrays explicitly.

String Expressions

Text within quotes (a literal) is the simplest form of a string. Any ASCII character can appear within the quotes. If you want to print quotation marks themselves you must use string concatenation and the CHR$ string function. This is because the quotes are used as the literal delimiter. This will be dealt with fully in the next chapter. String expressions may contain any of the following:

- Text within quotes.
- String variable names.
- Substrings.
- String concatenation operations.
- Built-in string functions.
- User-defined string functions.

As with numeric expressions, a string expression can be enclosed in parentheses if necessary.

Substrings and string concatenation are covered in this chapter. The next chapter covers the built-in string functions and shows how to define your own string functions.

Substrings

A substring is a portion of a string rather than the entire string. Normally, a reference to a string variable refers to the entire string. For instance, if Example_string$ = "ABCDEF" and the statement LET B$ = Example_string$ is executed, B$ = "ABCDEF". However, sometimes it is necessary to reference only a portion of a string. Suppose B$ is to be set equal only to the last three characters of Example_string$. This can be done using the substring designator. Again, assume that Example_string$ = "ABCDEF". This time execute the statement LET B$ = Example_string$[4,6]. The result is now B$ = "DEF", not "ABCDEF".

There are three ways to designate a substring. The first method is to indicate the starting index (position) of the substring, followed by a comma and the ending index of the substring. All indexes are inclusive. For example, if Example_string$ = "ABCDEFGHIJ", then:

```
Example string$[1,3] = ”ABC”

Example_string$[4,6] = ”DEF”

Example_string$[7,10] = ”GHIJ”

Example_string$[1,10] = ”ABCDEFGHIJ”
```

Notice that Example_string$[1,10] is effectively the same as Example_string$.

The second method of designating a substring is to give the position of the first character, followed by a semicolon and the length of the substring. Continuing the above example:

```
Example_string$[1;3] = ”ABC”

Example_string$[4;3] = ”DEF”

Example_string$[7;4] = ”GHIJ”

Example_string$[1;10] = ”ABCDEFGHIJ”
```

Here too, Example_string$[1;10] is effectively the same as Example_string$.

The third method of designating a substring is to give the starting index of the substring only. This is really a special case of the first method, in which the ending index is assumed to be the length of the string. Continuing the example:

```
Example_string$[1] = ”ABCDEFGHIJ”

Example_string$[5] = ”EFGHIJ”

Example_string$[10] = ”J”
```

So Example_string$[1] is effectively the same as Example_string$.

Regardless of which method is used, the first position indicator, the second position indicator, and the length indicator may be any valid numeric expression.

NOTE: If the numeric expression is not a whole number, it will be rounded to the nearest integer.

Here are more examples:

```
Example_string$[N,M]

Example_string$[F;L]

Example_string$[N + SQR(Z);L−1] ! so long as  Z > 0!

Example_string$[A(I),A(J)]
```

The NULL String

The null string is a string without any characters at all, not even blanks or non-printing characters. It is obtained by entering two sets of quotes ("") or giving a substring length of 0. For example:

```
A$=Example_string$[1;0] ! Set A$ to the NULL string
```

```
Test$="A null"&A$&"string" !There will be no space between null and
string
```

```
A nullstring
```

There is a special case, however, with a null substring. When a null substring is assigned to a string then the characters covered by that substring become blanks. The string length is not altered. For example, using our previous test:

```
Example_string$="ABCDEFGHIJ"
Example_string$[3;2]=""!Assign a null string to characters 3 and 4
```

```
AB  EFGHIJ
```

Characters 3 and 4 are changed to blanks and Example_string$ is still 10 characters long.

All strings are initialized to the null string by executing RUN, SCRATCH C, or SCRATCH V.

String Concatenation

The ampersand sign (&) is the string concatenation operator. It joins strings, substrings, and string expressions. No blanks are inserted between strings. Note the following example:

```
20 The$="the"
30 Example$="I am "&The$&" "&CHR$(38)&" sign."
40 PRINT Example$
I am the & sign
```

Examples of String Use

The following example shows substrings in use. Substrings are frequently used to insert or change characters in a string without affecting the rest of that string. Suppose a university wished to print students' results. Every letter heading would be the same; the only change would be in the faculty name.

```
10   DIM Faculty_title$[22]
```

The string Faculty_title$ has maximum length of 22 characters.

```
20   Faculty_title$="School of "    Initially, Factualty_title is a
30   LET B$="Business"            10 character string "School of_"
50   En$="Engineering"               ( _ indicates a blank space)
60   L$="Law"
70   Bw$="Basket Weaving"
80   C$="Civil"
100  Faculty_title$[11]=B$
```

The string B$ is inserted into the string Faculty_title$ beginning at the 11th character of Faculty_title$.

```
130  PRINT Faculty_title$              Prints: School of Business
170  Faculty_title$[11]=En$

210  PRINT Faculty_title$              Prints: School of Business
```

If single subscripts are used when assigning substrings (as in the examples above) and the item being assigned is too short to fill the string, then the rest of the receiving string is truncated. Thus, continuing the example, when the string L$ is inserted at character 11 in Faculty_title$, the rest of Faculty_title$ is erased:

```
250  Faculty_title$[11]=L$
290  PRINT Faculty_title$              Prints: School of Law
```

If single subscripts are used and the item being assigned is too long for the substring, an error is returned:

```
420  Faculty_title$[11]=Bw$
        ERROR 18 IN LINE 420
```

An attempt to insert the 14 chars of Bw$ in the 11 remaining chars of Faculty_title$ returns ERROR 18.

If two subscripts are used when assigning substrings, then the changes will only be made in the range of the receiving string specified by the subscripts. The length of the receiving string is retained and any receiving string characters outside the subscript range will be unaffected. To continue the example:

```
450 Faculty_title$[11,22]=B$L$
460 PRINT Faculty_title$            Prints: School of BusinessLaw
```

Note that the concatenation operator (&) does not insert a blank. A better title would be printed by:

```
500 Faculty_title$[11,22]=B$&" "L$
510 PRINT Faculty_title$              Prints: School of Business Law
```

Now for the Civil Law faculty. Note that if the substring to be added is too short, the rest of the receiving string, within the range specified by the two subscripts, is filled with blanks:

```
450  Faculty_title$[11,18]=C$
460  PRINT Faculty_title$             Prints: School of Civil____Law
```

If the substring to be added is too long, it is truncated to fit in the length specified by the subscripts—no error is indicated.

```
590 Faculty_title$[11;12]=Bw$
600 PRINT Faculty_title$              Prints: School of Basket Weavi
```

Here is a summary of substring errors and their error codes:

Table 2 **Columns**

Invalid Designator	Action
Starting index < 1	Error 18
Ending index < 0	Error 18
Starting index > ending index + 1	Error 18
Starting index = ending index + 1	Null string
String length = 0	Null string
Starting index > current length of string + 1	Error 18
Ending index > dimensioned length of string	Error 18

Numeric Variables

Real Variable Numeric Ranges

The range of numeric values which can be entered or stored lies between –9.99999999999 x 10^{125} through –1 x 10^{-130}, 0, and 1 x 10^{-130} through 9.99999999999 x 10^{125}. The range of intermediate calculations is the same as above, but 16 digits.

Integer Variable Numeric Range

Integer and dinteger variables hold values between -2^{31} to $+2^{31}-1$. All intermediate calculations are made using the full precision above.

Numeric Precisions

The precision of a number is a function of its size. The greater the number of digits, the greater the accuracy that the variable is able to record. Of course, the greater the number of digits, the larger the variable and the more storage space it needs. To give you the maximum choice between space and accuracy, Eloquence numeric variables can be stored in any of three forms—INTEGER, SHORT, or REAL (full) precision. The forms used in a program affect the speed of execution, the precision of the results, and the amount of space needed to store the values.

If you do not specify the form, it defaults to REAL (full) precision. Real precision variables are allotted twelve significant digits of precision. They are the most accurate form of holding numeric data but take up the most space.

The following list describes the differing characteristics of the various forms of Eloquence numeric variables. Use the form most appropriate for your own application. Once you have decided, use the declaration statements to specify the format in each program.

- REAL precision variables hold whole or fractional numbers. They are represented internally with a mantissa of 12 significant digits and an exponent in the range from −130 through 125. Although real values offer much greater precision than short values, they occupy twice as much storage space. Note that short values are included in the range of real values. This form is the default. All other numeric variable forms must be explicitly defined.
- SHORT precision variables also hold whole or fractional numbers. They are represented internally with a mantissa of six significant digits and an exponent in the range from −130 through 125.

- INTEGER precision variables hold whole numbers only (no fractional part). Integer-precision numbers range from -2^{31} to $+2^{31}-1$. They are held in binary 2's complement form, (not exponent and mantissa).
- DINTEGER precision variables hold whole numbers only (no fractional part). Integer-precision numbers range from -2^{31} to $+2^{31}-1$. They are held in binary 2's complement form, (not exponent and mantissa).

All numbers are REAL (full) precision unless otherwise defined using a SHORT, INTEGER or DINTEGER statement as shown later. Twelve digits of precision is the maximum. Numbers entered with more than twelve digits will be truncated. In other words, any digits after the twelfth will be ignored. Note that this removes only the least significant digits. For example, entering 1234.5678912365 as the value of a real variable will store 1234.56789124. The same entry, however, will be rounded to 1234.57 in a short variable or 1235 in an integer variable.

NOTE: Confusion can arise between the maximum size of a numeric variable, and its maximum precision. The maximum size of a numeric variable is the largest number it is able to store and is a function of the size of its exponent. The precision shows the accuracy to which the variable is held and is a function of the mantissa size. Thus a real variable may hold a number as large as 9.0×10^{99}, much greater than twelve digits in decimal form, but its accuracy is still that of the mantissa.

NOTE: When an INTEGER is loaded from a program into memory is created via a calculation, it takes up to four bytes of main memory, whereas a DINTEGER takes up the same space on memory as on disk. When the INTEGER is stored it is put into two bytes of physical disk space. The DINTEGER remains 4 bytes long. So, if the INTEGER stored, does in fact take more than two bytes of main memory, it will not fit when stored and error 20 ("integer overflow") occurs. To get around this error either store the number in REAL precision, DINTEGER precision, or decrease the size of the INTEGER causing the problem.

Numeric Display Formats

Three formats are available for displaying and printing numbers—standard, fixed point, and floating point (scientific notation). Standard format is the default; it is automatically set when the machine is switched on or reset. The display format can be changed to fixed or floating point by using the FIXED and FLOAT statements. The STANDARD statement returns the machine to standard format.

Unless IMAGE and PRINT USING statements are used to precisely control the form of output, all numbers are output with a trailing blank and either a leading blank or a minus sign, (if the number output is negative). For more information concerningIMAGE and PRINT USING see "The PRINT Statement" on page 278.

Standard Format

The default standard format is convenient for most displays. It is the form commonly used when writing numbers. Standard format is set at power on, RUN, and SCRATCH A. To reset standard format after a FIXED or FLOAT statement has been executed, execute the STANDARD statement:

STANDARD

In standard format, the most significant twelve digits of a number are output. For example, 9876543210.12345 is output as 9876543210.12. Leading and trailing zeros are suppressed. For example, 000032.100000 is output as 32.1.

Any number whose absolute value lies between 1 and 10 is output in fixed format showing all significant digits. Numbers between –1 and 1 are also output in fixed format if they can be represented precisely in 12 or fewer digits to the right of the decimal point. All other numbers are output in scientific notation. The form is the same as FLOAT 11. see “Floating Point Format” on page 87.

Here are a few examples of standard output:

Number	Standard Output
15.00	15
.23500	.235
.0547^9	4.38415537301E–12
00987	987
10000^6	1.00000000000E+24

Fixed Point Format

Fixed format is similar to standard, except with fixed format you can specify the number of digits to appear to the right of the decimal point. If the number output has fewer digits than that specified, trailing zeros are inserted (leading zeros are still suppressed). If the number output has more digits than specified in the FIXED statement then the number is rounded. The following statement sets fixed point format:

FIXED number of digits

The parameter *number of digits* is a numeric expression, rounded to an integer, which specifies the number of digits to the right of the decimal point. Its range is from 0 to 12.

NOTE: Even if the number displayed is an integer, fixed notation adds a decimal point and the required number of trailing zeros. Similarly FIXED 12 outputs 12 places of decimals for a SHORT variable. As such a variable only extends to 6 places of decimals. The last 6 places for a short variable in FIXED 12 display are always zeros. The internal accuracy is never affected.

Here are some numbers and their FIXED 4 output form:

Number	FIXED 4 Output
18	18.0000
−.000006	−.0000
−2.75327	−2.7533
5.3111E4	53111.0000
1234567891234.5	1.23456789123E+12

When fixed point is set and the absolute value number to be output is >= 1E12 or has more than 17 digits, the format temporarily reverts to floating point. For example, in FIXED 12, 100000 is output as 1E+05.

Floating Point Format

When working with very large or very small numbers, the floating point format is most convenient. This scientific notation outputs a number as a mantissa in the range >1 and <10, followed by an exponent expressed as a power of 10. Syntax of the FLOAT statement is as follows:

FLOAT number of digits

The parameter *number of digits* is a numeric expression, rounded to an integer, which specifies the number of digits of precision for the mantissa (the number of digits to the right of the mantissa's decimal point). It ranges from 0 to 11.

A number output in floating point format has the form:

```
±d.d  ...  dE±dd
Mantissa   Exponent
```

- If the number is negative, a minus sign will precede the mantissa; if the number is positive or zero, a space precedes it.
- A decimal point follows the first digit, except in FLOAT 0.
- Digits may follow the decimal point; the number of digits after the decimal point is set

using the *number of digits* parameter following the FLOAT statement.

- The character E is followed by a plus sign or minus sign and two digits. This is the exponent, and it represents the power of 10 by which the mantissa should be multiplied in order to express the number in standard format. *A negative exponent does not mean a negative number, it means that the number is <1.*

Here are some examples of FLOAT 2 format:

Number	FLOAT 2 Output
–3.2	–3.20E+00
271	2.71E+02
26.377	2.64E+01
.000004	4.00E–06
2.4E78	2.40E+78

Rounding

A number is rounded before being displayed or printed if there are more digits to the right of the decimal point than the numeric display format allows. The rounding is performed as follows: The first excess digit on the right is checked. If its value is 5 or greater, the digit to the left is incremented by one; otherwise it is unchanged.

Whenever a number is rounded as a result of a numeric display format, it is only the display which is affected. Internally, the number remains as accurate as the variable holding it will allow. For example, execute the lines below:

```
A=1.235
1.235
FIXED 2
A
1.24
FIXED 3
A
1.235
```

The value of A does not alter, only its display.

Simple Numeric Variables

Any simple numeric variable can be assigned a value using the LET statement. Syntax for this statement is as follows:

[LET] simple variable$_1$ [=simple variable$_2$ . . .]=numeric expression

Notice the keyword LET is optional. For example, each of the following statements assigns the value 12 to X:

```
X=12
```

```
LET X = 12
```

```
X=SQR(144)
```

As another example, the following assigns the value 12 to Y and X:

```
X=Y=3*4
```

All variables are set to zero when created or when a program is run. They remain as zero until a value has been assigned to them.

To check the current value of a variable, type in its name, then press RETURN.

The values of simple variables are erased by executing a SCRATCH statement, as shown in page 25 .

Types and Values

Eloquence is not a strongly typed language. Thus a numeric variable of one type (for example, INTEGER) may be assigned a value from a variable of another (for example, REAL). The value will be converted to the receiving variable's type when transferred. The following example explains further:

```
10 A=1.2345678 ! REAL A implicitly defined as REAL is default
20 SHORT B
30 INTEGER C
40 B=A
50 C=A
60 PRINT "A is ";A;" B is ";B;" C is ";C
70 END
RUN
A is  1.2345678 B is  1.23457 C is  1
```

In this example it is not just the display of the value that is altered. The value is permanently changed in the receiving variable. No error will be indicated in these transfers, but any attempt to assign a string value to a numeric variable (or vice versa) will not be accepted.

Numeric Arrays

An array is a structured group of data items, all of the same type. Arrays are very useful when manipulating large amounts of associated data, as an array is stored or retrieved as a unit. (When an individual item is required from within an array it may be accessed uniquely.) *The individual data items in arrays are called array elements, they have all the properties of simple variables of their type*.

An array may have from one to six dimensions. Examples of one and two dimensional arrays are shown below. Arrays of three or more dimensions are not so conveniently represented on paper, but they can be set up and manipulated easily in a program.

A one-dimensional (1*10) element array—ten elements in all:

1.5	2.3	3.4	4.7	10.7	.8	3.5	4.6	2.0	1.1

A two-dimensional (3*4) element array—12 elements in all:

12.95	12.95	11.50	11.50
3.95	3.50	3.50	3.50
.80	.80	.80	.80

Subscripts and Array Size

Every numeric array must be explicitly defined in a variable declarative statement; these are described later in this chapter. There, its size is specified by numbers in parentheses after its name. These numbers are known as subscripts, and their presence tells the Eloquence interpreter to assign dimensions to the array—6 dimensions maximum. Thus a maximum of six numbers, separated by commas, will be accepted. The size of the number tells the interpreter how large the dimension will be. Assigning a size to an array is known as dimensioning an array. The number of elements in an array is the product of the number of elements in each dimension. The above two-dimensional array has one dimension of 3 elements and the other of 4 elements. Their product, 12, gives the number of elements in the array.

An element is accessed by quoting its position in the array. This is done by quoting first the array name and then the subscript(s) which point to the element's position in the array. The number of subscripts which need to be quoted is the same as the number of dimensions. In a two dimensional array there will be two subscripts. It may help, to think of them as coordinates.

Subscripts are integer expressions separated by commas and enclosed in parentheses. (Any non-integer numeric value quoted as a subscript will be rounded to the nearest integer.) If a subscript is outside the range defined for an array (either too large or too small), then **`ERROR 17`** is returned. The range of each subscript is –32767 through 32767, but the size of an array is limited by memory. The size of an array depends on both upper and lower subscript bounds.

For example, an array M dimensioned as M(2,3) is an array with two dimensions having upper bounds of 2 and 3. If the lower bound for each dimension is 0 (the default value), the array has a total of 12 elements. (As the lower bound for each dimension is 0, the total number of elements is "0..2" in the first dimension, and "0..3" in the second. There are thus 3*4 possible combinations of subscript or 12 elements in all.)

Table 3

Here is a representation of array M:

Columns	0	1	2	3
Row 0	(0,0)	(0,1)	(0,2)	(0,3)
Row 1	(1,0)	(1,1)	(1,2)	(1,3)
Row 2	(2,0)	(2,1)	(2,2)	(2,3)

The use of columns and rows is to assist in explaining the concept of arrays, and how an element is accessed. Arrays are not held as grids within the computer, but the principle is the same.

The OPTION BASE statement is used to change the default lower bound, but the lower bound may always be defined for any array dimension by using double subscripts. The array M could also be dimensioned M(–1:1,–2:1). The upper and lower bounds are separated by a colon. Note that the number of elements is still the same; the size of array M has not altered.

Table 4

The subscripts for array M would now be the following:

Columns	–2	–1	0	1
Row -1	(–1,–2)	(–1,–1)	(–1,0)	(–1,1)
Row 0	(0,–2)	(0,–1)	(0,0)	(0,1)
Row 1	(1,–2)	(1,–1)	(1,0)	(1,1)

Array Elements

Every array element must be of the same type; therefore, you cannot mix REAL and INTEGER elements in one array. As each element in the array is referenced by using subscripts (and can be used like a simple variable), M(1,0) refers to an element in array M which may be assigned a value and used in calculations and other programming operations. For example:

```
M(1,0) = 10
```

```
A = M(1,0)/7
```

```
PRINT M(1,0)
```

Since a subscript is a numeric expression evaluating to an integer, numeric expressions and variables may be used as subscripts. This allows powerful programming constructs:

```
30 FOR I = −2 TO 1
40 READ M(1,I)
50 NEXT I
```

It is thus much easier to loop through the elements of an array, when loading large amounts of data, than to use multiple INPUT statements. The larger the array, the greater the time saved.

All elements of an array can be specified collectively in an input or output operation by using the array identifier. For example:

```
PRINT A(*)
```

prints the entire array A. The MAT PRINT statement is also available to print arrays, as shown in page 297 .

The maximum size of an array is specified in a DIM, COM, REAL, SHORT, INTEGER or DINTEGER statement, as shown later in this chapter. If an array is used without being explicitly defined, the default upper bound of the array is set to 10. The lower bound is either 0 or 1, depending on the OPTION BASE setting. To minimize memory waste and make your programs clearer, it is recommended that you define arrays and other variables explicitly at the start of a program.

User defined Types

The Eloquence language supports user defined data types. A user defined data type consists of a list of variables, called member variables. When a type is derived it inherits all properties (in this case, the member variables) of the base type. When you derive a type from a base type, you can use the derived type to call functions or subprograms which accept the base type.

Type definition

Type definitions are enclosed in the TYPE .. END TYPE keywords. All variable declarations between will be part of the new data type. When the EXTENDS keyword is present, the new data type is derived from the given base type.

Syntax:

TYPE *type_name* [EXTENDS base*_type_name*]
.
.
.
END TYPE

They are different to the string and numeric variable types known in Eloquence, because they have to be defined before they can be used. The standard types are internally defined. The type definition can be compared with a design, as example for a vehicle. With the type definition, it is defined on which parts the vehicle consists. This vehicle has an engine_type, a number of wheels, a colour and so on.

Example:

```
TYPE Tvehicle
   DIM Engine$[1]
   INTEGER Wheels
   DIM Colour$[20]
   .
   .
   .
END TYPE
```

In a TYPE constructions only DIM, INTEGER, DINTEGER, SHORT, REAL and Comment lines are allowed.

Exporting Types

Type definitions can be "exported" to lower calling levels. This is done with the new EXPORT TYPE keyword:

TYPE *Tname* [EXTENDS *Tbase*]

EXPORT TYPE *Tname* [EXTENDS *Tbase*]

IN DATA SET ... DEFINE TYPE *Tname* [EXTENDS *Tbase*]

IN DATA SET ... EXPORT TYPE *Tname* [EXTENDS *Tbase*]

By default a type definition is local to the current segment. This is also true for types defined in the main segment. Therefore, types used in COM statements must be "exported" in order to be usable in a subprogram. Otherwise a runtime error message will be issued.

NOTE: This is a behaviour change.Initially, all types defined in the main segement were global.

Type scope

A type definition is local by default and is only visible within the local segment. When a type is "exported" it becomes visible to subsequently called levels. When a segment defines a type with a name which has been exported from a higher level, it will temporarily replace the exported type in the current function or subroutine.

When a segment defines and EXPORTs a type with a name which has been exported from a higher level, the newly exported type permanently replaces the former type for all subsequently called lower level segments. If the type which is most recently defined is not EXPORTed, the previously exported type again becomes available to lower called segments.

Derived Types

It is very helpful to create types, which are a superset of an existing type. They are derived from an existing type, called Base-Type. It inherits all the members of the Base-Type and have individuell members added.

In the above example derived types as car, track or motorcycle from the Base-Type Tvehicle are possible. All of them have inherited the members of Tvehicle and some individuell members.

Example:

```
TYPE Car EXTENDS Tvehicle
```

```
   DIM Engine_type$[30]
   INTEGER Power, Persons
END TYPE
```

A new type can be derived from *car* again and all members are inherited, the one from the Base-Type *Vehicle* and the members from *car*.

Please note, that a base type can be defined after a derived type. It must be defined however before a derived type is instantiated.

Nested types are not supported, this means that it is not possible to define a user defined type as a member of a user defined type.

Type instantiation

Since a type definition is merely a blueprint rather than a real object, it must be instantiated before it can be used. This can either be done before execution by using the COM and DIM statements or at runtime with the NEW statement.

DIM and COM statements:

```
COM Instance_name:Type_name
COM Instance_name AS Type_name
DIM Instance_name:Type_name
DIM Instance_name AS Type_name
```

The NEW statement:

```
NEW Instance_name:Type_name
NEW Instance_name AS Type_name
```

The Instance_name is the variable name and the Type_name is the name of the data type. The instance name and data type name are either separated by a colon or the keyword **AS**.

Example

```
DIM Vehicle AS Tvehicle
or
DIM Vehicle:Tvehicle
```

In addition, the NEW STRUCT statement can be used to create a new, identical copy of the referenced object.

NEW STRUCT Instance_name=Instance_Name

The example below creates a new object named *Clone*. It will be an exact copy of the referenced object *Entry*.

```
DIM Entry:Tphone
Entry.Name$="Joe Sample"
Entry.Phone$="(202) 243 1440"
NEW STRUCT Clone=Entry
```

Using member variables

Member variables can be used like any other variable. A member variable is specified by giving the variable name (instance name) and the name of the member variable, separated by a dot.

```
Vehicle.colour$="red"
PRINT Vehicle.colour$
```

In addition to accessing single variables, you can specify the whole object at once. The example below prints all member variables of Entry.

```
PRINT STRUCT Vehicle
```

The STRUCT statement can also be used to copy the value of an object:

```
STRUCT A=B
```

When copying an object to another, both must be compatible.

- Both objects must have the same data type. In this case, all member variables are copied.
- Both objects must have a common base type. In this case, only the common mem ber variables are copied.

The STRUCT keyword

The STRUCT keyword can be used with some statements to operate on the whole object instead of

a single member variable. This is similar to the Array(*) notation in Eloquence which causes

an operation on the whole array instead of a single element.

The following statements can be used with STRUCT:

- PRINT
- READ

- PRINT #, READ #
- IN DATA SET USE, IN DATA SET LIST
- PACKFMT
- XPACK, XUNPACK

Runtime type identification

The TYPEOF$ function can be used to identify an instance.

X$=TYPEOF$(instance_name)

This returns the type name of the given instance.

The IS A operator can be used to categorize an instance.

IF instance_name IS A type_name THEN ...

If the instance is either of the specified type or derived from it, the IS A operator returns nonzero.

For example:

```
   TYPE Tbase
      INTEGER A
   END TYPE
   TYPE Tderived EXTENDS Tbase
      INTEGER Q
   END TYPE
!
   DIM Derived:Tderived,Base:Tbase
   DISP "Base is of type ";TYPEOF$(Base)
   DISP "Inst is of type ";TYPEOF$(Derived)
   DISP "Base    is a Tbase    =";Base IS A Tbase
   DISP "Base    is a Tderived =";Base IS A Tderived
   DISP "Derived is a Tbase    =";Derived IS A Tbase
   DISP "Derived is a Tderived =";Derived IS A Tderived
   STOP
```

Data base integration

The user defined type concept is designed to operate with the Eloquence data base. The IN DATA SET ... DEFINE TYPE statement can be used to define data types from the data base schema at runtime, the PACKFMT, IN DATA SET LIST

and IN DATA SET ... USE have been enhanced to support user defined data types. For more information about this statement, see chapter , The IN DATA SET Statement, in the Data Base Manual.

For example:

```
DBOPEN(Db$,"",1,S(*))
   ...
   IN DATA SET "CUSTOMER" DEFINE TYPE Tcust
   NEW Cust:Tcust
   IN DATA SET "CUSTOMER" USE STRUCT Cust
   ...
   DBGET(Db$,"CUSTOMER",7,S(*),"@",Buf$,Key$)
   ...
```

Of course, types can also be defined statically in your program:

```
  TYPE Tcust
    DIM No$[6]
    DIM Name$[30]
    ...
  END TYPE
  DIM Cust:Tcust
!
  DBOPEN(Db$,"",1,S(*))
  ...
  IN DATA SET "CUSTOMER" USE STRUCT Cust
  ...
  DBGET(Db$,"CUSTOMER",7,S(*),"@",Buf$,Key$)
  ...
```

Error Messages

The following runtime errors are used with types:

ERRN	Description
13	Array dimensions not specified or undefined type
900	Undefined base type
901	Nested types are not supported
902	Statement not allowed in type definition
903	Illegal or incomplete type definition
905	No such member variable. This runtime error occurs, whenever a specified member variable cannot be found.

Example program

This section provides a more useful example. It demonstrates, how user defined types can be used to enhance or replace the current usage of the COMmon block.

```
! common block
TYPE Tglobal
  INTEGER Iv
  DIM Xv$[18]
  INTEGER A(1:2)
END TYPE
!
COM Global:Tglobal
READ STRUCT Global
DATA 123,"COMMON",1,2
!
PRINT "Global.Iv=";Global.Iv
PRINT "Global.Xv$=";Global.Xv$
PRINT "Global.A(*)=";Global.A(1);Global.A(2)
CALL Sub
STOP
!
SUB Sub
   COM Global:Tglobal
   PRINT "Global.Iv=";Global.Iv
   PRINT "Global.Xv$=";Global.Xv$
   PRINT "Global.A(*)=";Global.A(1);Global.A(2)
SUBEND
```

Declaring and Dimensioning Variables

Six variable declarative statements are available to dimension arrays and strings and declare the precision of numeric variables:

COM

DIM

INTEGER

DINTEGER

SHORT

REAL

They can be placed anywhere in a program. The size (number of dimensions and bounds of each dimension) of the array, which is specified, is known as the physical or maximum size. A new working size can be specified for the array, which cannot be greater than the total number of elements of the physical size. This can be done using a REDIM statement. The working size refers to the total number of elements being used. An array identifier, consisting of the array name and *, can be used to refer to all elements in the working size.

The OPTION BASE Statement

When dimensioning arrays, you may want to specify that the default lower bound be 1 rather than 0. This is done using the OPTION BASE statement:

```
OPTION BASE 1
```

This statement must come before any of the variable declarative statements used in a program. Then any lower bound not specified is 1. (Explicitly defining a lower bound for an array always over-rules an OPTION BASE statement.)

If OPTION BASE 1 is not declared in a program, you may wish to include the statement:

```
OPTION BASE 0
```

for documentation purposes.

The OPTION BASE statement cannot be executed from the keyboard.

The DIM Statement

The DIM (dimension) statement is used to dimension and reserve memory for real-precision numeric arrays and initialize each element to 0. It is also used to dimension and reserve storage space for simple strings and string arrays. Syntax for the DIM statement is as follows:

DIM $item_1$ [,$item_2$ . . .]

DIM [*instance* : *type_name*]

DIM [*instance* AS *type_name*]

Each *item* can be one of the following:

- Numeric array (subscripts).
- Simple string [number of characters].
- User defined type.
- String array (subscripts) [number of characters].

For example:

```
10   OPTION BASE 1
20   DIM A(4,4),B$[56],C$(2,5),D$(10,10)[30],E(−5;5,−5;5)
```

Line 20 dimensions array A to be of 16 elements maximum. (The elements are REAL numeric precision, the default.) B$ is dimensioned as a simple string of 56 characters maximum; C$ is a string array of ten 18-character strings maximum (the default maximum length for strings); D$ is a string array having one hundred 30-character strings; and E is a real numeric array of 100 elements.

The maximum number of characters that may be specified for a simple string (or string array element) is 32767. This size may be limited by the memory available.

Note that in a DIM statement the subscripts must be explicitly quoted; it is not possible to use the default maximum array or string size.

User Defined Type example:

```
DIM Vehicle AS Tvehicle
```

```
DIM Vehicle:Tvehicle
```

The type has to be defined before the variable can be dimensioned, see chapter , User defined Types.

The NEW statement

The DIM statement is executed during the prerun of the program, so the TYPE has to be known and defined in the programcode at starttime. This is not always possible with User Defined Types.

The NEW statement makes it possible to dimension a variable during runtime of the program.

Syntax:

NEW *instance* : *type_name*

NEW *instance* AS *type_name*

It is possible to create a variable form an already existing one:

Syntax:

NEW STRUCT *A* = *B*

The variable *A* has the same type as the variable *B*, after executing this statement. The contant of variable B is not copied to A.

The INTEGER Statement

The INTEGER statement is used to dimension and reserve memory for integer-precision variables.

INTEGER numeric $variable_1$ [(subscripts)] [,num $variable_2$ [(subscripts)]. . .]

For example:

```
40   INTEGER X,Y(2,2)
```

declares a simple integer X and an integer array Y.

The DINTEGER Statement

DINTEGER numeric $variable_1$ [(subscripts)] [,num $variable_2$ [(subscripts)]. . .]

declares a double integer variable. (see also INTEGER above.)

The SHORT Statement

The SHORT statement is used to dimension and reserve storage for short-precision variables. Syntax is as follows:

SHORT numeric $variable_1$ [(subscripts)] [,num $variable_2$ [(subscripts)]. . .]

For example:

```
50   SHORT A(4,5,6),B(3,2,1),D
```

declares A and B as short-precision arrays and D as a simple, short precision variable.

The REAL Statement

The REAL statement is used to dimension and reserve memory for real-precision variables. Syntax is as follows:

REAL numeric $variable_1$ [(subscripts)] [,num $variable_2$ [(subscripts)]. . .]

For example:

```
60   REAL M(2,3,4,5),N
```

dimensions the array M and simple variable N.

The COM Statement

The COM (common) statement is used to dimension and reserve memory for simple, array and user defined type variables. This includes strings, all four numeric precisions and user defined types. COM is unique because it reserves memory space in a special common area which allows data to be transferred to subprograms or to other programs. Of course, data may always be transferred to subprograms using parameters. The COM statement is useful when you wish to share data among many programs. The syntax is as follows:

COM $item_1$ [,$item_2$ . . .]

COM [*instance* : *type_name*]

COM [*instance* AS *type_name*]

Each *item* can be one of the following:

- Simple numeric.
- Numeric array (subscripts).
- Simple string number of characters.
- String array (subscripts) number of characters.
- #file number.
- User defined type.
- String array (subscripts) [number of characters].

In addition, any one of the keywords INTEGER, DINTEGER, SHORT, and REAL may precede one or more numeric variables.

For example:

```
70  COM A,B(2,4),C$,#3,INTEGER E,F$(5)[24],G,SHORT H(5),I,DINTEGER
 D1,D2
```

The variables A,B(2,4) and G are real precision. Real precision is assumed at the beginning of the COM list and for numeric variables declared after any string. All variables following a numeric precision keyword have that precision until another type is specified or a string is declared. Thus both H(5) and I are short precision. The #3 item allows passing an assigned data file of the same number to another program or subprogram. For example:

```
Main Program                        Overlay

10  COM A,B,#1                      200  COM A,B,#5
20  ASSIGN #1 TO "Data"
30  CALL Data_prog
.
.
10  SUB Data prog                   Start of Subprogram
20  COM C,D,#3
.
.
```

The file number item in line 10 allows the file assigned in line 20 to remain assigned in the subprogram (as file #3) and in the overlayed program (as file #5). COM may occur anywhere in each program and may be edited.

The names of variables in corresponding COM statements need not match. But all items must be of the same type and be in the same order. Arrays must have the same number of dimensions and elements. Once a string is dimensioned in common, it is automatically dimensioned to the same size in all subsequent subprograms.

COM statements in separate programs need not have the same number of items. You need only quote the items that other programs or overlays need. A second (or further) COM statement in the main program will, if shorter, cause the omitted items to be lost or the extra files to be closed. If the succeeding COM list is longer, the new items will be dimensioned and initialized.

EXAMPLE OF TYPE USED AS GLOBAL VARIABLE

Other Features of Variable Declarative Statements

DIM, COM, INTEGER, DINTEGER, SHORT, and REAL statements are programmable only. They may appear anywhere in a program but they must not precede an OPTION BASE statement. (It is recommended that they be placed near the start of a program; clearly defined variables make a program easier to read.)

At pre-run initialization, all variables declared in DIM, SHORT, INTEGER, DINTEGER, and REAL are dimensioned and initialized. (”Initialization” means that numeric variables are set to 0 and string variables to the null string.)

DIM need not be used to assign space for strings with 18 character or less or for arrays having upper bounds of ten or

less. These can be dimensioned implicitly. (They will be set to the default—18 characters per string and 10 elements per dimension.)

A program can have more than one DIM, SHORT, INTEGER, DINTEGER or REAL statement, but the same variable name can be declared only once in a program segment. The same name, however, may be used for a simple numeric, simple string, numeric array and string array. For instance:

```
10 OPTION BASE 1
20 DIM A(5,5),A$[50],A$(10)[80]
```

These variable names are legal, although confusing.

Redimensioning an Array

A new working size for an array can be established by using the statement.

REDIM array variable$_1$ (redim subscripts)[,array variable$_2$ (redim subscripts)...]

The REDIM subscripts have features and properties identical to normal array subscripts. Any array, once redimensioned, will behave as if it was originally defined with the new dimensions. That is, accessing an element beyond the new range will return **ERROR 17**. Data in variables beyond the new range cannot now be accessed.

REDIM cannot be used to release memory space for other uses. An array redimensioned as smaller will occupy the same amount of memory; it will just act as if smaller. When using REDIM, remember that the number of dimensions cannot change and the total number of elements may not exceed the number originally dimensioned (meaning, the *maximum physical size* may not be increased).

```
10   OPTION BASE 0
20   DIM A(100),B(20,20),Astring$(25)[50],Bstring$(25,25)
30   ! Various arrays dimensioned as examples.
40   REDIM A(50)   ! Change array A from A(0:100) to A(0:50).
50   !
60   REDIM A(100:199) ! Now alter A from A(0:50) to A(100:199).
70   !
80   REDIM B(10,15)   ! Change B(0:20,0:20) to B(0:10,0:15).
90   !
100REDIM B(1:21,1:21) !Restore B to original size with new lower b
ound.
110 !
120 REDIM A(80),B(5,35) ! Change both arrays at once.
130 !
140 REDIM B(40)!Incorrect number of dimensions may not be altered.
150 !
160 REDIM A(120) ! Incorrect original number of elements exceeded.
170 !
180 REDIM Astring$(10) ! Change Astring$ from (0:25)[50] to (0:10)
[50].
190 !
200 REDIM Bstring$(5,100) ! Change Bstring$ from (0:25,0:25)[18].
210 !to (0:5,0:100)[18] The default string length does not change.
220 !Bstring had 26*26 = 676 elements, now has 6*101=606 elements.
240 END
```

Assigning Values to Variables

Values can be assigned to variables either from within a program or from external sources (normally a keyboard or data file). This chapter describes most of the statements used to assign values; the others, used in file handling, are described in page 195 . The statements currently covered are as follows:

NOTE: The Keywords INPUT, LINPUT, EDIT, ENTER and LENTER are available on HP-UX systems, only.

LET

READ (from DATA)

INPUT

LINPUT

EDIT

ENTER

LENTER

The LET statement was introduced earlier. Many other statements also assign values to variables (READ#, ASSIGN, READ LABEL, etc.) as described in other chapters of the manual.

The READ and DATA Statements

To assign values to variables from within a program, the DATA statement is used with READ. The DATA statement provides values; READ specifies the variables for which values are to be obtained.

READ variable name$_1$ [,variable name$_2$ . . .]

$$\text{DATA} \begin{Bmatrix} \text{constant} \\ \text{text} \end{Bmatrix} \left[, \begin{Bmatrix} \text{constant} \\ \text{text} \end{Bmatrix} \right]$$

Text can be quoted or unquoted. For example:

```
70    DATA 88,April,"100","Pay=",95
80    READ A,Date$,Pay$[5,7],Pay$[1,4],Array(1)
```

The variables specified in the READ statement can be any variable type, including an array identifier which specifies an entire array. The subscripts can be any numeric expression except one containing a function subprogram (FN) reference. Array elements are read in order with the rightmost subscript varying fastest.

For example:

```
10   OPTION BASE 1
20   DIM A(2,2,2) ! A 3-dimensional 8 element array (2*2*2)
30   DATA 1,2,3,4,5,6,7,8
40   READ A(*)
50   PRINT A(*)
60   END
RUN
1                   2

3                   4

5                   6

7                   8
```

The array elements are assigned values in this order:

```
A(1,1,1)A(1,1,2)A(1,2,1)A(1,2,2)A(2,1,1)A(2,1,2)A(2,2,1)A(2,2,2)
```

READ is programmable only; it cannot be executed from the keyboard.

The DATA Pointer

The computer uses an internal mechanism called a DATA pointer to locate the next data item that is to be read. When the program is run, the Data pointer points to the first (leftmost) item of the first (lowest-numbered) DATA statement in the current segment. After this item is read the DATA pointer shifts one item to the right, pointing to the next item to be read. This operation is made each time a data item is read. After the last item in a DATA statement is read and another value is required by READ, the DATA pointer locates the next highest numbered DATA statement and is set to the first item in that statement. If there are no higher-numbered DATA statements, the data pointer remains at the end of the previous DATA statement; **ERROR 36** indicates the end of data.

The location of the DATA statement within a program segment is unimportant. If there are multiple DATA statements, however, make sure they are in the order you want.

The RESTORE statement

The DATA pointer can be repositioned to the beginning of any DATA statement using the RESTORE statement:

RESTORE [line id]

If no *line id* is specified, the pointer is repositioned to the beginning of the lowest numbered DATA statement. If the specified line is not a DATA statement, then the first DATA statement following that line is accessed.

The next example shows that several READ statements can apply to the same DATA statement. It also shows that string values can be quoted or unquoted, though quotes are not part of the string. Notice that 7.31 is a string value assigned to A$.

```
100   READ A,B,C
110   READ D$,E
120   READ F$
130   DATA 4,5,6,7.31,2.69,"Hours"
```

The next example illustrates the use of RESTORE. The values in line 30 are assigned to five simple variables, then re-used as the values in array B.

```
10 OPTION BASE 1
20 DIM B(5)
30 DATA 4,9,16,25,30
40 FOR I=1 TO 5
50    READ C
60    DISP "Square root of";C;" is ";SQR(C)
70 NEXT I
80 DISP LIN(2)
90 RESTORE 30 ! Parameter not essential as only one DATA line here
100 READ B(*)
110 PRINT B(*)
120 END
RUN
Square root of 4  is 2
Square root of 9  is 3
Square root of 16  is 4
Square root of 25  is 5
Square root of 30  is 5.47722557505

4      9      16        25
30
```

The INPUT Statement

NOTE: The Keyword INPUT is available on HP-UX systems, only.

The INPUT statement suspends program execution, allowing values in the form of expressions to be assigned to variables from the keyboard. Syntax is as follows:

$$\text{INPUT}\left[\text{"prompt"}\begin{Bmatrix};\\,\end{Bmatrix}\right]\text{var. name}_1\left[,\left[\text{"prompt"}\begin{Bmatrix};\\,\end{Bmatrix}\right]\right]\text{var. name}_2$$

When the INPUT statement is executed, a ? or the prompt, if present, appears in the display line. The prompt may be any combination of characters normally used to tell the user what the request is. A value can then be input for each variable designated in the INPUT statement. For instance, the following statement requests two values (the prompt will be printed once):

```
340   INPUT "ENTER NAME AND EMPLOYEE NUMBER";Emp_name$,Emp_number
```

Values can be entered individually or in groups (separate each variable with a comma). Values for strings can be quoted or unquoted but an unquoted value may not contain a comma. For example, the values "A.Jones" and 250 can be assigned to the variables above in many ways; here are two:

```
A.Jones, 250 RETURN
```

or

```
"A.Jones" RETURN 5*50 RETURN
```

The ? reappears after RETURN is pressed until all values are input. If there is only one prompt, it will not reappear; ? appears instead. So it is best to use a prompt with each variable:

```
360   INPUT "ENTER NAME";Name$,"EMPLOYEE NUMBER";Number
```

Using a semicolon after the prompt causes the input to be entered on the same display line as the prompt, as in the previous examples. Using a comma after the prompt places the entry on the next display line.

Pressing RETURN without entering a value causes execution to continue with the next variable in the list. Variables not assigned values retain their previous value. For example:

```
40   X=5
50   PRINT X
60   INPUT "ENTER THREE VALUES:";A,B,X
```

By responding to the INPUT statement with 2,4, X retains its previously assigned value of 5.

The variable list can also include array identifiers:

```
370   INPUT A,B(*)
```

The INPUT statement is also used without parameters to suspend program execution. The operator can then enter data into the display, to be read by succeeding ENTER or LENTER statements (described later). Program execution is resumed by pressing RETURN.

The INPUT statement is programmable only; it cannot be executed from the keyboard.

The LINPUT Statement

NOTE: The Keyword LINPUT is available on HP-UX systems, only.

The LINPUT statement pauses program operation and allows the operator to enter an entire display line of information to a string variable. Pressing RETURN assigns the line to that string variable. Syntax for the LINPUT statement is as follows:

LINPUT ["prompt" { ; | , }] string variable

When LINPUT is executed, a ? or the prompt, if present, appears in the display line. Up to 160 characters can be entered with each LINPUT, although string subscripts could limit the input to less. For example:

```
380   LINPUT "ENTER HIS RESPONSE:",Response$[1,30]
```

The response could be: "Maximum 30 chars", David said

If a semicolon had followed the prompt above, the string value would appear immediately following the prompt. Pressing RETURN would then input the prompt along with the string. Pressing RETURN without typing in a value (not even a space) erases the current value of the string and sets it to the null string.

Notice that the LINPUT statement allows quotation marks to be input within a string variable; this is not possible with the INPUT statement.

The LINPUT statement cannot be executed from the keyboard and LINPUT cannot enter information from a protected display line (see section , Display Enhancements, on page 248).

The EDIT Statement

NOTE: The Keyword EDIT is available on HP-UX systems, only.

The current value of a string can be changed by using the EDIT statement. Syntax is as follows:

$$\text{EDIT}\left[\text{"prompt"}\left\{\begin{matrix};\\,\end{matrix}\right\}\right]\text{string expression}$$

When the EDIT statement is executed, a ? or the prompt, if present, is displayed and followed by the current value of the specified string variable.

This value can be edited like any keyboard entry. You can clear the line, allowing a totally new value to be entered, as with LINPUT. Pressing RETURN stores the characters in the line as the value of the string. A trivial example is shown here:

```
100   DIM String$[60]
110   String$="Uncle Sam"
120   EDIT "CHANGE NAME?",String$
130   PRINT "CURRENT NAME IS:";String$
```

When line 120 is executed, CHANGE NAME is displayed. The string Uncle Sam then appears on the next line. The character editing keys may now be used to change the name. Pressing RETURN inputs the entire line into the variable. Line 130 prints the new name.

If a semicolon had followed the prompt in the EDIT statement, the string value would be displayed immediately following the prompt. Pressing RETURN would then input the prompt along with the string. Pressing RETURN without entering a value re-enters the current value of the string. Clearing the line before pressing RETURN, will set the variable to the null string.

The limit on the length of the string being edited is 160 characters, the display line length. This can be avoided by using substrings and multiple EDITs.

The EDIT statement cannot be executed from the keyboard.

The ENTER Statement

NOTE: The Keyword ENTER is available on HP-UX systems, only.

The ENTER statement reads data already on the display into the specified variables. It does not pause to allow keyboard entry. The syntax is as follows:

ENTER variable $name_1$ [,variable $name_2$ . . .]

Data input begins at the current position of the display cursor and continues until the variable list is filled. As with INPUT, the variables read must satisfy the variable types in the list. If not, an error occurs.

The cursor position can be altered before executing ENTER by using the CURSOR statement (see section , Display Enhancements, on page 248). The ENTER statement cannot be executed from the keyboard.

The ENTER statement is intended for use with software which places forms on the display and specifies locations or input fields where the operator enters data (see the next sample form). After data is entered into the fields, ENTER and LENTER statements are used to read the data into variables according to a preset order.

```
                              New Customer Entry

Customer Name:

Surname:______________ First name:__________________

Customer Address:

Street:___________________

Town: ______________  State: _____________

Zip Code:________________

Telephone:

Area code:______  Number: _______
```

NOTE: Enter all the customer address information. Press TAB to move through the form, and press RETURN when complete.

The following program segment governs the above form. The INPUT statement on line 240 halts the program. Information on a new customer may then be entered. The TAB key is used to move from field to field. The RETURN key should be pressed upon completion of the form. Program execution then restarts and the ENTER statements read the data into the program variables in the preset sequence. Note that the Address lines, although separate entries on the screen, are loaded as substrings of one large string variable.

```
220 Input_data:!
230     CURSOR (5,5)    ! set cursor at first field.
240     INPUT           ! allow operator to fill fields.
250     CURSOR (5,5)    ! Reset cursor to first screen input
260     ENTER Surname$,Firstname$
270     ENTER Address$[1,19]
280     ENTER Address$[20,34],Address$[35,39]
290     ENTER Zip$,Phone$[1,9],Phone$[10,19]
```

These operations are described more fully in page 249 .

The LENTER Statement

NOTE: The Keyword LENTER is available on HP-UX systems, only.

The LENTER statement inputs data from one line of display into a specified variable, like LINPUT, but, like ENTER, does not pause to allow keyboard entry. Syntax is as follows:

LENTER string variable

The line length is limited to 512 characters, although string subscripts in LENTER could limit the input to less.

LENTER may only be used in a program, and it may not enter data from a protected display line (**`ERROR 38`** is returned; protected display lines are fully covered in page 249). Use XLENTER to read a protected line.

The following program uses LENTER to load information provided by a CAT statement into a string array. The information can then be analyzed. (CAT does not store data; it merely outputs it to the SYSTEM PRINTER, here, of course, the display.)

The program prompts for the required volume and the file type which you wish to examine. The CAT (catalog) statement displays the file catalog for the requested directory. The Load_array routine then stores the catalog, line-by-line, into the string array Cat$. The CURSOR statement, line 50, positions the display cursor at the first line of the catalog. The Search routine finds and lists the files of the specified type.

The FOR loop in Load_array is performed 150 times (once for each array element), unless there are fewer than 150 files in the directory.

```
10 DIM Cat$(150)[80],Dir$[6],Type$[4]
20 DISP "~~" ! Alternate char set "CURSOR HOME, CLEAR DISPLAY"
30 INPUT "Enter Volume name: ";Dir$
40 INPUT "Now enter the File Type to be listed";Type$
50 CAT ","&Dir$  ! List required catalog
60 CURSOR (1,5) ! Set cursor to 1st char 4th line
70 ! note that the first 4 lines of the display must be skipped
80 !
90 Load_array:  ! Load CAT output into array Cat$
100    FOR Line=1 to 150 ! so don't over-run array boundary
110        LENTER Cat$(Line)
120    NEXT Line
130 !
140 Search:  ! Find and display files of specified type
150    DISP "~ ~" ! Alternate char set "CURSOR HOME, CLEAR DISPLAY"
160    DISP SPA(7);"FILE TYPE: ";Type$;SPA(5);" ON VOLUME: ";Dir$
170    FOR Line=1 TO 150
180        IF POS(Cat$(Line),"."&Type$ THEN DISP Cat$(Line)
200    NEXT Line
210 END
```

Here is the result of a sample run:

```
       FILE TYPE: PROG      ON VOLUME: LOCAL
-rw-rw-rw    1  john     users       580  Mar  2  1997 AK.PROG
-rw-rw-rw    1  john     users       182  Feb  8 16:35 BK.PROG
-rw-rw-rw    1  john     users       344  Mar  2  1997 INFO.PROG
-rw-rw-rw-   1  john     users       512  Mar  2  1997 KEYTST.PROG
-rw-rw-rw-   1  john     users       960  Feb 12 11:36 LENTER.PROG
-rw-rw-rw-   1  john     users      1504  Jan 28 17:40 akprog.PROG
```

A full description of the CURSOR statement is given in page 249 . CAT and the other Volume (HP-UX) operations are described in page 195 and page 25 .

The XLENTER Statement

NOTE: The Keyword XLENTER is available on HP-UX systems, only.

With extended LENTER up to 512 characters and protected lines may be ENTERed.

Syntax:

XLENTER string variable

Sample hardcopy routine using XLENTER:

```
SUB Hardcopy
   DIM A$[512]            ! Line buffer
   X=XPOS                 ! Save current x/y position
   Y=YPOS
   PRINTER IS 0           ! Open printer
   FOR I=1 TO 21
      CURSOR(1,I)         ! Read and print screen line
      XLENTER A$
      PRINT A$
   NEXT I
```

```
   PRINTER IS 8          ! Close printer
   CURSOR (X,Y)          ! Restore cursor position
SUBEND
```

This routine will work with form and protected lines too.

The ACCEPT Statement

NOTE: The Keyword ACCEPT is available on HP-UX systems, only.

The ACCEPT statement loads a string into a string variable without displaying (echoing) the input. It is used for the entry of sensitive information such as passwords, where you do not wish the input to be visible on the screen. Syntax for the statement is as follows:

ACCEPT string variable

The ACCEPT statement has certain other characteristics:

- The cursor does not appear while you are inputting the string. Once the RETURN key is pressed, the cursor appears in its usual state.
- All softkeys are ignored while input is being ACCEPTed.
- Line drawing characters are ignored if you input them. They will not be returned in the input string.

Here is a sample of code using the ACCEPT statement. The operator has only one attempt at the password here; more often a loop is used to allow two or three tries.

```
10    PRINT "Please enter the Sales Ledger Password"
20    ACCEPT Pass$
30   IF Pass$<>"Hyacinth" THEN Inval_error ! Invalid password; no
entry
40         ELSE Sales_ledger ! Password O.K so goto sales ledger
.
.
450 Inval_error: !
460     DISP "Invalid Sales ledger password - entry not permitted"
470     STOP
```

KBCODE

NOTE: The Keyword KBCODE is available on HP-UX systems, only.

Waits for a key and returns internal code of key.

This internal code is simply the ROMAN8 code ('A' = 65) for all non function keys. All function keys return codes of 256 + id (e.g. F1 = 265, F2 = 266as defined by curses.h).

In this sample program you could use normal keys as softkeys:

```
DISP "Select option (1 .. 3 or E)"
LOOP
   K = KBCODE
   EXIT IF CHR$(K)="E"
   SELECT CHR$(K)
   CASE "1","2","3":
      DISP "-> option ";K-NUM("0")
   CASE ELSE
      BEEP
   END SELECT
END LOOP
END
```

This is a more complex example. It could be used as a replacement of the ACCEPT statement. It will maintain an edit string of given length. For each character you enter a '*' as output. You can correct your input with the BACKSPACE key. Input will be finished either if you press RETURN or if you enter Max_len characters:

```
DEF FNAccept$(Max_len)
   DIM A$[Max_len]
   LOOP
      C=KBCODE
      SELECT C
      CASE 13                              ! CR
         EXIT IF 1
      CASE 0 TO 7,9 TO 12,14 TO 31,>127    ! CONTROL
         BEEP
      CASE 8                               ! BACKSPACE
         IF LEN(A$) THEN
            A$=A$[1,LEN(A$)-1]
            DISP "~ ~";                    ! "BS SPACE BS"
         ELSE
            BEEP
         END IF
      CASE ELSE                            ! CHARACTER
         A$=A$&CHR$(C)
         DISP "*";
         IF LEN(A$)=Max_len THEN
            BEEP
            EXIT IF 1
         END IF
      END SELECT
   END LOOP
   RETURN A$
FNEND
```

Eloquence keyboard handling

Two statement groups allow better control of the Eloquence keyboard processing.

NOTE: These keywords are not available when working with graphical user interface.

Typeahead

Eloquence normally rejects any key pressed while it is busy with a beep signal. The TYPEAHEAD statement enables the program to control Eloquence keyboard handling

```
TYPEAHEAD n
```

The typeahead statement adjusts the keyboard mode. Changing the keyboard mode implies a TYPEHEAD CLEAR operation.

n = 0 Default behavior. Any key pressed while Eloquence is not in input state will be rejected.

n = 1 Partial typeahead. Any key, which is not a function key is saved in a typeahead buffer for later processing.

Function keys are executed immediately.

n = 2 Full typeahead. All keys are saved in the typeahead buffer for later processing.

In TYPEAHEAD mode 1, the program must be aware, that there may be some saved keystrokes, if reacting upon a function key.

In TYPEAHEAD mode 2, it's not possible to interrupt a program (for example abort a very long printout) using a function key. Function keys loose their meaning, while not in INPUT mode.

TYPEAHEAD CLEAR

Clear the typeahead buffer. If a character is currently in typeahead buffer, a beep is output to indicate the loss of input.

```
N=TYPEAHEAD
```

The typeahead function returns the number of characters currently in typeahead buffer.

XPACK, XUNPACK statements

The XPACK and XUNPACK statements provide a convenient way to transfer string and numeric data to and from a string variable. They are particular useful in conjunction with Dialog Manager record objects.

Unlike the PACK USING and UNPACK USING statements, the XPACK and XUNPACK statements include the variable name in the packed string and use a variable length string format.

The XPACK statement

The XPACK statement transfers data from each variable of the variable list to the destination string. A variable list may be specified in three different ways:

- a string expression evaluating in a variable list
- a variable list
- a reference to IN DATA SET LIST variable lists

A reference to a whole array will be resolved into a list of array elements.

As the data is transferred to the destination string, it is converted into string format and stored along with the variable name and array index.

The XPACK and XUNPACK statements can operate on a STRUCT. STRUCT is treated as a list of variables.

The XPACK statement packs each member variable in a buffer. The XUNPACK statement unpacks to a user defined type if it is passed as an argument to XUNPACK and the member variable matches the name in the buffer.

XPACK ... USING

XPACK Dest$ USING <string expression>

Pack destination string variable from variable list specified by the string expression. The string expression contains a list of variable names separated by a comma.

For example:

```
List$="A,A$,B,B$,A$(1),Array$(*)"
```

XPACK ... USING REMOTE LISTS

XPACK Dest$ USING REMOTE LISTS Label [,Label . . .]

Pack destination string variable from variable list as defined by the referenced remote list(s).

For example:

```
XPACK Dest$ USING REMOTE LISTS Label1,Label2
Label1: IN DATA SET LIST A,A$
Label2: IN DATA SET LIST ...
```

XPACK ... FROM

XPACK Dest$ FROM Variable [,Variable . . .]

Pack destination string variable from variable list.

For example:

```
XPACK Dest$ FROM A,A$,B,B$,A$(*)
```

The XUNPACK statement

The XUNPACK statement transfers data from a string variable into the original variables. If a variable list is specified, only those variables are unpacked, that are included in the variable list. It's not possible to unpack a buffer into a different variable than used to pack the buffer.

XUNPACK Buf$

Unpack buffer variable into variables as named in the buffer.

For example:

```
A$="Test"
B=123
XPACK Buf$ FROM A$,B
A$=""
B=0
XUNPACK Buf$
```

This recovers the initial values of A$ and B.

`XUNPACK Buf$ USING <string expression>`

Unpack buffer variable into variables as named in the buffer. The string expression contains a list of variable names separated by a comma. Only variables included in the variable list are unpacked.

For example:

```
XPACK Buf$ FROM A$,B,C
List$="A$,B"
XUNPACK Buf$ USING List$
```

This should unpack the variables A$ and B only.

XUNPACK Buf$ FROM Variable [,Variable ...]

Unpack buffer variable into variables as named in the buffer. Only variables included in the variable list are unpacked.

For example:

```
XPACK Buf$ FROM A$,B,C
XUNPACK Buf$ FROM A$,B
```

This unpacks the variables A$ and B only.

XUNPACK Buf$ USING REMOTE LISTS Line_id[,Line_id ...]

Unpack buffer variable into variables as named in the buffer. Only variables included in the referenced REMOTE LISTS list are unpacked.

For example:

```
XPACK Buf$ FROM A$,B,C
XUNPACK Buf$ USING REMOTE LISTS Label
Label: IN DATA SET LIST A$,B
```

XPACK format description

Each variable in a XPACK statement is converted into a string format containing fields describing variable type, name, index and value.

The following rules apply:

- Format fields are separated by tilde characters ('~').
- All fields are printable (no binary data).
- Numeric values are converted into string.

Table 5 Format Fields

Type	Name	~	Idx	~	Len	~	Value	...	$

~ Field separator. The Tilde character ('~').

Type Field type. One of the following:

N numeric
X string
$ End-of-list

Name Name = field name. Up to 15 characters. Must be valid Variable name. Trailing $ of string variables is omitted.

Idx Array index. 0 = simple variable.

First array element is 1 (independent of array bounds and number of dimensions).

Len Size (number of bytes) of value field.

Value Value field. Any number of characters.

Simple numeric variable

```
Simple = 47
XPACK Buf$ FROM Simple
```

Results in the following buffer:

```
"NSimple~0~2~47$"
```

Equivalent Eloquence code:

```
Buf$="N"&"Simple~"&"0~"&VAL$(LEN(VAL$(Simple)))
     &"~"&VAL$(Simple)&"$"
```

Simple string variable

```
Simple$ = "String"
XPACK Buf$ FROM Simple$
```

Results in the following buffer:

```
"XSimple~0~6~String$"
```

Equivalent Eloquence code:

```
Buf$="X"&"Simple~"&"0~"&VAL$(LEN(Simple$))&"~"&Simple$&"$"
```

Array element

```
A$(1)="TEST"
XPACK Buf$ USING "A$(1)"
```

Results in the following buffer:

```
"XA~1~4~TEST$"
```

Array

```
DIM A$(0:3)
A$(0)="000"
A$(1)="111"
A$(2)="222"
A$(3)="333"
XPACK Buf$ FROM A$(*)
```

Results in the following buffer:

```
"XA~1~3~000XA~2~3~111XA~3~3~222XA~4~3~333$"
```

Example Eloquence XUNPACK program

The example program below shows how to unpack a buffer in XPACK format. This example has been provided only for clarification of XPACK format.

```
DIM Type$[1],Var_name$[15],Value$[80],Variable$[22]
INTEGER P,Idx,Len

LOOP
   Type$=Buf$[1,1]             ! Type
   EXIT IF Type$="$"
   Buf$=Buf$[2]

   P=POS(Buf$,"~")             ! locate separator
   Var_name$=Buf$[1;P-1]       ! Variable name
   Buf$=Buf$[P+1]

   P=POS(Buf$,"~")             ! locate separator
   Idx=VAL$(Buf$[1,P-1])       ! Array index
   Buf$=Buf$[P+1]

   P=POS(Buf$[P],"~")          ! Separator
   Len=VAL(Buf$[1,P-1])        ! Value length
   Buf$=Buf$[P+1]

   Value$=Buf$[1;Len]          ! Value
   Buf$=Buf$[Len+1]

   IF Idx THEN
      Variable$=Var_name$&"("&VAL$(Idx)&")"
   ELSE
      Variable$=Var_name$
   END IF

   IF Type$="X" THEN
      COMMAND Variable$&"$=Value$"
   ELSE
      COMMAND Variable$&"="&Value$
   END IF
END LOOP
```

Memory Consumption

Use the following tables to work out the number of bytes needed in main memory for each type of variable.

	Simple Variables
Real precision	4 bytes + 12 bytes
Short precision	4 bytes + 12 bytes
Integer precision	4 bytes + 4 bytes
Dinteger	4 bytes + 4 bytes
String	8 bytes + length (1 byte per character, rounded up to a 4 byte boundary).

	Array Variables
Real precision	8 bytes + 8 bytes per dimension + 12 bytes per element
Short precision	8 bytes + 8 bytes per dimension + 12 bytes per element
Integer precision	8 bytes + 8 bytes per dimension + 4 bytes per element
Dinteger	8 bytes + 8 bytes per dimension + 4 bytes per element
String	12 bytes + 8 bytes per dimension + 4 bytes per element + length of each string (1 byte per character, rounded up to a 4 byte boundary).

The rightmost number shows the space needed in memory for the value; the preceding numbers show the overhead required for the variable description. (This is information on the type of variable and subscript information for arrays.) For example, one REAL variable takes 4 bytes for the variable description and 12 bytes for the full precision number—16 bytes in all.

Control Byte Overhead

Control bytes are automatically added to each end of a string containing display-enhanced characters or characters from an alternate set (for example, line drawing or underlining characters).

In general, one byte is added to each end of the string for each alternate character set used. So a string of blinking, inverse-video, line-drawing characters requires four additional bytes of memory—two for setting and resetting the enhancement mode and two for setting and resetting the line-drawing set.

5

Operators and Functions

This chapter describes the various operations and functions that can be performed on numeric and string data. Most applications use only a small subset of the available operators and functions.

Operators and Expressions

An operator indicates a mathematical or logical operation to be performed on one or more values (operands) resulting in a single value. The combination of one or two operands with an operator is called an expression. For example, 10 + 5 is an expression consisting of two operands (10 and 5) and one operator (+).

The term operand can refer to a number, a string, or a variable.

An operator is generally placed between two operands, but some operators can precede a single operand. For instance, the minus sign is an operator which indicates subtraction when it appears between two operands (for example, 512–88), but it indicates negation when it appears before a single operand (for example, –1).

Some examples of expressions are as follows:

```
A − B
X + 1
"AB"&"CD"
−1
+2.14
2.14*
N*
N$**
"STRING"**
```

* These are expressions in which the operator is assumed to be a plus sign (+).

** Strings or string variables can also be considered expressions.

Operators are divided into four classes depending on the kind of operation performed—arithmetic, string, relational, and logical (Boolean).

Arithmetic Operators

The arithmetic operators are listed in the following table:

Table 6 Columns

Operators	Operations	Examples
+	add	10+5 = 15
–	subtract	10–5 = 5
–	negate	–2
*	multiply	10*5 = 50
/	floating point divide	15/10 = 1.5
^ or **	exponentiate	8^3 = 512
DIV	integer divide: A DIV B =	15DIV10 = 1
	SGN (A/B)*INT(ABS(A/B))	–15DIV10 = –1
MOD	modulo:	38MOD6 = 2
	A MOD B = A–(B*INT(A/B))	–13MOD2 = 1
		–13MOD–2 = –1

Note that, unlike algebraic notation, implied multiplication does not exist in Eloquence; thus, A times B must be written as A*B rather than just AB. The operation of raising a number to a power also requires an explicit operator; thus, A^B is written as either A^B or A**B.

There are two division operators—/ and DIV. Division with the / operator (called floating-point division) results in a value in the real-precision range. When the DIV operator is used (integer division), the result is computed using the integer value of each operator. In either case, the results are returned in real precision. For example:

```
3/2 = 1.5
3 DIV 2 = 1

−10/5 = −2.0
−10 DIV 5 = −2

9.999999999/1 = 9.999999999
9.999999999 DIV 1 = 9
```

The function INT(X), which is used to calculate A MOD B, returns the greatest integer less than or equal to X. So, using the formula

A MOD B = A – B * INT (A/B), we have:

```
38MOD6      = 38 – 6 * INT (38/6)
            = 38 – 6 * 6
            = 38 – 36
            = 2

–13MOD2     = –13 – 2 * INT (–13/2)
            = –13 – 2 * (–7)
            = –13 – (–14)
            = 1

–13MOD–2    = –13 – (–2) * INT (–13/–2)
            = –13 – (–2) * 6
            = –13 – (–12)
            = –1
```

Expressions with more than two operands are evaluated according to the following hierarchy of arithmetic operators:

```
^ **             highest
* / DIV MOD
+ –              lowest
```

Note the following examples:

```
5 + 6 * 7 = 5 + 42 = 47
5 * 6 + 7 = 30 + 7 = 37
```

If operators are at the same level, the order is from left to right in the expression:

```
30 – 40 + 100 = –10 + 100 = 90
2 + 3^2 – 1 = 2 + 9 – 1 = 11 – 1 = 10
```

Parentheses can be used to override this order, as shown in the following examples:

```
30 – (40 + 100) = 30 – 140 = –110

2 + 3^(2–1) = 2 + 3^(1) = 2 + 3 = 5

5 + 6 * 7 = 5 + 42 = 47
(5 + 6) * 7 = 11 * 7 = 77

14/7*6/4 = 2*6/4 = 12/4 = 3
14/(7*6)/4 = 14/42/4 = .333.../4 = .08333...
```

When parentheses are nested, operations within the innermost pair are performed first:

```
100/((4+6)*2) = 100/(10*2) = 10/20 = 5
2*((3+4)—5)/6 = 2*(7—5)/6 = 2*2/6 = 2*2/6 = 4/6 = .6666...
```

Relational Operators

Table 7 The relational operators are:

Operators	Operations	Examples
<	less than	A < B
>	greater than	A > B
<=	less than or equal to	A <= B
>=	greater than or equal to	A >= B
=	equals	A = B
<> or #	not equal to	A <> B

When relational operators are used in a numeric expression, the value 1 is returned if the relation is found to be true; the value 0 is returned if the relation is false. For instance, A=B is evaluated as 1 if A and B are equal in value, 0 if they are not equal. If A=1, B=2, and C=3, then (A*B)<(A–C/6) is evaluated as 0 (false) because A times B equals 2, which is not less than 0.5 (the result of (A–C/6)).

Here is a sample program:

```
10    INPUT “ENTER THREE VALUES:”;A,B,C
20    Logic=(A*B<C)
30    If Logic THEN
40        PRINT “(A*B<C) is TRUE and has a value of “;Logic
50    ELSE
60        PRINT “(A*B<C) is FALSE and has a value of “;Logic
70    END IF
80    END
```

Entering the values 3, 4, and 5 results in:

```
(A*B<C) is FALSE and has a value of 0
```

Notice that the IF expression is true for any non-zero value.

Relational operators are also used to compare strings. Strings are compared according to their associated numeric values in the ASCII code (see page 391). The strings are compared character-by-character until a difference is found or until the end of a string is reached. If the ends of both strings are found at the same

time, the strings are equal. If the end of one string is reached first, however, then that string is an initial substring of the other, and is considered to be less than the longer string. For example, all the following expressions are true:

″AB″ < “ABC″   “AB″ is an initial substring of “ABC″.

″AB″ = “AB″   Both strings are exactly equal.

″B″ > “ABC″   “B″ has a higher numeric equivalent than “A″ in the ASCII code.

″AB$″ < “AB*″ “$″ has a lower numeric equivalent than “*″ in the ASCII code.

The null string ("") is always less than every other string and equal only to another null string. More information on substrings is given in page 73 . Relational operators are also usable with foreign-language keyboards, but results may not match ASCII values.

Logical Operators

The logical operators (sometimes called Boolean operators) are AND, OR, EXOR and NOT. These operators are most frequently used as part of an IFTHEN statement, as shown in the next chapter.

AND compares two expressions. If both expressions are true (a non-zero value), the result is true. If one or both of the expressions are false (zero), the result is false.

numeric expression AND numeric expression

OR (inclusive OR) compares two expressions. If either expression is true, the result is true. If both expressions are true, the result is true. If both expressions are false, the result is false.

numeric expression OR numeric expression

EXOR (exclusive OR) compares two expressions. If only one of the expressions is true, the result is true. If both are true or both are false, the result is false.

numeric expression EXOR numeric expression

NOT changes the logical value of an expression. If the expression is true, NOT changes it to false. If the expression is false, NOT changes it to true.

numeric expression EXOR numeric expression

As in the case of relational operators, if the result is true, a 1 is returned; if the result is false, a 0 is returned.

The expressions that the logical operators compare can be either relational or non-relational:

- If the expression is relational , its true or false designation is determined by the relational operation.
- If the expression is non-relational (A), it is true if its arithmetic value is not equal to 0; it is false if its arithmetic value equals 0.

For the following examples, assume A=0, B=2, C=4, and D=4:

A < B AND C = D True. Both relational expressions A<B and C=D are true.

A AND C = D False. The arithmetic value of A equals 0 (false).

A OR B True. The arithmetic value of B is not 0 (so B is true).

A EXOR B True. One value is true and one value is false.

NOT A True. A is 0 (false).

NOT B OR NOT C False. NOT B is false and NOT C is false.

A truth table can be used to summarize logical operations:

Table 8 **Columns**

A	B	A AND B	A OR B	A EXOR B	NOT A
T	T	1	1	0	0
T	F	0	1	1	0
F	T	0	1	1	1
F	F	0	0	0	1

Binary Operations

Binary operations test and manipulate the individual bits within a single 32-bit word in random access memory. The bit strings referred to in the descriptions are a contiguous series of sixteen bits (1's and 0's) stored in an INTEGER. The rightmost bit (least significant) is designated as bit 0; the leftmost (most significant bit) is bit 31. Operands may be real expressions but must evaluate to a 32-bit signed integer.

BINAND (BINary AND) compares the individual bits in two values which have been rounded to integers. The results are stored in an INTEGER and consist of a 1 in each bit position for which both inputs were 1, and 0's in each bit for which both or either inputs were 0. An error occurs if the values cannot be stored in sixteen bits.

BINAND (numeric expression, numeric expression)

BINIOR (BINary Inclusive OR) compares each bit of two expressions. When a 1 appears in either expression, a 1 is stored in the result. If both bits are 1, a 0 is stored in the result.

BINAND (numeric expression, numeric expression)

BINEOR (BINary Exclusive OR) compares each bit of two expressions. If both bits are 1's or 0's, the result is 0. If only one of the bits is 1, the result is a 1.

BINAND (numeric expression, numeric expression)

BINCMP (BINary CoMPlement) returns a 1 for each bit where the input was a 0, and a 0 in each bit for which the input was a 1 (this gives the complement of the input expression). An error occurs if the rounded numeric expression cannot be stored in an INTEGER.

BINAND (numeric expression, numeric expression)

BIT (Binary Index) contains two expressions. The second expression is evaluated and rounded to an integer which must be in the range 0 to 31. This integer is then used as the index into the binary representation of the integer value of the first expression. Bit 0 is the least significant bit; bit 31 is the most significant bit. The function returns an integer containing the value (0 or 1) found in this position.

An error results if the value of the first expression cannot be rounded to an integer.

BINAND (numeric expression, numeric expression)

ROTATE (binary ROTATE function) evaluates two expressions and rounds them to integers. The first expression is considered a 32-bit word. The function returns the value, as an INTEGER, of the string obtained by rotating the first expression the number of positions specified by the second expression MOD 32. If this second expression is positive, the rotation is toward the least significant bit, and this (least significant) bit is rotated to the most significant bit position. If the second expression is negative, then the rotation is in the opposite direction and bits are transferred from the most to the least significant position. ROTATE does not change the value of the first argument.

An error results if either of the rounded numeric expressions cannot be stored in an integer.

SHIFT (binary SHIFT function) consists of two expressions which are evaluated and rounded to integers. The first expression is considered a 32-bit word which is shifted the number of positions specified by the second expression MOD 32. If the second expression is positive, the shift is toward the least significant bit. If it is negative, the shift is in the opposite direction. Bits which are shifted out are lost and replaced by zeros on the opposite end of the argument. SHIFT does not change the value of its first argument.

An error occurs if either of the rounded numeric expressions cannot be stored in an INTEGER.

SHIFT (numeric expression, numeric expression)

Operational Hierarchy

When arithmetic, relational, and logical operators appear in the same expression, the operations are performed according to this hierarchy:

```
() (parentheses)                    highest
^ or ** (exponentiation)
NOT, unary +,−
*,/,MOD,DIV
+,−
=,<,>,<=,>=, or # (relational)
AND
OR,EXOR                             lowest
```

Remember that the order of execution for operations of the same priority level is from left to right. When parentheses are used, however, the operations within parentheses are executed first.

The Default ON/OFF Statements

Many arithmetic errors occur due to either an improper argument value or by exceeding the computing range. This suspends program execution. It is possible to override some of these errors by providing a default value for the number which is out of range. The default values are enabled by executing the DEFAULT ON statement:

DEFAULT ON

Errors that can be overriden and their corresponding default values are shown in the following table:

Table 9 **Default values**

Error (Number)	Default Value
Integer precision overflow (20)	2147483647 (2^{31}–1) or –2147483648 (–2^{31}) (DINTEGER)–32768.+32767 (INTEGER)
Full precision overflow (22)	+ or –9.99999999999E125
Zero to negative power (26)	9.99999999999E+125
LGT or LOG of zero (29)	–9.99999999999E+125
Division by zero (31)	+ or –9.99999999999E+125
X MOD Y, where Y=0 (31)	0

Default values are disabled by executing the DEFAULT OFF statement:

DEFAULT OFF

DEFAULT OFF is set at power on or when any SCRATCH statement is executed.

Built-in Numeric Functions

A function is a routine that manipulates numeric or string data and produces a numeric or string value as a result. A set of commonly used functions, such as one to compute the square root of a number, is supplied as part of the Eloquence language. These functions are known as built-in functions. The built-in functions that perform standard numeric routines are:

ABS (*X*) Absolute value of X.

BACKGROUND Returns 1 if executing in background (stdout redirected or –b switch on commandline). Implies NO OPERATOR. Otherwise returns 0.

DROUND (*X*,*Y*) Returns the value of X rounded to Y number of digits.

EXP (*X*) Naperian e raised to the power X.

FRACT (*X*) Returns the fractional part of X.

INT (*X*) Largest integer %<= X.

LGT (*X*) Logarithm to the base 10 of X; X > 0.

LOG (*X*) Natural logarithm; X > 0.

NO OPERATOR Returns 1 if executing in batch mode (stdin redirected); else 0. E.g., eloq PROG %< infile.

NUMREC *file number* Returns the highest used record number of the file ASSIGNED to a file number. If the file number is ASSIGNED to an HP-UX sequential file, this will result in error 58. If this file is being used as a workfile, it will return the same as WFLEN.

PI Returns the value of π, 3.14159265359.

PID Returns process/parent process id.

PNTR Current ‘PRINTER IS’. Returns current printer number, or:

–3	if PRINTER IS “SPOOLFILE”
–6	if PRINTER IS “STDOUT”
–7	if PRINTER IS “STDERR”
–8	if PRINTER IS “TTY”
–9	if PRINTER IS “CONSOLE”

PPID Returns parent process id.

PROUND (*X*,*Y*) Returns the number X rounded to the power of 10 position specified by Y.

RND Returns a pseudo-random number in a standard sequence of numbers > 0 and < 1.

RANDOMIZE (*start value*) This will reset the (pseudo-) random number generator to a new starting value. If the start value is omitted, random number generator starting value is derived from current system time. If the optional start value is given, it will first be converted to an integer value and then the lower 48 bits of it will be used to initialize random number generation.

SGN (*X*) Returns –1 if X is negative, 0 if X is 0, and 1 if X is positive.

SQR (*X*) Returns the square root of X.

MAX (*list*) Returns the highest value in the list of numeric expressions.

MIN (*list*) Returns the lowest value in the list of numeric expressions.

USRID Returns the user id number. This is calculated using the HP-UX function tty slot(). The following program in C displays this slot # for the terminal where you execute the program:

```
main() {
int ttyslot () ;
printf ("ttyslot: %d\\n",ttyslot());
}
```

TASKID Returns the task id number.

CURKEY Returns the number of the most recently pressed softkey or the value of an ONCONDITION interrupt if TIO is used.

Some functions are available for compatibility reasons, only. To get the syntax of this functions,see section , Built-in Numeric Function, on page 3

NOTE: There is a built-in string function called REVISION$ (not to be confused with REVISION) that returns the version of Eloquence currently running, as a seven-character string. For example, A.06.00.

Built-in numeric functions usually consist of a function name followed by one parameter. The parameter may be a number (as in line 10 below), a numeric variable (line 20), or a numeric expression (line 30). Since the result of a numeric

function is always a single value, a numeric function can be used as an operand in an expression (line 40) or as a parameter of a numeric function (line 50). Here are some examples:

```
10   A=INT(1.6)
```
A equals the integer value of 1.6 = 1.

```
20   B=ABS(A)
```
B equals the absolute value of 1 = 1.

```
30   C=SQR (A+B)
```
C equals the square root of (1 + 1) = 1.41421356237.

```
40   D=LGT(A)^2+LOG(A)^2
```
D equals the sum of LGT(A) squared plus LOG(A) squared (0).

```
50   E=INT(SQR(10))
```
SQR(10) = 3.14; the integer value of 3.14 = 3.

```
60   Max=MAX(1,35,7)
```
Max = the largest value in the list = 35.

```
70   R=PROUND(125.2,2)
```
125.2 rounded to the second power of ten = 100.

```
80   PI=DROUND(PI,4)
```
PI rounded to four digits = 3.142.

The last three lines show built-in functions which allow more than one parameter.

Other built-in functions are available to manipulate strings, to return information on file storage operations, and to control the format of program output. In addition, you may define your own functions as explained later in this chapter.

Trigonometric Statements and Functions

The trigonometric functions use one of three angular units: radians, degrees, or gradients (grads). Radians are the default units (set at power-up or after the computer is reset). To change the angular units, execute one of these statements:

DEG sets degees units. A degree is 1/360th of a circle.

GRAD sets grads units. A grad is 1/400th of a circle.

RAD resets radians units. There are 2(PI) radians in a circle.

The trigonometric functions available are:

ACS (X) returns the principal value of the arccosine of the numeric expression X, which can range from -1 to +1.

ASN (X) returns the principal value of the arcsine o the numeric expression X, which can range from -1 to +1.

ATN (X) returns the principal value of the arctangent of the numeric expression X.

COS (X) returns the cosine of the angle represented by the numeric expression X.

SIN (X) returns the sine of the angle represented by the numeric expression X.

TAN (X) returns the tangent of the angle represented by the numeric expression X.

Here are some examples of trigonometric operations:

```
DEG
FIXED5
DISP SIN(60)
 .86603
COS(45)
 .70711
ASN(.5)
 30.00000
ACS(.5)
 60.00000
ATN(.5)
 26.56505

RAD
FIXED5
DISP SIN(PI/6)
 .50000
COS(PI/6)
 .86603
TAN(PI/4)
 1.00000
ASN(.5)
 .52360
ACS(.5)
 1.04720
ATN(.5)
 .46365

GRAD
FIXED5
DISP SIN(-70)
-.89101
COS(-70)
 .45399
TAN(50)
 1.00000
ASN(.5)
 33.33333
ACS(.5)
 66.66667
ATN(.5)
 29.51672
```

NOTE: To use the SIN function you need to precede it with DISP otherwise it conflicts with *space independent* (SI) and causes an error message.

Random Numbers

A random number generator, the RND function, is provided for programs that perform simulations. Each time the RND function is called, a random number between 0.0 and 1.0 is returned. The following program demonstrates the RND function:

```
10   FIXED 12
20   FOR Line = 1 TO 10
30     PRINT RND,RND,RND,RND
40   NEXT Line
50   END
```

Built-In String Functions

The built-in string functions available are:

CHR$ (*X*) Returns the ASCII character-equivalent of the numeric expression X MOD 256 (a number from 0 through 255). For example, CHR$(65) = "A". ASCII decimal equivalents are listed in page 391 .

ERRMSG$ (error number) Returns error description of given error number. The error messages are located in the message catalog file eloq.cat which will be located at the standard NLS path.

Example:

```
DISP ERRMSG$(2)
"Memory overflow"
```

Error handling:

ERRMSG$ will not bring a program error if either message or message catalog could not be found, to avoid looping in the error handler.

Instead it will return an error message which will contain the error number and a descriptive text where '#' will replaced by the error number parameter:

```
(ERR#): Message catalog not found
(ERR#): Error message not found
```

GETENV$ (*S$*) Returns the value of environment variable.

MAPPNTR$ (printer) Returns printer mapping. This is necessary if you use TIO and you want to configure the device using stty because PORT number may be configured to any devicefile.

Return values:

NONEXISTENT if this printer is not available

INTERNAL if printer is 9 or 10 (and local printer available)

PIPE if printer is mapped to a pipe

	"/dev/" - if printer is mapped to a (device-)file

It may also be used to check for existing printers.

Sample program sequence:

```
COMMAND "!stty 2400 icanon <"&MAPPNTR$(15)
```

This will configure device mapped to PRINTER/PORT 15 to use icanon mode at 2400 baud.

MAPVOL$(Vol-spec$) Returns hp-ux path of mapped volume name or device specifier.

REVISION$ Returns Eloquence revision.

SYSID$ Returns the revision of the used operating system.

RPT$ (*S$,X*) Takes the string S$ and repeats it X times. If X = 0 a null string is returned.

TRIM$ (*S$*) Returns a string which is equal to S$ with all leading and trailing blanks stripped off.

VAL$ (*X*) Converts the value of the numeric expression X to its corresponding string of ASCII digits. For example, VAL$ (65) = "65".

LWC$ (*S$*) Returns a string with all characters converted to lowercase.

UPC$ (*S$*) Returns a string with all characters converted to uppercase.

TYPEOF$(X) Returns the name of the type of the given User Defined variable.

A number, not a string, is returned by the following functions:

LEN (*S$*) Returns the number of ASCII characters in the string S$.

LEX (*S$,T$*) Compares the string S$ with the second string T$ and returns -1 if S$ is less than T$; 0 if S$ equals T$; 1 if S$ is greater than T$.

NUM (*S$*) Returns a numeric value between 0 and 255 corresponding to the first character of the string S$. For example, NUM ("A") = 65.

POS (*S$,T$*) Searches the string S$ for the first occurrence of the string T$. Returns the starting position (index) if found; otherwise returns 0.

SCAN (*S$,T$*) Searches the string S$ for the first occurrence of any single character in the string T$. Returns the starting position (index) if found; otherwise returns 0.

VAL (*S$*) Returns the numeric equivalent of the string S$, which must be a string of ASCII digits. (S$ may include a decimal point.) For example, VAL ("1") = 1. The argument may be any string that begins with a valid number (integer or real). VAL will return the value of this number and ignore any remainder of the string. VAL ("1.2 ABC") will return the value 1.2; VAL ignores the "ABC". If a string begins with anything other than an integer or real, **`ERROR 32`** will result.

Some functions are available for compatibility reasons, only. To get the syntax of this functions, see section , Built-in String Function, on page 4

Most of the string functions are used to manipulate and create strings. For example:

```
10    String$=”REPEAT”
20    PRINT RPT$(String$,10)
30    T$=”      PART NO.      ”
40    PRINT T$;”*”;TRIM$(T$);”*”

REPEATREPEATREPEATREPEATREPEATREPEATREPEATREPEATREPEATREPEAT
     PART NO.      *PART NO.*
                   no leading or trailing blanks here

50    S$=”STRING”
60    PRINT S$,LEN(S$)
70    S$[LEN(S$)+1]=”FUNCTIONS ”
80    PRINT RPT$(LWC$(S$),4)

STRING              6
stringfunctionsstringfunctionsstringfunctionsstringfunctions

90    T$=”TELEVISION”
100   V$=”VISION”
110   PRINT T$,POS(T$,V$)

TELEVISION          5

120   T$[POS(T$,”V”)]=UPC$(”phones”)
130   PRINT T$
140   Invent$=V$T$[POS(T$,”P”)]
150   PRINT Invent$,LWC$(Invent$)
160   END

TELEPHONES
VISIONPHONES        visionphones
```

VAL$ and VAL are used to go back and forth between strings of ASCII digits ("1234") and numeric values. VAL$ results are returned in the current output format (FIXED, FLOAT or STANDARD). For example:

```
10   DIM Pay$[50]
20   FIXED 2
30   Pay$="BASE: 295.50 TAX: 125.30"
40   Takehome=VAL(Pay$[7,12])−VAL(Pay$[19,24])
50   PRINT Pay$, "BUT YOU GET:";Takehome
60   Pay$[25]=" TAKEHOME: "VAL$(Takehome)
70   PRINT LWC$(Pay$)
80   END
```

produces

```
BASE: 295.50 TAX: 125.30            BUT YOU GET: 170.20
base: 295.50 tax: 125.30 takehome: 170.20
```

NUM and CHR$ are used to go back and forth between an ASCII character and its ASCII index. For example, this program tests the numeric value of each character in the string Test$ to see if it contains an integer:

```
10  FOR I = 1 to LEN(Test$)
20    IF NUM(Test$[I]) < 48 OR NUM(Test$[I]) > 57 THEN
30       PRINT VAL(Test$);" is not an integer"
50    END IF
50  NEXT I
60  END
```

Here is a simple program which displays each standard ASCII character and its decimal ASCII code.

```
110  FOR Char=32 TO 127
120    DISP Char;CHR$(Char);SPA(3);
130  NEXT Char
140  END
```

```
RUN
32      33 !    34 “    35 #    36 $    37 %    38 &    39 ’    40 (    41 )
42 *    43 +    44 ,    45 -    46 .    47 /    48 0    49 1    50 2    51 3
52 4    53 5    54 6    55 7    56 8    57 9    58 :    59 ;    60 <    61 =
62 >    63 ?    64 @    65 A    66 B    67 C    68 D    69 E    70 F    71 G
72 H    73 I    74 J    75 K    76 L    77 M    78 N    79 O    80 P    81 Q
82 R    83 S    84 T    85 U    86 V    87 W    88 X    89 Y    90 Z    91 [
92 \    93 ]    94 ^    95 _    96      97 a    98 b    99 c    100 d   101 e
102 f   103 g   104 h   105 i   106 j   107 k   108 l   109 m   110 n   111 o
112 p   113 q   114 r   115 s   116 t   117 u   118 v   119 w   120 x   121 y
122 z   123 {   124 |   125 }   126 ~   127
```

Outputting decimal character values below 32 sends ASCII control codes such as FF (form feed) and CR (carriage return). Sending character values above 127 enables program control of display enhancements and alternate characters on the display. Refer to page 249 for more details.

The SYSID$ Function

The SYSID$ function is a user-definable system identifier. Using the capability of HP 9000s to communicate asynchronously with each other, a user can configure one system's terminal to act as a terminal on another. In a more complex network, the user on the first system may be linked to the second system, then to the third system, and so on. In order to determine which system the user is on, use the system identifier function—SYSID$.

The syntax for the SYSID$ function is as follows:

```
SYSID$
```

The system identifier will return a string of max. 20 characters in length.

NOTE: The Eloquence SYSID$ function returns the same information as the HP-UX uname command. In the HP-UX environment the term "hostname" is used instead of "system identifier". The HP-UX command to display the hostname is as follows:

```
uname -n
```

Defining a Function

When a numeric or string operation has to be evaluated several times, it is convenient to define it as a function. This is done using the DEF FN statement, which specifies a user-defined function and returns a single value as the value of the function. The simplest form is the single-line function, which can be used to define a numeric or string function. To define a numeric single-line function, use this syntax:

DEF FN function name [(formal parameter list)] = numeric expression

To define a string single-line function:

DEF FN function name$ [(formal parameter list)] = string expression

The *function name* must be a valid name (as defined in page 73).

To use the defined function, use the FN statement.

Syntax of the FN statement for a *numeric* function is as follows:

FN function name [(pass parameter list)]

Syntax for the FN statement for a *string* function is as follows:

FN function name$ [(pass parameter list)]

The values of the pass parameters are substituted for the formal parameters and the expression is evaluated. Its value is then returned as the value for the referencing syntax. page 177 provides a detailed explanation of parameters and also covers multi-line functions.

To show the use of single-line functions, consider the following program fragment:

```
30   A=A+(SQR(A)—PI/20)
 .
 .
 .
80   B=B+(SQR(B)—PI/20)
 .
 .
 .
200   C=C+(SQR(C)—PI/20)
```

By defining:

```
20   DEF FNNum(X)=X+(SQR(X)—PI/20)
 function name  formal parameter
```

lines 30, 80, and 200 can be simplified:

```
30    A=FNNum(A)
 .            passed parameter
 .
 .
 .
80    B=FNNum(B)
 .            passed parameter
 .
 .
 .
200    C=FNNum(C)
 .             passed parameter
```

Recursion is not allowed in single line functions.

```
300    DEF FNBad(X,Y,Z)=X+X+FNBad(A,B,C)*3
310    PRINT FNBad(A,B,C)
320    END
```

This would cause **ERROR 48 IN LINE 310**. FNBad must not reference itself either directly or indirectly via another function (in this case, FNBad calls FNWorse which then calls FNBad).

Single-line functions are local to the program segment in which they are defined (see section 7, Subprograms, on page 177 for an explanation of program segments.) For example:

```
10    DEF FNA(X)=PI*X
20    INPUT Z
30    Y=FNA(Z)
40    PRINT Y
50    CALL Set
60    END
70    SUB Set
80      INPUT I
90      J=FNA(I)
100     PRINT J
110    SUBEND
```

Lines 70 through 110 represent a subprogram.

When run, **ERROR 7 IN LINE 90 (SUB Set)** would occur since FNA is not defined in subprogram Set.

Multiple-line function subprograms can also be used to define a function. (see section , Multiple-Line Function Subprograms, on page 184)

Branching and Subroutines

Normal program execution is in sequential order from the lowest numbered line to the highest numbered line. Branching alters this process by transferring control to a statement which is out of the sequential flow. Branching is

just one method of altering the normal flow of program execution. This chapter covers conditional and unconditional branching, looping, subroutines and branching using soft-keys.

The following statements, functions, and commands are discussed in this chapter:

GOTO Branches (unconditionally) to a specified program line.

ON GOTO Branches to one of a list of specified program lines.

IF THEN Branches or executes a statement upon a stated condition.

FOR Defines the beginning of a FOR-NEXT loop.

NEXT Terminates a FOR-NEXT loop.

GOSUB Branches (unconditionally) to a subroutine.

RETURN Terminates a subroutine and returns control to the main program.

ON GOSUB Branches to one of a list of specified subroutines.

SOFTKEYSET ON/OFF Statement to switch softkey set indicator on or off.

ON KEY # Branches to a specified program sequence when the specified special function key is pressed.

OFF KEY # Disables any previous ON KEY # statement for the corresponding key number.

ON ERROR Branches to a specified program sequence when an error occurs.

OFF ERROR Cancels any previous ON ERROR.

DISABLE Prevents ON KEY # declaratives from interrupting program execution.

ENABLE Reactivates ON KEY # declaratives.

CURKEY Returns the value of the most recent interrupting condition, including softkeys, timed delay interrupts, and terminal input/output (TIO) interrupts.

ON HALT Branches to a specified program sequence when BREAK is pressed.

OFF HALT Cancels any previous ON HALT.

INDENT Changes all program line indentation.

ON SIGNAL Branches to a specified program sequence when SIGUSR1 signal is caught. For more detailed description of this and the two following functions,see chapter 14.

OFF SIGNAL Cancel previous ON SIGNAL, see chapter 14.

SEND SIGNAL Send SIGUSR1 signal to specified taskid, see chapter 14.

ON KEYBD Arbitrary keys can be used as function keys.

OFF KEYBD Cancel previous ON KEYBD.

Structured programming techniques improve the programming task via better program organization. A set of enhanced Eloquence statements is available with the Eloquence operating system to improve program readability.

IF THEN ELSE States one of two statements to be executed depending upon the result

of a conditional expression.

WHILE . . . . . .
END WHILE Repeats execution of a block of statements while a conditional expression remains true.

LOOP . . . .
END LOOP Continually repeats a block of statements until a branch out occurs via an EXIT IF statement.

REPEAT . . . . .
UNTIL Repeats execution of a block of statements until a conditional expression becomes true.

SELECT . . . . .
END SELECT Allows branching to any of a set of CASE statements depending on the value of a conditional expression.

Unconditional Branching

The GOTO Statement

The GOTO statement provides unconditional branching by transferring control to a specified line. If the specified line is not an executable statement, control is transferred to the first executable statement following that line.

GOTO line id

Here is an example using GOTO to branch to both higher-numbered and lower-numbered lines:

```
10   Name$="Gwendolyn"
20   GOTO Print
30   INPUT "NEW NAME?";Name$
40   GOTO Compute
50 Print: PRINT "NAME IS:";Name$
60         GOTO 30
70 Compute: ! Continue program.
 .
 .
 .
```

NOTE: A GOTO with line number is not recommened and doesn’t work in the IDE.

The ON GOTO Statement

The ONGOTO (computed GOTO) statement allows control to be transferred to one of a list of statements based on the value of a numeric expression.

ON numeric expression GOTO line id list

The *numeric expression* is evaluated and rounded to an integer. A value of 1 causes control to be transferred to the first statement specified in the list, a value of 2 causes control to be transferred to the second statement specified in the list, and so on. For example:

```
10    INPUT "IS ITEM OVERSTOCKED(1), OK(2),OR OUT OF STOCK(3)?";Status
20    ON Status GOTO 30,Ok,Reorder
30 Over:  ! Overstock routine.
 .
 .
 .
70     STOP
80 Ok:   !
90     PRINT "CHECK ITEM NEXT TIME."
100    STOP
110 Reorder:  ! Reorder routine.
 .
 .
 .
```

If the value of the *numeric expression* is less than 1 or greater than the number of line ids listed, **ERROR 19** (improper value) occurs. For example, when line 130 in the next sequence is executed for the fourth time, the value of I exceeds the number of line ids in the list.

```
120   I=1
130   ON I GOTO Print,Print,Print
140 Print: PRINT "I=";I
150   I=I+1
160   GOTO 130
170   END


I= 1
I= 2
I= 3
ERROR 19 IN LINE 130
```

Conditional Branching

The IF THEN Statement

The IF THEN statement is used to provide branching which is dependent on a specified condition. Syntax for this statement is as follows:

IF numeric expression THEN line id

NOTE: Working with line number is not recommened and doesn't work in the IDE.

If the *numeric expression* has a value other than 0, it is considered true and branching occurs to the specified *line id*. If its value is 0 (false), execution continues with the line following the IF THEN statement. For example:

```
10   INPUT A
20   IF A THEN 50
30   PRINT "A=0"
40   GOTO 10
50   PRINT "A=";A
60   END
```

The IF THEN statement is used most often with relational operators. For example:

```
100   INPUT "HOURS WORKED?";Hours
110   IF Hours>40 THEN Overtime
120   DISP "NO OVERTIME ENTERED."
130   GOTO 100
140  Overtime:   Over=Hours-40
150              PRINT "OVERTIME PAY =";Over*Pay*1.5
160              GOTO 100
```

Another form of the IF THEN statement provides conditional execution of a statement without branching. Syntax is as follows:

IF numeric expression THEN statement

When the value of the *numeric expression* is not 0 (true), the *statement* is executed. When the value of the *numeric expression* is 0 (false), execution continues with the following line. For example:

```
200   READ X,Y
210   IF X=Y THEN PRINT "X=Y"
220   PRINT "X=";X,"Y=";Y
```

A special expression exists for User Defined Types.

IF *Instance_name* IS A *Type_name* THEN

With this expression it is possible to find out if the given Instance has has been derived from the given Type. If the given Instance and the given Type have been derived from the same Base_type, then the expression is true, too.

Example:

```
IF My_car IS A Car THEN CALL Write_car(STRUCT My_car)
```

All *executable* Eloquence statements are allowed following THEN.

The following statements are not allowed after THEN since they are *declaratory* statements, not *executable* statements:

```
COM                    INTEGER
DATA                   OPTION BASE
DIM                    REAL
DEF FN                 REM
FN END                 SHORT
END                    SUB
IMAGE                  SUBEND
```

Here is another example use of IF THEN, which branches to one of many routines depending on the input response.

```
10   INPUT "DO YOU WISH DAILY, WEEKLY, OR MONTHLY REPORTS?";Report$
20   IF UPC$(Report$[1,1])="D" THEN Daily
30   IF UPC$(Report$[1,1])="W" THEN Weekly
40   IF UPC$(Report$[1,1])="M" THEN Monthly
50   DISP "INCORRECT ENTRY"
60   WAIT 2000
70   GOTO 10
80 Daily:  ! Print daily report.
 .
 .
 .
130         STOP
140 Weekly:  !  Print weekly report.
 .
 .
 .
220         STOP
230 Monthly: !  Print monthly report.
 .
 .
 .
300 Continue: ! Continue program.
 .
 .
```

Looping

The FOR and NEXT Statements

Repeatedly executing a series of statements is known as looping. The FOR and NEXT statements are used to enclose a series of statements in a FOR-NEXT loop, allowing them to be repeated a specified number of times. The sequence is as follows:

```
FOR loop counter = initial value TO final value [STEP increment value]
    .
    .
    .
NEXT
```

The FOR statement defines the beginning of the loop and specifies the number of times the loop is to be executed. The *loop counter* must be a simple numeric variable.

The *initial*, *final*, and *increment values* can be any numeric expression. If the *increment value* is not specified, the default value is 1, causing the value to be incremented by 1 each time the loop is repeated.

Here is a simple example:

```
10   FOR I=1 TO 5
20     PRINT I
30   NEXT I
40   PRINT "LOOP DONE, I=";I
50   END
```

```
1
2
3
4
5
LOOP DONE, I=6
```

The variable I is established as the *loop counter* and is set to 1 when the FOR statement is executed. The FOR-NEXT loop is executed 5 times—when I = 1, 2, 3, 4 and 5. Each time the NEXT statement is executed, the value of I is incremented by 1, the default *increment value*. When the value of I exceeds the *final value* (when I = 6) the loop is finished and execution continues with the statement following NEXT.

The advantages of using FOR-NEXT looping instead of an IFTHEN statement are shown in the following examples, where the numbers 1 through 1000 are printed in succession.

IF THEN	FOR-NEXT
`10  Increment = 1`	`100  FOR Increment=1 TO 1000`
`20 Label:PRINT Incremen`	`110   PRINT Increment`
`30  Increment=Increment+1`	`120  NEXT Increment`
`40  IF Increment<=1000 THEN Label`	`130  BEEP`
`50  BEEP`	`140  END`
`60  END`	

The *initial*, *final* and *increment values* are calculated upon entry into the loop. These calculated values are used throughout execution of the loop, and any subsequent alterations to these values will not affect the number of times the loop is repeated.

Here is a simple example:

```
10   A=3
20   INPUT B
30   PRINT "X","A","B"
40   FOR X=A TO A*B STEP B-2
50     A=A+X
60     B=B-1
70     PRINT X,A,B
80   NEXT X
90   END
```

If 4 is input for the value of B, the loop is repeated five times and the output is:

```
X              A              B

3              6              3
5              11             2
7              18             1
9              27             0
11             38             -1
```

The following examples show that differing FOR statements can perform the same task. In each example, the FOR-NEXT loop is executed ten times. Notice the value of the *loop counter* while the loop is executing and after it is complete. An often overlooked aspect of FOR-NEXT looping is that the actual value of the counter when the loop is complete does not equal the final value.

```
10  FOR I=1 TO 10               RUN
20    PRINT I                   1
30  NEXT I                      2
40  PRINT "I=";I                3
50  END                         4
                                5
                                6
                                7
                                8
                                9
                                10
                                I=11
```

```
100  FOR J=10 TO 1 STEP -1      RUN100
110    PRINT J                  10
120  NEXT J                     9
130  PRINT "J=";J               8
140  END                        7
                                6
                                5
                                4
                                3
                                2
                                1
                                J=0
```

```
200  Start=10                         RUN200
210  Finish=40                        10
220  FOR I=Start TO Finish STEP 3.1   13.1
230    PRINT I                        16.2
240  NEXT I                           19.3
250  PRINT "I=";I                     22.4
260  END                              25.5
                                      28.6
                                      31.7
                                      34.8
                                      37.9
                                      I=41
```

```
300   FOR Fraction=.1 TO 1 STEP .1        RUN300
310     PRINT Fraction                     .1
320   NEXT Fraction                        .2
330   PRINT "Fraction=";Fraction           .3
340   END                                  .4
                                           .5
                                           .6
                                           .7
                                           .8
                                           .9
                                           1
                                           Fraction=1.1
```

```
400   FOR A=1 TO 19 STEP 2                 RUN400
410     PRINT A                            1
420   NEXT A                               3
430   PRINT "A=";A                         5
440   END                                  7
                                           9
                                           11
                                           13
                                           15
                                           17
                                           19
                                           A=21
```

If the *initial value* is greater than the *final value* when the loop is entered, the *loop counter* is set to the *initial value* and the loop is skipped. For example:

```
10    PRINT "START, A=";A
20    FOR A=5 TO 1
30      PRINT A
40    NEXT A
50    PRINT "DONE, A=";A
60    END

START, A=0
DONE, A=5
```

Nesting FOR-NEXT Loops

When one FOR-NEXT loop is contained entirely within another, the inner loop is said to be nested. The next example illustrates assigning values to an array using a nested FOR-NEXT loop.

```
10    OPTION BASE 1
20    DIM Array(4,3)
30    FOR L1=1 TO 4
40      FOR L2=1 TO 3
50        Array(L1,L2)=L1+L2
60      NEXT L2
70    NEXT L1
80    PRINT Array(*)
90    END

 2            3             4

 3            4             5

 4            5             6

 5            6             7
```

One FOR-NEXT loop cannot overlap another. For instance:

```
     Correct Nesting                      Incorrect Nesting

10   FOR I=1 TO 10                  100   FOR I=1 TO 10
20     FOR J=1 TO 5                 110     FOR J=1 TO 5
30       PRINT I,J                  120       PRINT I,J
40     NEXT J                       130   NEXT I
50   NEXT I                         140     NEXT J
60   END                            150   END
```

In the incorrect example, the I loop is activated before the J loop is activated. The J loop is cancelled when **`NEXT I`** is executed because it is an inner loop. When the I loop is completed and **`NEXT J`** is accessed, **`ERROR 6 IN LINE 140`** is displayed. This is because the J loop was cancelled and was not reactivated after the last I loop.

When nesting FOR-NEXT loops, do not use the same loop counter variable more than once; therefore, a FOR I loop cannot be nested within another FOR I loop.

FOR-NEXT Loop Considerations

Execution of FOR-NEXT loops should always start with the FOR statement. Branching into the middle of a loop will produce **`ERROR 6`** if NEXT is executed before a corresponding FOR.

Execution of loops normally end with the NEXT statement. It is permissible to transfer control out of the loop by a statement within the loop. After an exit is made through this method, the current value of the counter is retained and is available for later use in the program. In any case, it is permissible to re-enter the loop via the FOR statement, thereby reinitializing the loop counter.

For example, here is a routine which checks each character of each string in a 100-string array Page$. The string is printed unless an * is found (line 30). If an * is found, control exits the inner loop and re-enters the outer loop to avoid printing the string.

```
10    FOR Line=1 TO 100
20      FOR Char=1 TO 50
30        IF Page$(Line)[Char,Char]="*" THEN 60
40      NEXT Char
50      PRINT Page$(Line)
60    NEXT Line
70    END
```

Subroutines

The same sequence of statements may be executed often within a program. A subroutine allows the group of statements to occur only once and yet be accessed from different places in a program segment.

The GOSUB Statement

The GOSUB statement transfers control to a subroutine which begins at a specified statement.

GOSUB line id

A subroutine ends, logically, with **RETURN** which transfers control back to the statement immediately following the GOSUB statement.

The following example shows the use of GOSUB and RETURN:

```
10    X=5
20    Y=3
30    GOSUB Sub
40    X=X+Z
50    Y=Y/2
60    GOSUB Sub
70    Y=Y^3
80    X=0
90    GOTO Continue
100 Sub:! Print value of Z. *
110     Z=X+Y               *     Subroutine
120     PRINT Z             *
130     RETURN              *
 .
 .
200 Continue:!
 .
 .
```

Subroutines may be nested; this is, a second subroutine can be entered before the RETURN of the first is executed. For example:

```
1000  INPUT X,Y
1010  GOSUB Sub1
 .
 .
1090  STOP
1100  Sub1:!
1110    PRINT X,Y
1120    IF XY THEN GOSUB Sub2
 .
 .
1190    RETURN
1200  Sub2:!
1210    PRINT "XY"
 .
 .
1290    RETURN
```

The subroutine Sub2 is nested within subroutine Sub1.

Subroutines can be nested as deeply as available memory allows. When each RETURN is executed, control returns to the previously entered subroutine.

The ON GOSUB Statement

The ON GOSUB (computed GOSUB) statement accesses one of many subroutines, based on the value of a numeric expression.

ON numeric expression GOSUB line id list

The *numeric expression* is evaluated and rounded to an integer. A value of 1 causes the subroutine at the first line id in the list to be accessed, and so on. For example:

```
10   Main:         ! Setup report cycle.
20     INPUT "DO YOU WISH DAILY(1) OR WEEKLY(2) REPORTS?";Cycle
30     ON Cycle GOSUB Daily, Weekly
40     INPUT "SHOULD REPORTS BE DISPLAYED(1), PRINTED(2),OR OTHER(3)?";O
utput
50     ON Output GOSUB Display,Print,Other
 .
 .
110  STOP
120 Daily:            ! Setup daily schedule.
 .
 .
 .
160  RETURN
170 Weekly:           ! Setup weekly schedule.
 .
 .
 .
220  RETURN
230 Display: PRINTER IS 8
240  RETURN
250 Print: PRINTER IS 0
260  RETURN
270 Other:            ! Specify another output device.
 .
 .
 .
```

The main program begins with a series of prompts which the operator answers while setting up a report cycle. Each ON GOSUB branches the program to the appropriate subroutine—Daily, Weekly, Display, Print, or Other.

If the value of the *numeric expression* is less than one or greater than the number of line ids in the list, **`ERROR 19`** occurs.

Branching Using Softkeys

ON KEY # Statement

Within Eloquence there are 24 softkeys. These softkeys are set up in three sets of eight. The softkeys can be used to interrupt a running program and cause branching. This interrupt capability is declared with an ON KEY # statement which specifies the branching operation to occur when the corresponding softkey is pressed.

NOTE: Softkeys can’t be used together with the graphical user interface and so it is not supported on Windows platforms.

ON KEY# key number, [#key number 2 , ... , # key number n] [:"label"]

$$\left\{\begin{array}{l}\text{GOTO } \textit{line id}\\ \text{GOSUB } \textit{line id}\\ \text{CALL } \textit{subprogram}\end{array}\right\}$$

NOTE: Although a subprogram may be CALLed using the ON KEY # statement, no parameters may be passed.

The *key number* is an integer expression from 1 through 24. Multiple keys can be defined within the same ON KEY # statement.

When a key is pressed and an ON KEY # has been declared for it, the specified branch is made after the current line has been executed. The optional *label* is a string expression; the first 18 characters of it appear above the defined softkeys (keys 1 through 8), so labelling the softkeys on the screen.

ON KEY # statements which specify GOTO or GOSUB are active only in the program segment in which they were declared.

NOTE: The eighteenth character of the last function key in each of the three sets (8, 16, and 24) cannot be displayed on some terminal types (for example, the HP 700/92). Therefore, avoid assigning an eighteen character long label to softkeys 8, 16, and 24.

As an example, here is a program sequence which defines some of the softkeys, allowing the operator to select the desired routine:

```
10      OFF KEY #3,4,5,6,7
20      ON KEY #1,#9,#17:"START APPLICATION" GOTO Init
30      ON KEY #2:"ENTER CONFIG" GOTO Config_applic
40      ON KEY #8:"EXIT" GOTO Halt1
50      PRINT PAGE,LIN(10),SPA(20),"APPLICATION STARTUP"
60      PRINT SPA(20), "SELECT OPERATION BELOW"
70      WAIT               ! Wait for softkey.
80 Init:                   ! Start application.
90      ON KEY #1,#9,#17:"ACCOUNTING" GOTO Key1
100     ON KEY #2:"ORDERING" GOTO Key2
110     ON KEY #7:"RESTART" GOTO 10
120     PRINT PAGE,LIN(10),SPA(20),"SELECT MODULE TO BE
        STARTED BELOW"
130     WAIT               ! Wait for softkey.
140 Config_applic:         ! Enter application parameter.
  .
  .
  .
```

Line 10 cancels any previous ON KEY # definitions for keys 3 through 7. Lines 20 through 40 define the keys for initial selection. The program then displays instructions and waits at line 70 until a defined softkey is pressed.

The Init routine, lines 80 through 130, show how softkey 1 and softkey 2 are redefined and how softkey 7 is defined to restart the program. The previous definition for softkey 8 remains in effect here. Another WAIT statement holds program execution until a defined softkey is pressed.

The OFF KEY # Statement

The ON KEY # declarative is in effect for a key until another declarative for the same key or a SCRATCH, STOP, END, RUN, or OFF KEY # statement is executed:

OFF KEY # [key number list]

Omitting key numbers causes all ON KEY # definitions to be cancelled. See line 20 in the preceding example program.

The DISABLE Statement

The DISABLE statement prevents :ON KEY # declaratives from interrupting program execution.

DISABLE

The ON KEY # declaratives are still active; pressing each softkey causes the interrupt to be logged for execution when the declaratives are re-enabled.

The ENABLE Statement

To re-enable ON KEY # declaratives to interrupt program execution, execute the ENABLE statement:

ENABLE

The CURKEY Function

The CURKEY function returns the value of the most recent interrupting condition, including softkeys, timed delay interrupts, and terminal input/output (TIO) interrupts:

The possible values returned by CURKEY, and their meaning, are given here:

0	No interrupt.
1 - 24	Softkeys 1 through 24.
25 - 53	TIO interrupts: (*port number* * 3)+25 = ON INPUT number
54	ON DELAY
55	ON KEYBD
56 - 74	Unused.

When an ON KEY #GOTO causes a branch, any previous value for CURKEY is lost. When ON KEY #GOSUB or ON KEY #CALL causes a branch, any previous CURKEY value is retained on entry to the subroutine or subprogram. While in the subroutine or subprogram, CURKEY returns the number of the key used to call the routine. The previous number is returned when the routine is exited.

The CURKEY value is cleared by any SCRATCH operation and by pre-run initialization (RUN, GET, etc.).

Here is a simple program which defines the softkeys 1 through 8 and then displays each softkey number pressed.

```
10   FOR Key=1 TO 8
11     A$=VAL$(Key)
20     ON KEY #Key:A$ GOSUB 50
30   NEXT Key
40   WAIT
50   DISP "KEY";CURKEY;"PRESSED"
60   RETURN
```

LASTKEY

The LASTKEY keyword returns the key number of the last key that caused an ON KEYBD interrupt.

SOFTKEY ON/OFF Statement

The SOFTKEYSET ON or SOFTKEYSET OFF statements will switch the softkey set indicator on or off. The default state is off. The softkey indicator appears at the right margin above the softkey labels.

SOFTKEYSET ON

SOFTKEY SET OFF

U1 - U3 user keys (ON KEY #) S1 - S2 system

The SOFTKEYSET statement will switch softkeys to the level given by the SOFTKEYSET expression:

1	program keys #1 - #8
2	program keys #9 - 16#
3	program keys #17 - #24
-1	system keys set #1
-2	system keys set #2
-3	system keys set #3

Error Testing and Recovery

Run-time errors are those which occur when a program is running. Dividing by 0 is an example. These errors normally halt execution. Through use of the ON ERROR statement, run-time errors need not abort the program. Execution may continue with specified code following the execution of the line in which the error occurred. The ON ERROR statement causes a branch which takes place after any error.

ON ERROR { GOTO *line id* | GOSUB *line id* | CALL *subprogram name** }

* Parameter can not be passed

The ON ERROR statement declares what should happen if an error occurs. It need be executed only once in each program segment to establish the ON ERROR condition. Execution of another ON ERROR statement cancels the previous one.

When a run-time error occurs and the ON ERROR condition has been established, execution is transferred to the specified line. Then the ERRN and ERRL functions discussed next can be tested, and either error recovery procedures or DISP ERRM$ can be executed. The error is ignored if the statement referenced by a GOSUB is a RETURN statement; execution continues with the line after the one in which the error occurred.

If the error-recovery routine itself contains an error, the program may possibly run in an endless loop. This can be stopped by pressing BREAK or CTRL Y.

If the ON ERROR statement specifies a GOSUB or CALL, computer priority is set at the highest level until the routine has been completed. The routine can be interrupted only by an ON END or another ON ERROR interrupt. (ON END is described in page 195 .) A routine accessed with GOTO can be interrupted because system priority is not changed.

The following string and numeric functions return information related to the last error trapped with ON ERROR:

ERRL The error line function returns the line number in which the most recent program execution error occurred.

ERRN The error number function returns the number of the most recent program execution error.

ERRM$ The error message string returns the most recent program execution error message.

ERRMSG$(ERRN) The error message, which belongs to the error number, is read from the message catalog and will be displayed.

ON ERROR is disabled with the OFF ERROR statement:

OFF ERROR

The following program sequence shows how ON ERROR can be used to detect errors and display an appropriate message:

```
10 OPTION BASE 1
20 DIM A(50)
30 ON ERROR GOTO Recovery1
40 Print: INPUT "Enter file name:",F$
50        ASSIGN #1 TO "F$"
60        FOR R=1 TO 50
70        PRINT #1,R;A(R)
80        NEXT R
90        PRINT "DATA PRINTED ON FILE."
100       GOTO Read
110 Recovery1:    ! Check for errors 53 and 56.
120         IF ERRN=53 THEN E53
130         IF ERRN=56 THEN E56
140         GOTO Exit
150 E53:  PRINT PAGE;"IMPROPER FILE NAME - PRESS <RETURN> TO CONTINUE."
160       INPUT
170       GOTO Print
180 E56:  PRINT PAGE;"FILE NAME IS UNDEFINED -
 PRESS <RETURN> TO CONTINUE."
190       INPUT
200       GOTO Print
210 Exit: PRINT "ERRM$"   ! Print error number and line.
220       STOP
230 Read: ! Continue program.
 .
 .
 .
```

Line 30 activates an ON ERROR condition which will branch the program to the Recovery1 routine if an error occurs. The Print routine prints data elements of array A into sequential records of a data file. If any error occurs here, the program would branch to Recovery1, rather than print **DATA PRINTED ON FILE** and continue at the Read routine.

The Read routine begins by deactivating the first ON ERROR routine and activating a new one.

The Recovery1 routine checks for errors 53 and 56. The routine responds to errors 53 and 56 with a displayed message and then returns to the Print routine where the operator can correct the error.

Notice that if any error but 53 or 56 occurs while Recovery1 is active, the Exit routine first displays ERRM$, containing the error number and line, and then stops.

The ON HALT Statement

The ON HALT statement sets up a branching condition which will occur if the BREAK key is pressed.

ON HALT {
GOTO *line id*
GOSUB *line id*
CALL *subprogram name**
}

* Parameters cannot be passed between calling program and subprogram.

The branch occurs immediately after the current program line is executed.

Here is an example sequence which checks for the BREAK key and branches to a routine to store the contents of an array in a data file before stopping the program:

```
10   ON HALT GOTO Stop
 .
 .
240  STOP
250 Stop: ! Save data  exit program.
260    ASSIGN #1 TO "SaveAray"
270    PRINT #1;Array(*)
280    DISP LIN(10);"PROGRAM HALTED"
290    END
```

The ON HALT condition is cancelled after SCRATCH, STOP, END or RUN. The condition is only active while the program is running after ON HALT is executed and during an INPUT state, but it is temporarily deactivated during a PAUSE.

To cancel any previous ON HALT condition, use the OFF HALT statement:

OFF HALT

The KEYBD function

NOTE: The KEYBD function is not available when working with the graphical user interface and so it is not supported on Windows platforms.

ON KEYBD

The ON KEYBD statement makes it possible to use arbitrary keys like functions keys.

Syntax:

ON KEYBD #n[,#n...][,priority] {GOTO|GOSUB|CALL} target

NOTE: Key numbers are defined by curses and returned by the KBCODE keyword. You can't catch the break key (key number 0).
There is a maximum of 32 active ON KEYBD in one program segment
An ON KEYBD statement overrides any default sense of the key. Catching an important key (like carriage return) may result in a unusable keyboard within Eloquence

NOTE: A lot of special keys makes it hard to remember for the user, which keys are active or what keys result in what function. It's most likely a bad idea to catch any regular characters or control characters. If you catch some special keys (for example next line), the program reaction should be closely implied by the key.

OFF KEYBD

The OFF KEYBD# removes the interrupt handling for the specified key or all keys, if no key number has been specified.

Syntax:

OFF KEYBD #[n [,n . . .]]

Structured Programming

Structured programming techniques improve the programming task via better program organization.

Structured IF THEN ELSE

The structured IF statement allows program execution to resume at one of two points depending on the result of a test expression. The syntax for the structured IF is as follows:

IF conditional expression THEN
(statements)
.
.
.
ELSE
(statements)
.
.
.
END IF

The ELSE statement is optional. If left out and the conditional expression is false, execution is passed to the statement after END IF. If a branch occurs into the statements immediately after IF, program execution continues after the END IF when the ELSE is encountered. When an END IF is required but cannot be found, an error occurs.

Here is an example sequence from a text processing program. It checks the first character of string Line$. If it contains a period, the statements immediately following IFTHEN determine the exact string and branch to perform the appropriate function. If the first character is not a period, the ELSE statements are executed.

```
200  IF Line$[1,1]="." THEN         ! Check for command line.
210    IF Line$[1,3]=".BR" THEN Break
220    IF Line$[1,3]=".PA" THEN Page
230    IF Line$[1,3]=".AP" THEN Append
240  ELSE
250    PRINT Line$                  ! Print text line.
260    Printpos=Printpos+1
270  END IF
      .
      .
      .
```

The WHILE Block

The WHILE . . . END WHILE statements allow repeated execution of a series of statements while a conditional expression remains true. This condition is evaluated at the beginning of the loop. If the expression is initially false, the loop is bypassed without ever being executed.

The syntax for the block is as follows:

WHILE conditional expression
(statements)
.
.
.
END WHILE

If control is transferred into the loop via GOTO or other non-structured construct, an error results when the END WHILE is encountered. If an END WHILE cannot be found when required, an error occurs.

Here is a sequence which reads and prints numbers in successive records of a disk file. Control exits the block when the TYP function returns 3, indicating that the end of file is reached.

```
10   ASSIGN #1 TO "Parts, SYSTEM"
20   WHILE TYP(1)3             !Exit at end of file.
30     READ #1:Part$
40     PRINT Part$
50   END WHILE
60   END
```

The LOOP Block

The LOOP block allows repeated execution of a series of statements until an explicit request is made to exit. The statements between LOOP and END LOOP are executed until an EXIT IF statement terminates the loop. The syntax for the block is as follows:

LOOP
(statements)
.
.
.
END LOOP

The loop may be exited only by an EXIT IF statement.

EXIT IF conditional expression

If the condition is true, the loop is exited. If control is transferred into the loop via GOTO or another non-structured construct, an error results when the END LOOP is encountered.

Note in the next example that EXIT IF exits the inner-most structured block. Other structured blocks active within LOOP (for example, REPEATs, WHILEs, or FORs) are deactivated by removal from the system stack.

```
10  DIM Ltr$[1],Char$[1]
20  INTEGER Xpos,Badguess,Goodguess
30  DATA 31,T,33,E,35,L,37,O,40,B,42,O,43,C
40  DISP " Guess the statement by entering one letter at a time:",LIN(2)
50  DISP SPA(30);"_H_ _O_P _LK"
60  RESTORE
70  READ Xpos,Ltr$
```

```
80   LOOP
90     CURSOR (Xpos,4)                    !  Put cursor at current blank.
100    INPUT "";Char$[1;1]
110    IF Char$=Ltr$ THEN
120      DISP " Good Guess!"
130      WAIT 1000
140      DISP "    "
150      EXIT IF Ltr$="C"                 ! Exit loop if last guess is C.
160      READ Xpos,Ltr$
170      Goodguess=Goodguess+1
180    ELSE
190      BEEP
200      DISP "Wrong letter ... try again."
210      WAIT 1000
220      DISP "    "                      ! Clear message line.
230      CURSOR (Xpos,4)
240      Badguess=Badguess+1
250    END IF
260  END LOOP
270  DISP "You guessed it within";Goodguess+Badguess;"tries,";
280  DISP "and";Badguess;"incorrect guesses!"
290  END
```

The REPEAT Block

The REPEAT and UNTIL statements allow repeated execution of a series of statements until a certain condition is true. This condition is evaluated at the end of the loop. The loop is always executed at least once. The syntax for the block is as follows:

REPEAT
(statements)
.
.
.
UNTIL conditional expression

If control is transferred into the loop via a GOTO or other non-structured construct, an error results when UNTIL is encountered.

The next sequence repeats lines 50 and 60 until X=Xsqr.

```
10    INPUT "Enter an integer >1:";X
20    Xsqr=X*X
30    DISP "Sum of";X;"through";Xsqr;"is:";
40    REPEAT
50      Sum=Sum+X
60      X=X+1
70    UNTIL X=Xsqr
80    DISP Sum
90    END
```

The SELECT Block

The SELECT block allows any of a variety of blocks of statements to be executed depending on the value of a selection expression. The syntax of the SELECT block is as follows:

SELECT string or integer selection expression
CASE case list
(statements)
.
.
.
CASE ELSE
(statements)
.
.
END SELECT

The *case list* is a list of case items separated by commas. A case item is defined as follows:

$$\left\{\begin{matrix}\text{Constant}\\ \textit{"string"}\end{matrix}\right\} \qquad [\text{TO}] \qquad \left\{\begin{matrix}\text{Constant}\\ \textit{"string"}\end{matrix}\right\}$$

or

$$\begin{bmatrix} < \\ > \\ <> \end{bmatrix} \left\{\begin{matrix}\text{constant}\\ \text{"string"}\end{matrix}\right\}$$

Any number of CASE statements may be used. CASE ELSE is optional. At syntax time, all constants in the *case list* are verified to be either type string or integer. At execution time, a check is made to verify that the type of selection expression matches the types in the CASE statements. Some example CASE statements are shown here:

```
100 CASE 1
100 CASE <5
100 CASE >59
100 CASE 1 TO 10
100 CASE "A" TO "J"
```

If the *selection expression* matches the ranges specified in any of the CASEs, the statement block following that CASE is executed. If no CASE is matched, the statement block following the CASE ELSE is executed. If there is no CASE ELSE, the SELECT block is entirely bypassed. Any CASEs following the first CASE ELSE are ignored.

If control is passed into the SELECT block via GOTO or other non-structured construct, the first CASE or CASE ELSE encountered causes control to transfer to the line following END SELECT. Statements following the SELECT, but preceding the first CASE or CASE ELSE, are not executed unless control is specifically passed to them via GOTO or another non-structured construct.

The next program sequence traps many errors typically encountered in a text processing program. The SELECT BLOCK defines a display message to explain each error. The Disperr routine inserts the error message at the current cursor position in test.

```
1000 Err_msgs:    !Display error message and wait for ENTER key.
1010     BEEP
1020     SELECT ERRN
1030     CASE 18
1040       Line1$="INPUT LINE IS TOO LONG."
1050       Line2$="HIT ENTER, EDIT AND RE-INPUT THE LINE."
1060     CASE 53
1070       Line1$="IMPROPER FILE NAME FORMAT."
1080       Line2$="HIT ENTER AND RE-SPECIFY FILE NAME."
1090     CASE 56
1100       Line1$="FILE NAME IS UNDEFINED."
1110       Line2$="HIT ENTER AND RE-SPECIFY FILE NAME."
1120     CASE 132 TO 134
1130       Line1$="PRINTER OFF-LINE OR SWITCHED OFF."
1140       Line2$="READY PRINTER AND PRESS ENTER TO CONTINUE."
1150     CASE ELSE          ! Other errors trapped here.
1160       Line1$=ERRM$
1170       Line2$="CALL SYSTEM MANAGER FOR HELP."
1180       WAIT              ! Wait for softkey.
1190     END SELECT
1200     GOSUB Disperr
1210     IF (CURKEY=5) OR (CURKEY=6) THEN RETURN
1220     GOTO Start
1230 Disperr:   !
1240     Ypos=YPOS
1250     CURSOR (1,Ypos)
1260     DISP "  "      ! Make room for message in text.
1270     CURSOR (1,Ypos),IV(80)
1280     DISP Line1$
1290     DISP Line2$
1300     INPUT          ! Wait for ENTER key.
1310     CURSOR (1,Ypos)
1320     DISP "  "      ! Delete message.
1330     CURSOR (1,Ypos)
1340     RETURN
```

7

Subprograms

Programs developed for business applications such as company payroll or inventory control can easily contain hundreds of statements. A large program becomes easier to develop, to debug, and to document if it is divided into several program segments, each of which performs a single task. The term **program segment** may refer to either a main program or a subprogram.

A **subprogram** is a group of one or more statements that performs a certain task under the control of the calling program segment. The machine runs each main program and each subprogram independently of each other. The program segment which is currently being executed is called the current environment.

A subprogram enables you to repeat an operation many times, substituting different values each time the subprogram is called. Subprograms can be called at almost any point in a program and are convenient and easy to use. In addition to giving greater structure and independence to your programming, subprograms may conserve memory through the use of local variables and dynamic memory allocation.

There are two types of subprograms—multiple-line function subprograms and subroutine subprograms. The **multiple-line function subprogram** is designed to return a value to the calling program, and is used like a built-in function such as SGN or CHR$. It is defined using the DEF FN statement. A **subroutine subprogram** is designed to perform a specific task. It is defined using the SUB statement. Subprograms are separate program segments located after the main program in memory.

The statements described in this chapter are:

CALL	Accesses a subroutine subprogram.
DEF FN	Defines multiple-line function subprograms (single line functions are covered in page 125).
FNEND	Declares end of a multiple-line function subprogram.
RETURN	Returns control and a value to the calling program segment.
SUB	Defines a subroutine subprogram.
SUBEND	Terminates a subprogram and returns control to the calling program segment.
SUBEXIT	Returns control from a subroutine subprogram before SUBEND.
DEL SUB and DEL FN	Delete entire subprograms or function programs from memory.

Subprograms are recorded into disk files, with their accompanying main programs, by using the STORE or SAVE statements described in page 195 . The LOAD SUB statement is available to bring one or more STOREd subprograms back into memory. See page 195 for details.

Parameters

Values are passed between a subprogram and the calling program segment using **parameter lists**. There are two kinds of parameters—formal and pass. **Formal parameters** are used to define the subprogram. **Pass parameters** are used to pass values from the calling program segment to the subprogram. Each pass parameter must correspond to a formal parameter.

The formal parameter list is used to define the subprogram variables and relate them to calling program variables. In addition, the parameter list includes non-subscripted numeric and string variable names, array identifiers (an array name followed by (*) specifies use of the entire array), User Defined Types(see section , User defined Types, on page 93) and file numbers (see page 195). Parameters must be separated by commas, and the parameter list must be enclosed in parentheses.

Numeric types REAL, SHORT, INTEGER and DINTEGER can be declared in a formal parameter list by placing the keyword before either a parameter or group of parameters. For example:

```
10   SUB X(A,B$,INTEGER C(*),D,SHORT E,F,#3,G)
```

The array C and simple variable D are declared as integer precision; E, F and G are short precision; and A is real precision. The #3 parameter refers to data file number 3.

NOTE: Instead of declaring a formal parameter to be a REAL, SHORT, INTEGER or DINTEGER, you could declare it as NUMERIC. Then it will be one of the passed type, i.e. if passed an INTEGER, it returns an INTEGER, if passed a REAL, it returns a REAL.

When calling a subprogram or function with a STRUCT argument, you can pass an object either anonymously or specify a type name.

CALL *Sub*(STRUCT *A*)

. . .

SUB *Sub*(*A* {AS|:} *TypeName*)

SUB *Sub*(STRUCT *A*)

Like regular variables, types can either be passed by value or by reference. When passed by value, a copy is passed to the subroutine.

For example:

```
   TYPE Type
    INTEGER I
   END TYPE
!
   DIM Inst:Type
   Inst.I=0
   CALL Sub(Inst)
   PRINT Inst.I
   CALL Sub((Inst))
   PRINT Inst.I
   STOP
!
   SUB Sub(STRUCT A)
      A.I=A.I+1
   SUBEND
```

A STRUCT can be passed to a subprogram or function in two manners:

Instances can be passed to a SUBroutine either anonymous or with an associated type.

- When no type is specified in the SUBprogram or function definition, the struct is passed "anonymously" as an argument. Any type is valid and no validation is performed on the argument. RTTI (TYPEOF$, IS A) can be used to operate on it.
- When a type is specified in the SUBprogram or function definition, only variables of the given (or derived) type are accepted, else a runtime error 8 is issued. The type must have been exported previously.

For example:

```
CALL X(STRUCT X)
SUB X(STRUCT Any)              ! Anonymous


CALL Y(STRUCT X)
SUB Y(STRUCT Known:Tknown)     ! Passed value must be of type Tknown
                              ! or derived from it. Tknown must have
                               ! been definined previously
```

The pass parameter list used in calling the subprogram can include numeric and string variable names, array elements and identifiers, numeric and string expressions, user defined types and data file numbers. The pass parameter list must also be enclosed in parentheses.

Parameters must be separated by commas. All arrays in the pass parameter list must be defined within the calling program segment.

When a subprogram is called, each formal parameter corresponds to, and is assigned, the value of the pass parameter which is in the corresponding position in the pass parameter list. The parameter lists must have the same number of parameters, and the parameters must match in type—numeric precision or string, simple or array, or file. Notice that numeric types must match in precision—real, integer, or short.

Notice the correspondence between pass and formal parameters in the next example:

```
20   INTEGER C(2,2),D(2,2)
30   CALL X(A,B$,C(*),(D(1,2)),3,E+F,#6,(G))
.
.
.
70   CALL X(5,(C$[1,12]),D(*),4,(X(4,3)),(A),#2,E*3)
.
.
.
120  SUB X(X,Y$,INTEGER Z(*),SHORT K,L,M,#9,N)
```

Parameters are passed either by reference or by value. When a parameter is passed by reference, the corresponding formal parameter shares the same memory area with the pass parameter. Thus, changing the value of the variable in the subprogram also changes the value of the variable in the calling program. Arrays are always passed by reference.

When a parameter is passed by value, the variable defined by the corresponding formal parameter is assigned the value of the pass parameter and given temporary storage space in memory. Numeric and string expressions are necessarily passed by value. Enclosing a pass parameter in parentheses causes it to be considered an expression and thus passed by value, rather than by reference. Passing by value prevents the value of a calling program variable from being changed within a subprogram.

In the following example, all parameters in line 200 are passed by value, while those in line 250 are passed by reference:

```
200   CALL Active(Y+3,(X(2,4)),(X(1,4)),.5,(Y),(Line$[5,8]))
.
.
.
250   CALL Active(Y,Data(2,4),X(1,4),A,Z,Line$)
.
.
.
290   SUB Active(A,B,C,D,E,F$)
```

Any parameters passed by value are converted, if necessary, to the numeric type of the corresponding parameter in the formal parameter list. For example, say that PI is passed by value to an INTEGER formal parameter. Its value would be rounded to 3 when the subprogram is called.

Those passed by reference must match exactly, otherwise **`Error 8`** occurs. No conversion is made.

For instance:

```
10   DIM C(3,3),D(3,3)
20   CALL X(A,B$,C(*),D(1,2),3,E,#5,G)
30   END
40   SUB X(X,Y$,INTEGER Z(*),SHORT K,L,M,#3,N)
50   SUBEND
60   END

RUN
ERROR 8 IN LINE 20
```

NUMERIC is used in parameter passing to subroutines and functions. NUMERIC will bypass type checking for parameters and will match any numeric data type.

Example program:

```
REAL R
INTEGER I

CALL Val(A$,R)
CALL Val(A$,I)

SUB Val(A$, NUMERIC V)
   ON ERROR GOTO E
   V=VAL(A$)
   SUBEXIT
E: V=0
SUBEND
```

Multiple-Line Function Subprograms

The multiple-line function subprogram is used to define a numeric or string function and return a value to the calling program segment. The first line of a numeric multiple-line function subprogram is:

DEF FN subprogram name [(formal parameter list)]

For a string function, the first line is:

DEF FN subprogram name$ [(formal parameter list)]

The subprogram name must be a valid name.

The last line in a multiple-line function subprogram should be:

FNEND

The value to be returned to the calling program segment as the value of the function is specified by:

RETURN numeric expression

or

RETURN string expression

The function subprogram is called automatically by specifying the function name and pass parameter list in a program line:

FN subprogram name [(pass parameter list)]

or

FN subprogram name$ [(pass parameter list)]

Here is an example of a numeric function:

```
10   DIM C(50)
20   A=10
30   B=20
40   FOR I=0 TO 50
50     C(I)=I
60   NEXT I
70   X=FNTotal(A,B,C(*))
80   PRINT "RESULT=";X
 .
 .
120  END
130  DEF FNTotal(X,Y,Z(*))
```

```
140     Tot=0
150     FOR I=X TO Y
160       Tot=Tot+Z(I)
170     NEXT I
180     RETURN Tot
190     FNEND

RUN
RESULT= 165
```

The function subprogram computes the value of **`Tot`** using the equation:

$$\text{Tot} = \sum_{I=X}^{Y} Z(I)$$

Notice that the variable X in the function subprogram is not the same as X in the main program.

Here is an example of a string function:

```
10    A$="HELLO"
20    B$="GOODBYE"
30    Rep=2
40    PRINT FNResult$(A$,B$)
50    END
60    DEF FNResult$(H$,G$)
70       String$=H$"..."G$
80       RETURN "****"TRIM$(String$)"****"
90       FNEND

RUN
****HELLO...GOODBYE****
```

There can be more than one RETURN statement in a function subprogram, but only one is executed each time the subprogram is executed. Here is an example based on the previous numeric function subprogram:

```
10    DIM C(2,2)
20    C(0,0)=C(0,1)=C(1,0)=C(1,1)=2
30    A=B=4
40    PRINT "RESULT=";FNTotals(A,B,C(*))
50    END
60    DEF FNTotals(X,Y,Z(*))
70      A=Z(0,0)+Z(0,1)+Z(1,0)+Z(1,1)
80      B=X+Y+A
90      IF XY THEN RETURN B
100     RETURN 2*B
110     FNEND

RUN
RESULT= 32
```

If a single-line and multiple-line function are defined with the same name and the name is referenced, the single-line function is the one that is accessed if it is defined in the calling program segment.

Subroutine Subprograms

Subroutine subprograms allow you to repeat a series of operations many times using different values or break a large problem down into a series of smaller ones. A subroutine subprogram performs a specific task. It consists of one or more statements following the SUB statement, which is the first statement in a subroutine subprogram. Syntax for the SUB statement is as follows:

SUB subprogram name [(formal parameter list)]

The *subprogram name* must be a valid name.

The last line in a subroutine subprogram should be:

SUBEND

This returns control back to the calling program segment.

The subroutine subprogram is accessed and values supplied by the CALL statement. Syntax for this statement is as follows:

CALL subprogram name [(pass parameter list)]

Here is a simple example of a subroutine subprogram used to write a heading for data output:

```
10   CALL Header
 .
 .
 .
200  END
210  SUB Header
220    PRINT TAB(11),"NAME",TAB(30),"CURRENT SALARY"
230    SUBEND
```

Here is a more complex example which outputs a readable table when values are supplied:

```
10   CALL Table(Dept,Total,C(*),Super$)
 .
 .
 .
40   END
50   SUB Table(Dept,Total,C(*),Super$)
60     PRINT "DEPARTMENT NUMBER:";Dept,"SUPERVISOR:";Super$,LIN(2)
70     PRINT "PRODUCT NUMBER","% OF TOTAL SALES",LIN(2)
80     FOR I=1 TO 60
90       PRINT SPA(5);C(1,I),SPA(10);C(2,I)/Total
100    NEXT I
110    SUBEND
```

The SUBEXIT statement is used to transfer control back to the calling program segment before SUBEND is executed.

For example:

```
120   SUB Pay(X,Y)
 .
 .
 .
160       IF XY THEN SUBEXIT
 .
 .
 .
200       SUBEND
```

Subprogram Considerations

Temporary Default States

The computer enters a new operating environment when entering each subroutine or function subprogram. The current environment is suspended until exiting the subprogram. The following default states are set when a subprogram environment is entered:

- Any READ statements in the subprogram refer only to DATA lists within that subprogram.
- Any assigned file numbers not passed in a COM statement or a parameter list are not accessible by the subprogram (they remain open).
- RAD, STANDARD, and OPTION BASE 0 modes are set.
- Any ON KEY#, ON HALT, ON ERROR, ON SIGNAL, ON DELAY, ON INPUT# associated with a GOTO or GOSUB is no longer active; ON KEY#, ON SIGNAL, ON DELAY, ON INPUT# interrupts, however, may be logged for processing upon return to the calling program.
- All ON END declaratives are deactivated.

Upon return to the main program, all of the above are restored to their previous states.

Adding and Deleting Subprograms

There are two ways to add a new subprogram to a program. It may either replace an existing subprogram or come after all other subprograms.

In order to delete the first line of a subprogram, the SUB or DEF FN statement, the entire subprogram must be deleted. To delete an entire subprogram, use the DEL SUB or DEL FN statements:

DEL SUB subprogram name [TO END]

or

DEL FN function name [$] [TO END]

Each statement deletes the named subprogram or function from memory. If TO END is specified, all successive subprograms and functions are also deleted.

The SUB statement can be edited as long as it remains a SUB statement or is changed to a DEF FN.

Using COM Statements

Values can also be passed to a subprogram with a COM statement. The list of items in the subprogram COM must be a subset of the main program COM statement; that is, it must match to some point in the main program COM.

Here are some valid examples:

```
20    COM A(4,4),B,INTEGER C,D(3,3),E$[28],F$(2,4)[56]
 .
 .
 .
150   END
160   SUB Payroll
170      OPTION BASE 1
180      COM X(*),Y,INTEGER Z,Q(1:3,1:3)
 .
 .
 .
220      SUBEND
230   DEF FNAccounts(X,Y,Z)
240      COM I(1:4,1:4)
```

Here is an invalid example using the same main program COM statement:

```
300    SUB Total
310      COM M(4,4),N
 .
 .
 .
 .
350    SUB Price
360      OPTION BASE 1                 There is no item correspondin
g
370      COM L(4,4),M,Q(3,3)           to array Q, causing error 47.
```

Arrays can be specified in a subprogram COM statement using an array identifier (see line 180 above). A variable cannot be an item in a subprogram COM statement, however, if it is also a formal parameter. For example, if the following is executed, **`Error 12`** occurs:

```
400    SUB Sub(X,Y,Z$)
410      COM A(2,2),X
```

Subprograms may also consist of any of the other variable-declarative statements—DIM, REAL, SHORT, INTEGER and DINTEGER. The variables declared, however, may not be in the subprogram COM statement or the formal parameter list.

Here is a valid example:

```
450   SUB X(X,Y(*),Z$,A)
460     COM B(3,3),C$[20],D,SHORT E
470     DIM F(5,2),G$(2,2)[50]
480     SHORT H,I(3,7,2)
```

Here is an *invalid* example:

```
520   DEF FNTaxes(A,B(*),C$,D)
530     COM E(3,4),INTEGER F
540     DIM C$[20],E(2,2)
```

C$ and **E** were already defined, so **ERROR 12 IN LINE 540 WITH C$** will occur.

All variables in a subprogram that are not part of the formal parameter list or the COM statement are known as local variables and cannot be accessed from any other program segment. Storage of local variables is temporary; the memory space is returned to user read/write memory upon return to the calling program. All variable names in a subprogram are independent of variables with the same name in other program segments or other subprograms.

File numbers can be passed to a subprogram in the parameter list. For example, the following statements assign **DATA** to file number 3:

```
10   ASSIGN #1 TO "DATA"
20   CALL Routine(#1)
 .
 .
 .
60   SUB Routine(#3)
```

Any operations, such as PRINT#, which involve file #3 in the above subprogram will affect file #1 (**DATA**) in the calling program. The data pointers in file #1 are maintained in file #3; then the status of file #3 is passed to file #1, and vice versa, when the subroutine is exited.

```
100   CALL X(#4)
 .
 .
 .
140   SUB X(#2)
150      ASSIGN #2 TO "Payroll"
```

When control returns to the calling program, file #4 is still assigned to the file Payroll.

A file can also be implicitly buffered in this manner:

```
200   CALL Data(#4)
 .
 .
 .
 .
240   SUB Data(#2)
250      ASSIGN #2 TO "Payroll"
```

When control returns to the calling program, file #4 is still assigned to Payroll and still buffered.

If a file is opened in a subprogram, but not passed as a parameter or in COM, it is automatically closed upon return to the calling program. See page 195 for an explanation of ASSIGN.

Note that if a variable in a COM declaration is wrong, any variables which follow will also be disregarded.

Busy Lines

When a subprogram is accessed from a calling program, a condition is created known as a busy line or a busy subprogram.

Here is an example of a busy line:

```
10    A=FNX(B)
 .
 .
 .
50    DEF FNX(D)
 .
 .
 .
90    FNEND
```

Line 10 is busy after the subprogram at line 50 is accessed and remains busy until FNEND is executed.

Here is an example of a busy program:

```
100   CALL X(A,B,C)
 .
 .
 .
140   SUB X(X,Y,Z)
150   CALL You
 .
 .
 .
190   SUBEXIT
200   SUB You
 .
 .
 .
240   SUBEXIT
```

The subprogram X at line 140 becomes busy when it is called (line 100). The subprogram remains busy until it is exited.

Busy lines and subprograms can have an effect when editing a running program or executing LINK, DEL SUB, or DEL FN. Attempting to delete or alter a busy line causes an error message. In order to delete or alter the line, either program execution will have to be STOPped or control must be “returned” to the calling program segment (for example, RETURN, EXIT). LINK is described in page 195 .

8

File Storage

Data and programs can be stored and retrieved for later use. This chapter discusses the following statements and functions, associated with file storage and retrieval:

MASS STORAGE IS (MSI) Specifies a default directory for successive file storage operations.

READ LABEL	Returns either the label of the directory currently in use or the labels of all volumes currently configured.
CAT	Lists the contents of a directory.
STORE	Creates a program file and records a program for later use.
LOAD	Copies a previously stored program into the computer memory.
LOAD SUB	Copies subprograms from a program file to the computer memory.
RE-STORE	Stores a program into an existing program file.
SAVE	Creates a special data file and stores a program as a series of strings.
GET	Gets data from a special data file and converts it into the computer memory as a program.
LINK	Same as GET, except all variable values are retained.
RESAVE	Saves program lines as a series of strings into an existing special data file.
MERGE	Inserts program lines from a file between lines currently in memory.
CREATE	Names and establishes a special data file or a regular data file.
ASSIGN	Opens a data file and assigns it a file number. Also used to close (de-assign) each data file.
PRINT#	Writes data into a data file.
READ#	Copies data from a data file into variables within a program.
ON END#	Branches to a recovery routine when an end-of-file (EOF) mark is detected during READ#.
OFF END#	Disables ON END#.
PURGE	Deletes a data file from the disk directory.
COPY	Duplicates the contents of one data file to another. Also for outputting the contents of a spool file to an output device.
RENAME	Assigns a new name to an existing data file.

LOCK# and UNLOCK# Control access to specific data files in multi-user applications.

TYP Determines the type of the next data item to be read (integer, string, etc.).

SIZE Returns the size of a specified file.

REC Returns the current position of the record pointer in a specified file.

WRD Returns the current position of the word pointer for a specified file; use with direct-word access.

RESET# file number Erases the contents of file to which the file number is ASSIGNED. File position is reset. Similar to purging and recreating the file. If this file is being used as a workfile, this will also reset workfile state. The file must be ASSIGNED either EXCLUSIVE (default ASSIGN) mode or must be locked in ASSIGNED in UPDATE mode.

Some functions are available for compatibility reasons, only. To get the syntax of this functions, see section , File Storage function, on page 2.

NOTE: The words "special data file" refer to a file that contains header information. The header consists of three lines of information. The first line tells the length of records currently in the file, the second tells the number of records currently in the file, and the third tells the maximum number of records defined for the file. This is different from a regular data file which contains ASCII data without any information about the file structure. Both special data files and regular data files have the extension .DATA.

If your application involves handling a large amount of data, consider using Eloquence DBMS (database management software). Database structures and operations are covered in the *Eloquence DBMS Manual*.

Syntax Terms

The following terms are used in file storage operations:

file name on HP-UX The maximum length of an Eloquence file name depends upon how the HP-UX operating system is configured; however, the maximum length bounds for configuration are from 14 to 255 characters. Of these 14 to 255 characters, 5 spaces are designated for the file name extension (for example, .DATA, .PROG, .FORM). The remaining 9 to 250 characters are user supplied. To further explain, suppose the system is configured such that file names can have a maximum length of 14 characters. Of these 14 characters, 5 character spaces are reserved for the file name extension (.DATA, .PROG, .FORM). This leaves a maximum of 9 characters spaces to be supplied by the user. The user can choose to supply a name that is from 1 to 9 characters in length. Note that the name supplied does not have to be 9 characters long. Nine is the maximum length the name can be.

The file name *cannot* contain a comma (,) or colon (:).

It is recommended that HP-UX wildcard characters *not* be used in Eloquence file names. Using them could cause problems when addressing these files using HP-UX commands. For example, if you delete the Eloquence file TEST*.DATA from the HP-UX prompt, all files beginning with the letters TEST and having the extension .DATA would be deleted. HP-UX wildcard characters are as follows:

```
$ ? * [ ] / \ ( ) @ " ` ' ^ # ;
```

file name on Windows NT The maximum length of an Eloquence file name under Windows NT can be 64 character, this includs the extension.It is not recommened to use characters, which have a special meaning on Windows NT.

file number The number assigned to a data file by an ASSIGN statement. Its range is from 1 through 10.

volume label A one- to eight-character string assigned to an HP-UX directory in either the global, group, or user configuration file. Blanks, nulls, commas, and colons are ignored.

volume spec A string within quotes containing either a unit spec (see below) or a volume label preceded by a comma.

file spec A string expression of the form "*file name*[,*volume spec*]" The *file name*, in this instance, is the user supplied name described above under “file name”. The optional *volume spec* is needed when addressing a mass storage device other than the default device (see page 204). Notice that the string expression must be within quotes.

unit spec A string expression of the form :*volume letter*[*select code*[,*device number*[,*unit code*]]] The *volume letter* can be any uppercase letter A through Z. The *select code*, *device number*, and *unit code* can be an integer from 0 through 9.

Unit spec are not supported on Windows NT and not recommended to use on HP-UX.

File Structure

An understanding of files and records is essential when discussing file storage; therefore, this section describes files and records as well as different ways they can be accessed.

Files

Files are the basic unit into which programs and data are stored; however, they must be created and named before they can be stored. Different types of files can be created:

- Program files (.PROG).
- Data files (.DATA).
- Forms files (.FORM).*
- Database (DBMS) files.*

* Refer to the corresponding programming manual for instruction on using these files.

Records

Each file contains one or more logical records. These records are established using the CREATE statement. They can have any number of bytes from 1 to 999999. A logical record is the smallest unit of storage which is directly addressable.

A disk file cannot be greater than the maximum available storage space on the disk, or 999999 records, whichever is greater.

EOFs and EORs

Files and logical records are bounded on the storage medium by marks which signify their ends. There are two types of marks—end-of-file (EOF) and end-of-record (EOR).

An EOF is placed at the end of the data in a file by specifying END in a PRINT# statement. The EOF mark takes up two bytes of storage space unless the last data item goes exactly to the end of the file.

An EOR mark can signify the end of data within a logical record. See the PRINT# statement for details.

Data Access Methods

There are three ways to store and retrieve data—serial access, direct access, and direct word access. You determine which method of data access best suits your needs. Since the decision will be based on the amount of available disk storage and the time required for your operations, an understanding of data file structure is necessary for the most efficient use of the system.

For example, suppose you are working with thousands of customer account numbers and their balances due. Your job is to output a daily list of all customers and their balances. In this situation, it is best to pack all data items (customer numbers and balances due) together tightly in a data file to save space on the disk and to save time when accessing the data. This is serial access.

To update individual customer balances, you will need another file containing customer numbers, names, addresses, items purchased, and balances due. The data in this file is arranged so that each individual item (customer name or number) can be accessed. This method of storing data usually takes more space on the disk. The advantage here is that any item can be easily updated since individual items can be accessed much faster. This is direct access.

When you wish to update many individual portions of a file as fast as possible, direct word access can be used. Using this method allows better storage efficiency than direct access.

Serial Access

Data treated as a unit of information (instead of as individual items) can be handled using serial PRINT# and serial READ# statements. When serial PRINT# statements are used to store data on the disk, data items are stored compactly without identifiable marks between items. These data items make up a file and can contain as many records as necessary. Data lists can contain both numerics and strings.

All or part of the information stored originally can be retrieved in one serial READ# statement. The list of data elements read does not have to be identical to the list originally printed in the file, but these data lists must be identical in size, type (numeric or string), and order. (The names and numeric precision you assign to these elements can still vary.) The beginning of a serial file is the only point where data access is possible.

Direct Access

When data items are to be handled individually (instead of as a unit), direct PRINT# and direct READ# operations can be used. The same PRINT# and READ# statements are used with an additional parameter to specify a record num-

ber. Each data item is stored in one (or more, if required) records so that each data item is directly accessible. Storing data directly may not utilize storage space effectively, since only a part of a record (or records) required for storage may be used.

Each of the data items stored originally can be retrieved by using a direct READ#. The READ# begins at the start of a specified record. The list of data items does not have to be identical to the list originally printed in the record, but the data items must be identical in size, type (numeric or string), and order. Notice that since the numeric precision need not be the same from PRINT# to READ#, numeric conversion is easily performed.

Direct Word Access

When you wish to handle individual data items and also wish to specify the exact point within a record where the data is to be printed or read, use direct word access. This access method is specified by adding another parameter, called a word pointer, to the READ# and PRINT# statements.

Direct word access offers the best accessibility to data, since you specify the exact word at which the read or print begins. Use of disk storage space is good, too, since end-of-record (EOR) marks are not added after the data; therefore, remaining space in the record can be used for more data storage.

Comparing Data Access Methods

As mentioned before, you decide on which method of data accessing is best for your particular needs. This decision is usually not made easily, because of the advantages and disadvantages of each method. For example, more efficient storage space utilization must be sacrificed for a shorter access time and vice versa. Once your decision has been made, it is difficult to change later, so make your decision carefully.

The advantages and disadvantages of accessing data with each method are summarized below:

Table 10 **Comparison of Data Access Methods**

	Access Time	Storage Efficiency
Serial	Varies - longer for higher-numbered records	Best - data is packed solidly
Direct	Good - direct access to any record	Varies - only part of a record may be used
Direct Word	Best - only part of a record need be accessed	Good

The Default Mass Storage Device

At startup of Eloquence or when SCRATCH A is executed, a default mass storage device is automatically specified. This is the device to which all file storage operations are directed if no other device is specified. To establish the default mass storage device, Eloquence looks consecutively in the user and group configuration files for an MSI statement. If an MSI statement is not found, the first VOLUME spec in the global configuration file is used as the default mass storage device.

The default device is changed by executing the MASS STORAGE IS statement:

MASS STORAGE IS volume spec

or

MSI volume spec

The *volume spec* is a string expression containing either a *unit spec* or a *volume label*. Note that a comma before the *volume label* is optional.

The following example shows how to change the default device by specifying the MSI command and a *unit spec*:

```
MASS STORAGE IS ":C2,7"
```

NOTE: The "unit spec" is not supported on Windows NT and not recommended on HP-UX.

As another example, if the label TEST is assigned to the directory /usr/test, any of the following statements can be used to set this directory as the default mass storage device:

```
10   MASS STORAGE IS "TEST"
 .
200  MSI "TEST"
 .
 .
500  Label$="TEST"
510  MSI Label$
```

To reset the mass storage device to its value at startup of Eloquence, enter the following command:

```
MSI ""
```

You can omit the *select code*, *controller address*, and *unit code* parameters from successive file specs by stating them in a MASS STORAGE IS at the beginning of a program.

Cataloging Files (CAT)

The CAT (catalog) statement outputs a listing of directory information for a storage medium, including read/write authority, the file owner, the group, and physical specifications.

CAT[ALOG] [catalog spec] [,volume spec] [,file type]

NOTE: The CAT statement is not supported on Windows NT platform

The *catalog spec* is an optional string expression consisting of 0 to 6 characters, followed by an optional *volume specifier*. When the *catalog spec* is specified, only those files whose names begin with that combination of characters are listed.

The optional *file type* is a four-character string which specifies you want to list only that type of file (for example, PROG, DATA, or FORM).

The catalog listing is sent to the standard output device.

For example, the following CAT statement lists, to the standard output device, all program files on the volume labeled Prog1 whose names begin with Ab:

```
CAT "Ab,Prog1",PROG
```

The next example shows a CAT statement and the listing it produces:

```
CAT",MASTER"
total 3
-rw-rw-rw-   1 john   tstctr       32 Nov 15 17:06 DATA.DATA
-rw-rw-rw-   1 john   tstctr     2816 Nov 15 17:32 FRM.FORM
-rw-rw-rw-   1 john   tstctr     1070 Oct 12 10:59 GAME.PROG
```

Listed below is an explanation of the columns of a catalog listing:

Access Authorization Indicates the type of access that is set for the file owner, the group associated with the file, and all others. This could be read (r), write (w), execute (x), or any combination of the three. Taking the above example (**-rw-rw-rw-**), the first **rw-** string specifies the access authorized for the file owner, the second (**rw-**) authorization for the group, and the third (**rw-**) authorization for all others.

Link Counter Defines the number of links to a file.

User Id A one-to-eight character string, defining the file owner.

Group Id A one-to-eight character string, defining the group that owns the file.

File Size The size of the file expressed in bytes.

Date The date the file was last saved.

Time The time the file was last saved.

Name The name given to the file when the information is stored on the medium. The file name includes a five-character extension defining what type of file it is (.DATA, .PROG, .ROOT, or .FORM).

NOTE: The CAT statement is tied to the HP-UX command ll (long list). Therefore, messages that occur upon executing a CAT statement are HP-UX messages. For example **`ERROR 56`** indicates that a file was not found, **`ERROR 170`** indicates that the HP-UX ll command failed. If a CAT statement is issued without any parameters and the directory is empty, no files are listed and no message is returned. For more information refer to the HP-UX documentation.

Using message Catalogs

The functions CATOPEN, CATCLOSE and CATGETMSG give application message catalog support. This allows eloquence applications to use HP-UX message catalogs (built using gencat).

Syntax:

```
CATOPEN "filename"
CATCLOSE
CATGETMSG ["Msg_set",] Msg_num;String_var$
```

Sample application code:

```
   DIM Msg$[80]
   CATOPEN “APP.CAT,TEST”
   Msg_set = 1
   Msg_num = 10
   CATGETMSG Msg_set,Msg_num;Msg$
   LDISP “Message = “;Msg$
   CATCLOSE
```

Msg_set may be omitted in CATGETMSG and will default to 1. If a CATGETMSG is executed with no previous CATOPEN, ERROR 51 (file not open) will be returned.

If a message could not be located, the result variable will contain an empty string.

A message catalog remains open across programs (like COM) and will be reset if program execution stops (e.g. END).

Identifying Volume Labels

To find the volume label(s) of the current directories in use, use the READ LABEL statement.

READ LABEL {*string variable* [ON *volume spec*] *string array name*}

If a *string variable* is specified, the label on the volume will be returned in that variable. If the *volume spec* is *not* specified, READ LABEL returns the volume label of the current directory. If the *string array name* is given, the volume labels found will be returned in that array in the following form:

volume label :*unit spec* [*]

An * indicates the current volume.

The following is an example of the READ LABEL statement:

```
10 DIM A$(1:100)
20 READ LABEL A$(*)               !Reads up to 100 volume labels.
30 FOR I=1 TO 100
40    IF LEN(A$(I))=0 THEN Done!Checks the length of the Ith
element of A$.
50    P=POS(A$(I),":")            !Looks for colon.
60    DISP A$(I)[1,P1];TAB(10);A$(I)[P;7];TAB(20);MAPVOL$(A$(I)[1,
P-1])
70    NEXT I
80 Done: END
```

MAPVOL$ in line 60 is used to show which volume label and unit spec are matched to which directory.

Here are more examples:

```
READ LABEL A$ ON ""
```

Returns the volume label of the current mass storage device.

```
READ LABEL A$ ON ":C2,7,2"
```

Returns the volume label associated with the unit spec :C2,7,2.

```
READ LABEL A$ ON ",TEST"
```

Simply returns the volume label TEST.

NOTE: "unit spec" are not supported on Windows NT and is not recommended on HP-UX.

Storing and Retrieving Programs

Programs are stored into program and data files using STORE and SAVE. Programs are retrieved using LOAD, GET, LINK, or MERGE. The RUN command can also be used to bring previously STOREd and SAVEd programs into memory and begin execution. (see section , Storing a Program, on page 54)

NOTE: The statements LOAD, GET, LINK, MERGE, {RE-}STORE and [RE-}SAVE can not be executed on the commandline of the IDE. This functions are implemeted in a different way. (see section , The Integrated Development Environment (IDE), on page 68)

The STORE Statement

The STORE statement creates a program file and stores the program currently in memory into the program file. A program file has the extension .PROG. The program is stored in an internally coded format, allowing rapid access with LOAD or LOAD SUB. Syntax for the STORE statement is as follows:

STORE *file spec* [,*ProtectCode*] [;*Option*]

As an example, the following program line stores the program currently in memory onto the disk labeled PROGS under the name Life2:

```
210  STORE "Life2,PROGS"
```

The *volume label* need not be included when addressing the default storage device.

ProtectCode The optional protect code must be specified in order to edit a protected program file or to save a program with password protection. When you load a protected program without the correct password, you will neither be able to edit the program, nor will you be able to store the program. If you give the wrong password, an error message is returned by the LOAD statement. When you specify a ProtectCode for the STORE and RE-STORE statements, the program will be stored protected. Since program protection is not possible with program formats before Eloquence A.06.00, this will use the new A.06.00 program format.

Option The Option argument contains either the desired program format or a comma separated list of program file options.

Program compatibility option This option simply specifies the Eloquence revision, the program format should be compatible to.
Example:
STORE "PrgName";"A.05.01"

Program options Alternatively, you can specify a comma separated list of options and settings which should be used to store the program file. The following options are currently supported:

Table 11

option	possible values	Comment
rev	A.xx.xx	Eloquence revision
fmt	HP260 or "0"	Compatible format
	EXT or "1"	A.05.01 program format
	TAG or "2"	A.06.00 program format

Example:

```
STORE "PrgName";"rev=A.05.01"
STORE "PrgName";"fmt=TAG"
STORE "PrgName";"fmt=2"
```

Error Messages

62 - File is protected or wrong protect code specified

66 - You are not authorized to store this program

The RE-STORE Statement

A program file can be loaded into memory and edited, then re-stored into the same file using the RE-STORE statement. Syntax for this statement is as follows:

RE-STORE *file spec* [,*ProtectCode*] [;*Option*]

The LOAD Statement

Programs recorded with STORE are retrieved with the LOAD statement. Syntax for this statement is as follows:

LOAD *file spec* [,*line id*] [,*ProtectCode*]

The LOAD statement erases any program and data in memory and loads the program. Any information stored in common, however, is preserved if the loaded program has a COM statement. If the LOAD statement comes from the keyboard and no *line id* is specified, control returns to the keyboard after loading. If it comes from execution of a program line in memory, execution begins at the first line of the loaded program. When the *line id* is specified, however, execution of the loaded program begins at that line.

The program can only be loaded when the right protectcode is provided, otherwise an error 66 occures.

For example, the following program line loads the program in the previous example back into memory and execution then begins with line 20:

```
220  LOAD "Life2",20
```

The LOAD SUB Statement

The LOAD SUB statement loads subprograms from a program file to the end of the program currently in memory.

LOAD SUB *file spec* [,*starting line number*[,*increment*]]

[;*starting segment* [,*last segment*]]

With no optional parameters, LOAD SUB loads all subprograms (both FN and SUB) from the specified program file. The *starting segment* and *last segment* parameters can be used to specify certain subprograms in the file. For example, if you wish to load only the third subprogram, specify LOAD SUB;3,3. An error occurs if either value is greater than the number of subprograms in the specified program file.

If the first line number in the subprogram is not greater than the last line of the current program in memory, the subprogram is automatically renumbered. The renumbering starts after the last line of the current program, in increments of ten. Another *starting line number* can be specified, if desired, but it must be greater than the last number of the current program. Another line-numbering increment can also be specified, if desired.

The SAVE Statement

The SAVE statement creates a special data file and stores the program and any subprograms currently in computer memory into the file. This special data file has the extension .DATA. Syntax for the SAVE statement is as follows:

SAVE *file spec* [,*beginning line id*[,*ending line id*]]

The program is saved as string data, one program line per string. In this way, the file can be read, modified, or rewritten as string data by other programs.

When only the *file spec* is given, the entire program is saved. If the *beginning line id* is specified, the program is saved from that number to the end. If both line ids are specified, the program section from the *beginning line id* through the *ending line id* is saved.

The following example saves the program currently in memory, beginning with line 30, into a special data file named MONEY on the default storage device:

```
230   SAVE "MONEY",30
```

As another example, this program line saves lines 50 through 250 into the special data file Prog1 on a disk named Software:

```
240   SAVE "Prog1,Software",50,250
```

NOTE: The words “special data file” refer to a file that contains information on the maximum number of records, the length of records, and the number of records currently in the file.

NOTE: The STORE and LOAD operations are faster than SAVE and GET. STOREd programs are already coded in an internal format, while SAVEd programs must be coded before they can be executed.

The GET Statement

The counterpart of the SAVE statement, the GET statement, returns into computer memory a program saved previously with the SAVE statement. GET will also read into memory any string data file consisting of valid Eloquence statements. Syntax for the GET statement is as follows:

GET *file spec*[,*line id*[,*line id*]]

If no line ids are specified, the GET statement erases any program and data in memory (except that associated with a COM statement), as it loads the specified file. Once the file is in memory, execution begins at the first line of the loaded program.

When one *line id* is specified, the loaded program is renumbered and executed starting with the specified *line id*. If a second *line id* is specified, the loaded program is renumbered starting with the first *line id* and is executed starting with the second *line id*. Any lower-numbered lines from a previous program are retained. The numbering remains the same on the storage medium.

Here is an example addressed to the default storage device:

```
250  GET "MONEY",50
```

The file MONEY is retrieved and renumbered; any lower-numbered lines already in memory are retained. Execution of the program starts at line 50.

Listed below is another example:

```
260   GET "Prog1,Software",50,10
```

This program is also renumbered with any lower-numbered lines being retained. Execution begins with line 10 already in memory.

NOTE: **`ERROR 18`** occurring during a GET, LINK or MERGE operation indicates that a line being entered into memory is too long to be accepted (512 characters maximum).

NOTE: The STORE and LOAD operations are faster than SAVE and GET. STOREd programs are already coded in an internal format, while SAVEd programs must be coded before they can be executed.

The LINK Statement

The LINK statement is identical to the GET statement discussed previously, except that the current values of all variables are retained. Syntax for this statement is as follows:

LINK *file spec* [,*line id* [,*line id*]]

If no line ids are specified, the program is loaded, erasing the current program in memory.

The first *line id* specifies that the loaded program is to be renumbered and starts with the line number of the specified line. If two line ids are specified, execution begins with the second line specified.

In effect, GET performs a SCRATCH V operation on the new program, whereas LINK performs a CONT (continue) operation, involving no initialization of variables.

NOTE: `ERROR 18` occurring during a GET, LINK or MERGE operation indicates that a line being entered into memory is too long to be accepted (512 characters maximum).

The RESAVE Statement

A program stored in a special data file can be loaded into memory and edited. It can then be resaved into the same file using the RESAVE statement. Any data RESAVE will purge and overwrite data already in the specified file.

```
{ RESAVE  }
{ RE-SAVE }   file spec   [,line id [,line id] ]
```

When no line ids are specified, the entire program is saved. When one *line id* is specified, the program is saved from that line to the end. When two line ids are specified, that block of lines is saved.

NOTE: The words "special data file" refer to a file that contains information on the length of records, the number of records currently in the file, and the maximum number of records defined for the file.

NOTE: The STORE and LOAD operations are faster than SAVE and GET. STOREd programs are already coded in an internal format, while SAVEd programs must be coded before they can be executed.

The MERGE Statement

The MERGE statement takes program lines from a data (.DATA) file and positions them in memory, either in front of the program currently there, between consecutive lines in the program currently there, or behind the program currently there.

MERGE *file spec* [,*line id* [,*line id*]]

If the first *line id* is specified, the program lines in the specified file are renumbered beginning with that line. The spacing between renumbered lines remains the same. The second *line id* specifies where program execution should begin.

To further explain the MERGE statement, suppose you have two files (OLD.DATA and NEW.DATA) and the line numbers in these files are as follows:

```
OLD.DATA      NEW.DATA
   1             10
   2             20
   3             30
```

Listed below are three examples using the sample files. Each example shows a different MERGE statement and its result.

```
GET "OLD"                GET "OLD"                GET "NEW"
MERGE "NEW"              MERGE "NEW",100          MERGE "OLD",21

Result:                  Result:                  Result:
1                        1                        10
2                        2                        20
3                        3                        21
10                       100                      22
20                       110                      23
30                       120                      30
```

NOTE: **`ERROR 18`** occurring during a GET, LINK or MERGE operation indicates that a line being entered into memory is too long to be accepted (512 characters maximum).

Storaging and Retrieving Data

In addition to storing program lines, files are used to store collections of data that are too large to be contained conveniently in DATA statements in your program. These files have the extension .DATA and throughout this manual are referred to as data files. Data files are often accessed by more than one program.

The use of data files involves these six operations which are explained in detail later in this section:

1. A data file must be CREATEd by specifying the file name and size. You may also specify on what volume the data file should be created.
2. A data file must be opened using the ASSIGN statement. This gives your program access to the data file and assigns a number to that data file. The file number is then used throughout a program to refer to that particular data file.
3. Data is put into the file via the PRINT statement.
4. Data is copied from the file into assigned storage areas in a program (variables or arrays) via the READ statement.
5. Closing a data file indicates that your program has finished using that file. The data remains in the file for future use.
6. You can PURGE a data file when the data in that file is no longer required.

Creating a Data File

The CREATE statement is used to create a data file. Syntax for this statement is as follows:

[F]CREATE *file spec* ,*number of defined records* [,*record lenght*]

The *record length* is a numeric expression specifying the length of logical records in bytes and is rounded up to an even integer. If it is not specified, a record length of 256-bytes is assumed.

The *number of defined records* and the *record length* can be numeric expressions in the range 0 to 999999.

On the HP 9000 Series 800, a value greater than zero for *number of defined records* creates a special data file (one containing header information). A value of zero for *number of defined records* creates a regular data file, in other words an ASCII text file with no header information. Note, however, that in this case you cannot specify a record length.

On the HP 260, if the *number of defined records* is zero, no data file is generated and **`ERROR 67`** is returned.

In contrast to HP BASIC/260, the Eloquence statements FCREATE and CREATE are identical. Neither of them generates an end-of-file (EOF) marker.

Two examples of the CREATE statement are listed here:

```
CREATE "DATA",0
```

Creates a regular data file

```
CREATE "DATA",10
```

Creates a special data file

Opening a Data File

Data files must be opened before they can be accessed. Each file is opened and assigned a file number with the ASSIGN statement. Syntax for this statement is as follows:

$$\text{ASSIGN}\begin{Bmatrix} \textit{file spec } \text{TO\# } \textit{file number} \\ \#\textit{file number } \text{TO } \textit{file spec} \end{Bmatrix} [,\textit{return variable}]\ [;\textit{class list}]$$

There is also a statement XASSIGN, which can ASSIGN a file with no DATA extension. See below.

The ASSIGN statement sets up an internal table of file numbers to be used with the PRINT# and READ# statements. The file table has room for 20 entries, allowing up to 20 files to be open for each user, but maximal 10 files in a segment. One data file can be assigned up to seven file numbers. The assignment table remains in effect until a new program is loaded or run, or until a SCRATCH, SCRATCH P, STOP, or END is executed. See page 237 , later in this chapter, for more details. The *file number* is a numeric expression; its range is 1 through 10.

The optional *return variable* can be a simple numeric variable or array element and is set after execution to indicate various results. Its value is used to check for errors. If no *return variable* is specified, an error occurs if the file is not found or is the wrong type. Possible values returned by the *return variable* are listed and explained in the following table:

Table 12 **Comparison of Data Access Methods**

Return Variable	Meaning
0	File available, assignment complete.
1	File not found (same as error 56).
2	File is protected.
4	Access error (errors 91 through 93).
5	Other error.

Here are some examples of how to use the ASSIGN statement:

```
100  ASSIGN #1 TO "DATA"
110  ASSIGN "Scores" TO #4,Return
120  ASSIGN "Scores" TO #5
130  ASSIGN #2 TO "Poker,Games"
140  ASSIGN #3 TO "Totals:F2,6,1"
```

Line 110 illustrates a *return variable*. Lines 110 and 120 show that a file can be assigned more than once. Lines 130 and 140 show that the *file spec* can include either a *volume label* (**Games**) or a *unit spec* (**F2,6,1**).

The optional *class list* parameter provides flexibility in determining the type of file access. The following class words are available—EXCLUSIVE, UPDATE, READONLY, or APPEND.

Only one of the three words can be used at a time. Thus, if EXCLUSIVE is specified, UPDATE or READONLY cannot be used. To explain further, the following statement assigns the file named PAYROL to file number 1 and specifies read-only access:

```
100  ASSIGN #1 TO "PAYROL";READONLY
```

EXCLUSIVE means that only one access may exist to the file anywhere in the system, but the same process can assign the file more than once. The file may not be assigned elsewhere; if anyone attempts this, an error occurs. Once a file has been assigned in exclusive mode, any attempt by the current program, or any other user's program, to assign the file results in an error. If no *class list* is specified, EXCLUSIVE is assumed. READ# and PRINT# statements may be done on the file without regard to locking. LOCK# and UNLOCK# are ignored if executed on a file assigned to exclusive mode. SORT workfiles must be assigned in exclusive mode.

If UPDATE is specified, shared access is granted to the data file. The ASSIGN fails, however, if there are any other references to the data file in either read-only or exclusive mode. Once a file has been assigned in update mode, READ# operations may be done. In order to PRINT to the file, however, the LOCK# statement must be used to gain write access. The LOCK# statement causes an error if the file is already locked by the current program through another reference.

If READONLY is specified, shared read access is granted to the file. The ASSIGN fails if there are any other references to the file in either update or exclusive mode. Once a file has been assigned in read-only mode, READ# operations may be done. There is no way to write to a data file assigned to read-only mode. The LOCK# and UNLOCK# statements are ignored if executed on a file assigned to read-only mode. The GET, LINK, and MERGE statements require read-only access to the file being accessed.

The **APPEND** file mode causes all output to a HP-UX sequential file automatically appended to the end of the file.

If a file is opened in **APPEND** mode, it will behave different:

- If file does not exist, it will be created upon ASSIGN with a zero size.
- File is *not* locked exclusive. Because all output will automatically appended to the end of the file, it's not necessary to protect file contents by locking it. If you need exclusive access, you may still use the **LOCK #** and **UNLOCK #** statements.

For example:

```
ASSIGN #1 TO "Logfile";APPEND
PRINT #1;DATE$;TIME$;"Message"
...
```

The example code above will cause output appended to Logfile. If Logfile does not exist, it will be created.

Note that in order to guarantee that the data being read from a data file is correct, the file must be LOCKed. If not, the data may be old (since data files do not use the sophisticated buffering scheme employed by Eloquence DBMS). It is also possible that the data on the disk has been only partially updated by the user who has the data file locked. In this case, an actual error may occur during a read. So, unless some special agreement exists between programs altering a file in update mode, it is best to also lock the file before reading.

The XASSIGN statement assigns a file of the type specified. This could be any hp260 file type (DATA, PROG or FORM). If the type is OTHR, *no* extension will be added to the filename.

XASSIGN # *file name* TO *file name*; *file type* [,*assign_options*]

Example:

```
XASSIGN #1 TO "/etc/passwd";OTHR;READ ONLY
```

will assign #1 to

```
/etc/passwd
```

Data Pointers

A record pointer is automatically maintained for each opened data file. This pointer is used to specify at which record data storage or retrieval begins in the data file. A word pointer is also maintained for each current record. It points to the first word of the next data item to be accessed in the record.

After executing an ASSIGN statement, the record pointer is positioned at the beginning of the first logical record in a data file. The word pointer is then incremented through the record as data items are stored (PRINT#) or retrieved (READ#). A new ASSIGN statement obsoletes the previous one and resets all pointers for the specified file.

The current position of each pointer can be found by using the REC (record) and WRD (word) functions, as described later in this chapter.

Converting a Text File

The ‘convrt’ program converts text files from HP-260 format into HP-UX ASCII format. The resulting file is assigned the same name as the HP-260 file plus the extension ’.txt’ in the current directory. If the file already has an extension, it will be overwritten to ’.txt’.

The syntax is as follows:

convrt [options] *file* [*file*]

Options

-v	Detailed listing of procedures
-s	Write to stdout
file	Show file or files to be converted.

Example

To convert text files dbmap1.DATA und dbmap2.DATA into HP-UX format:

convrt -v dbmap*.DATA

NOTE: The ‘convrt’ program converts text files, or parts of a file containing text until non-text data is detected.

Serial Access

Serial access is used to store or retrieve data items one after the other, without regard to logical record bounds. For each data file opened, the data pointers keep track of the data item currently being accessed. As you store or retrieve data, the pointers move serially forward through the data file.

The Serial PRINT# Statement

The serial PRINT# statement records values onto the specified file from the specified variables or strings. Syntax for this statement is as follows:

$$\text{PRINT\# file number;} \begin{Bmatrix} \text{data list [,END]} \\ \text{END} \end{Bmatrix}$$

The *data list* is a collection of items separated by commas. The items can be variables, array identifiers, and numeric or string expressions. Including the optional END causes an EOF mark to be printed at the end of the data; otherwise, an EOR mark is placed after the data list is printed.

Printing begins at the position of the data pointers (which is after the data item most recently stored or retrieved) or at the beginning of the file if nothing has been stored or retrieved. The record pointer can also be repositioned to the beginning of the file (see page 231).

Here is a simple program which creates a file named CLASS and prints the names and grades of five students:

```
10   CREATE "CLASS",1
20   ASSIGN #1 TO "CLASS"
30   FOR I=1 TO 5
40     INPUT "STUDENT'S NAME?";N$,"TEST SCORE?";S
50     PRINT #1;N$,S
60   NEXT I
70   PRINT #1;END
80   END
```

Line 50 prints students' names and grades, alternately, in the file. Line 70 places an EOF mark after the five sets of data are printed. The EOF prevents reading data beyond its position.

When printing a long string, it might possibly be too long to be contained in one logical record. In this case, the string is automatically broken up and stored into a series of logical records. This requires an additional two words each time the string crosses over into another logical record. The parts of the string are identified at the first record, intermediate records, and the last record.

Data can be stored using the PRINT# statement in a file created with the SAVE statement. SAVE, in effect, performs a serial print into a file.

Here are two examples:

```
100   PRINT #3;Apples,Bananas,Carrots
110   PRINT #3;Donuts,Eggs(*)
```

These two statements store values for all five variables into file `#3`. The EOR which was placed after the data when line 100 was executed is overwritten when line 110 is executed. Another EOR is printed after the data in line 110. Remember, an EOR signifies that there is no more data between the data pointers and the end of the record.

The serial PRINT# statement can also be used to generate program lines into a file. Such a file can be retrieved with GET.

Here are two examples:

```
50   P$="COUNTR"
60   CREATE P$,3,50
70   ASSIGN #1 TO P$
80   PRINT #1;"10 FOR I=1 TO 10","20 PRINT I","30 NEXT I","40 END"
90   GET P$,10,10

RUN
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
```

Executing LIST produces:

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

Below you find two examples concerning printing of User Defined Type variables.

In the example below, the Comment$ member variable from the Phone1 variable is PRINTed.

```
Phone1.Comment$="*Fancy Comment*"
```

```
PRINT #3;Phone1.Comment$
```

In addition to accessing single variables, you can specify the whole variable at once.

The example below prints all member variables of Phone1:

```
PRINT #3;STRUCT Phone1
```

The Serial READ# Statement

The serial READ# statement retrieves values for variables and strings of characters from the specified file. Syntax for this statement is as follows:

READ# *file number*; *variable list*

Before you can use data which has been stored in a data file with a PRINT# statement, you must read the data back into the computer memory. The data is not erased from the file, it is merely copied into the variables specified in the same order in which it was stored with the PRINT# statement. The User Defined Types can be used in the same way as with PRINT#. Variables do not have to have the same names specified in the PRINT# statement. Reading begins after the last item printed or read on the specified file. To begin reading from the beginning of the file, you must reposition the record pointer (see page 231) or do another ASSIGN.

As an example, the data printed in the previous example in a file named CLASS can be read by using this program:

```
10   ASSIGN #1 TO "CLASS"
20   PRINT "    NAME           GRADE"
30   FOR I=1 TO 5
40     READ #1;Name$,Score
50     PRINT Name$,Score
60   NEXT I
70   END
```

Notice that the serial READ# statement must specify the types of data (data elements or string variable) in the order in which they were originally stored in the file. Line 40 reads a string variable and then a numeric variable. This program can run only when the order of the data on file is known. Here is the printout:

```
    NAME           GRADE

Charlie Brown       79
Casey Jones         99
```

```
Sean Jackson        91
Jack Allison        83
Sam Amigo           95
```

The variables into which you read data items need not have the same names used when the items were printed on the file. Although the variable name changes (from N$ and S when stored, to Name$ and Score when retrieved), the order in which the two data types are accessed is the same.

When a serial READ# statement encounters the EOF mark previously placed by the last PRINT# statement, the program ends and an error indicates the end of the file. The program can be written to run without displaying an error by using the ON END# statement, described later in this chapter.

Positioning the Record Pointer

It is often necessary to position the record pointer to the beginning of a specific record in a file before executing a serial READ# statement. This is done by using only *file number* and *record number* parameters in a direct READ# statement:

READ# *file number* ,*record number*

A serial PRINT# or READ# statement can then be executed to access the beginning of the specified record, rather than the beginning of only the first record in the file.

To see how this works, first use the next program to store consecutive values beginning from the 8th record of a file named NUMBERS:

```
10   CREATE "NUMBERS",15
20   ASSIGN #1 TO "NUMBERS"
30   READ #1,8
40   FOR Value=1 TO 300
50     PRINT #1;Value
60   NEXT Value
70   END
```

The ASSIGN statement sets the record pointer to the beginning of the first record in the file. The pointer is then repositioned to the beginning of the eighth record by the READ# statement. The FORNEXT and PRINT statements fill the file with the numbers 1 through 300, starting at the eighth record.

Now use the following program to read and display the data, beginning at record 14:

```
10   DIM A(7)
20   ASSIGN #1 TO "NUMBERS"
30   READ #1,14
40   FOR I=1 TO 12
50     READ #1;A(*)
60     DISP A(*)
```

```
70   NEXT I
80   END

RUN
  193    194    195    196    197    198    199    200      record 14
  201    202    203    204    205    206    207    208
  209    210    211    212    213    214    215    216
  217    218    219    220    221    222    223    224

  225    226    227    228    229    230    231    232      record 15
  233    234    235    236    237    238    239    240
  241    242    243    244    245    246    247    248
  249    250    251    252    253    254    255    256

  257    258    259    260    261    262    263    264      record 16
  265    266    267    268    269    270    271    272
  273    274    275    276    277    278    279    280
  281    282    283    284    285    286    287    288
```

The ASSIGN statement automatically sets the record pointer to the beginning of the first record. The pointer is then repositioned to the beginning of record 14 by line 30. The serial READ# statement begins reading data from that point on.

NOTE: Reading record 17 causes an error if more than 12 values are read. Record 17 contains only 12 values (289–300).

Since each real-precision value uses 8 bytes of memory, 32 values can be printed into a 256-byte record. On the file NUMBERS, for example, the following values are stored on these corresponding records:

Table 13 **Comparison of Data Access Methods**

Record No.	Full-precision Values
1 through 7	(none)
8	1 through 32
9	33 through 64
10	65 through 96
11	97 through 128
12	129 through 160
13	161 through 192
14	193 through 224
15	225 through 256
16	257 through 288

Table 13 Comparison of Data Access Methods

Record No.	Full-precision Values
17	289 through 300

Data read must correspond to the type (numeric or string) that was printed. However, a numeric data item need not be one of the same precision. Precision is automatically converted. You can also print an array and read back simple variables or other arrays and vice versa.

Direct Access

Direct file access is used to store or retrieve data items from one logical record at a time.

The Direct PRINT# Statement

The direct PRINT# statement is nearly identical to the serial PRINT# statement except that it prints data onto the file starting at the beginning of a specified record. Syntax for this statement is as follows:

$$\text{PRINT\# file number, record number} \left[; \begin{Bmatrix} \text{data list [,END]} \\ \text{END} \end{Bmatrix} \right]$$

The *data list* is identical to that used in the serial PRINT# statement. The direct PRINT# statement prints data into the specified record of the file. Printing starts at the beginning of the specified record. Any previous data in the record is overwritten. Specifying END causes an EOF mark to be printed after the data (first syntax) or at the beginning of the record (second syntax). When END is not used, an EOR (end-of-record) mark is placed after the last item printed. The ON END# statement and the TYP function can be used to detect EOFs, as shown later in this chapter.

The program below prints consecutive numbers into each odd-numbered record of a 10-record file named TEN.

```
10   Data=1
20   ASSIGN #1 TO "TEN"
30   FOR Record=1 TO 10 STEP 2
40     PRINT #1,Record;Data
50     Data=Data+1
60   NEXT Record
70   END
```

By printing in specific records of the file TEN, previous data in those records is erased and replaced by the new data. File TEN now contains the following:

Table 14 **Comparison of Data Access Methods**

Record No.	Data
1	1 (EOR)
2	(Null)
3	2 (EOR)
4	(Null)
5	3 (EOR)
6	(Null)
7	4 (EOR)
8	(Null)
9	5(EOR)
10	(Nothing)

An EOR is automatically added at the end of each odd-numbered record.

When neither the data list nor END are used in a direct PRINT# statement, it erases the contents of the specified record and fills it with EORs. For example, the following program erases every third record of file TEN, which was opened and accessed in the previous program:

```
100   ASSIGN #1 TO "TEN"
110   FOR Erase=1 TO 10 STEP 3
120     PRINT #1,Erase
130   NEXT Erase
140   END
```

The information now left in the file is as follows:

Table 15 **Comparison of Data Access Methods**

Record No.	Data
1	(EORs)
2	(Null)
3	2 (EOR)
4	(Null)
5	3 (EOR)
6	(Null)
7	(EORs)
8	(Null)
9	5(EOR)
10	(Nothing)

When an EOR is detected by a serial READ# statement, it skips over the entire record and attempts to access data in the next record. You can use a direct PRINT# statement to write over the EOR marks.

When the data list is omitted from a PRINT# statement, as shown in the following statement, an EOF is placed at the beginning of the specified record:

PRINT# *file number* ,*record* ;END

If a serial or direct READ# attempts to read from that record, reading the EOF terminates the operation.

The Direct READ# Statement

The direct READ# statement is like the serial READ# statement except that reading of data into variables begins at the beginning of the specified record and will not go past an EOR mark. Like the serial READ# statement, the direct READ# statement will not read past an EOF mark. Syntax for the direct READ# statement is as follows:

READ# *file number* ,*record number* [;*variable list*]

As with serial READ# statements, the variables into which you read data items do not have to be the same variables used to print the data items on the record, but they must be the same type (numeric or string) and in the same order.

If the number of items making up the data list is greater than the data in the defined record, however, an EOR error occurs.

The following program reads the data printed in the 5th and 9th records of the previously-used file named TEN:

```
200  ASSIGN #1 TO "TEN"
210  READ #1,5;R5
220  READ #1,9;R9
230  PRINT "Data in record 5 =";R5
240  PRINT "Data in record 9 =";R9
250  END

Data in record 5 = 3
Data in record 9 = 5
```

The program reads the data from records 5 and 9 and outputs the data on the standard printer.

Repositioning the Record Pointer

If the *variable list* is omitted from a direct READ#, the pointer is repositioned to the beginning of the specified record. To reposition the pointer to the beginning of a file (for use with serial data access) execute the following:

READ# *file number* ,1

Direct Word Access

Direct word access allows you to begin printing or reading data at any given word within a specified record of a data file. This enables you to define subrecords within each logical record.

The Direct-Word PRINT# Statement

PRINT# *file number* ,*record number* ,*word pointer* [;*data list*[,END]]

The direct-word PRINT# statement stores data items in specific records of a file. The data is written to the specific *file number* and *record number*, starting at the word addressed by *word pointer*. The *word pointer* can be an integer expression in the range 1 through (bytes-per-record/2)+1. If the *word pointer* is exactly one greater than the highest word in the record, that word pointer will address the first word of the next record.

Example: If there are 256 bytes per record, and the word pointer is 129, the following would be equivalent:

```
PRINT #1,1,129
PRINT #1,2,1
```

The optional END parameter places an EOF after the last data item printed. When END is not used, the remainder of the record is left unchanged. Remember that ON END# and TYP can be used to detect EOFs, as explained later in this chapter.

Listed below is an example program which opens a 1,000-record file named STOCK. Each record can contain 256 bytes of data about each part to be stocked. For now the program enters only four items—part number, description, unit cost, and current quantity on hand.

```
10   !
20   !       OPEN NEW STOCK FILE
30   !
40   CREATE ”STOCK”,1000
50   DIM Part$[20],Desc$[20]
60   ASSIGN #1 TO ”STOCK”
70   INPUT ”TODAY’S DATE:”;Date$
80   PRINT LIN(5),SPA(30),”PARTS SET UP FOR”,Date$
90   PRINT ”PART NO.     DESCRIPTION”,TAB(40),”UNIT COST QTY. ON H
AND”,LIN(2)
100  FOR Record=1 TO 1000
110    INPUT ”PART NUMBER?”;Part$[1,20]
120    IF UPC$(Part$[1,4])=”DONE” THEN 220
130    INPUT ”DESCRIPTION?”;Desc$[1,20]
140    INPUT ”UNIT COST?”;Cost
150    INPUT ”INITIAL QUANTITY?”;Qty
160    PRINT #1,Record,29;Qty
```

```
170    PRINT #1,Record,13;Desc$
180    PRINT #1,Record,25;Cost
190    PRINT #1,Record,1;Part$
200    PRINT Part$,Desc$;TAB(35);Cost,Qty
210  NEXT Record
220  PRINT "DONE",LIN(5)
230  END
```

Lines 80 and 90 print headings for a table of input data. The FOR-NEXT loop inputs four items to be printed in each record, prints each item into subrecord within the currently-specified record, and outputs the data on the standard printer. Line 120 exits the loop when the operator enters **DONE** (upper or lower case) for a part number.

Notice that the word pointer parameter within each PRINT# specifies the first word for each subrecord.

These four items use only 64 bytes of each defined record; there are still 192 bytes available in each record for extra data.

It is important to know the exact length of each string variable printed using direct-word PRINT#. For example, lines 110 and 130 in the last program generate 20-character strings, regardless of the number of characters input. This, in turn, ensures that 20-character (20 byte) subrecords will be printed in lines 170 and 190. If the subscripts were not used in the INPUT statements, the first two sub-records printed in each logical record would vary in size, depending on the current string length.

The Direct-Word READ# Statement

READ# *file number* ,*record number* ,*word pointer* [;*data list*]

The direct-word READ# statement reads numbers and strings into variables from a specified record, starting from a specified word.

As with serial and direct READ# statements, the variables into which you read data items do not need to have the same names from which you printed the data items on the record, but they must be the same type and the same order as the originals. When the data list is not used, the direct-word READ# resets the record pointer and word pointer to the specified record and word.

The previous example program opened a file to hold information on parts to be stored. The next program opens a file named ORDRPT (for order-point) and then searches the file STOCK for any item with a current quantity less than 10 (lines 120 through 150). When such an item is found, lines 160 through 180 read the part number, print the part number and quantity in file ORDRPT, and output the same data on the standard printer:

```
10   !
20   !         PRINT ORDER-POINT REPORT
30   !
40   CREATE "ORDRPT",1000
50   DIM Part$[20],Desc$[20]
60   ASSIGN #1 TO "STOCK"
70   ASSIGN #2 TO "ORDRPT"
80   INPUT "TODAY'S DATE:";Date$
90   PRINTER IS 0
100  PRINT LIN(5),SPA(20),"PARTS TO REORDER ON ",Date$
110  PRINT "PART NO.  DESCRIPTION   QTY. ON HAND",LIN(2)
120  FOR Part=1 TO 1000
130    READ #1,Part,29;Qty
140    IF Qty>=10 THEN Stokok
150    ! 10 or more items so no re-ordering needed.
160 Rd:READ #1,Part,1;Part$,Desc$
170    PRINT #2;Part$,Qty
180    PRINT Part$,Desc$,Qty
190  Stokok:!
200  NEXT Part
210  PRINT "DONE",LIN(5)
220  END
```

Notice that logical READ# (lines 130 and 160) enables you to read only the items required from a record, thus saving memory space and program execution time. Data is printed into file ORDRPT serially, since it need not be accessed separately.

As another example of direct-word access, the next program can be used to update the cost and quantity data for each record of file STOCK used in the previous programs. As in the first program, this program exits the input loop when the operator inputs "**DONE**" for a part number.

After the operator enters each set of data, the subroutine **Search** looks for the appropriate record in the file. After the record is found, lines 160 and 180 print the new cost and total quantity into that record. Line 190 then outputs the new data on the standard printer.

```
10    !
20    !   UPDATE STOCK FILE
30    !
40    DIM Part$[20], Desc$[20], Stock$[20]
50    ASSIGN #1 TO "STOCK"
60    INPUT "Today's Date: "; Date$
70    PRINTER IS 0
80    PRINT LIN(5), SPA(15); "Parts Received On "; Date$
90    PRINT "PART NO.      COST      QTY. RECEIVED      QTY. ON HA
ND", LIN(1)
100  FOR Part=1 TO 1000
110 In:INPUT "PART NUMBER:";Part$[1,20]
120     IF UPC$(Part$[1,4])="DONE" THEN Done
130     INPUT "QUANTITY RECEIVED:"; Qty1
140     INPUT "UNIT COST:"; Cost1
150     GOTO Search
160  Pr:PRINT #1, Record, 25; Cost1
170     READ #1, Record, 29; Qty
180     PRINT #1,Part,29; Qty+Qty1
```

```
190     PRINT Part$, Cost1, Qty, Qty+Qty1
200   NEXT Part
210  Done: PRINT "DONE", LIN(5)
220        STOP
230  Search:  ! Search for record to update
240   FOR Record=1 TO 1000
250     READ #1, Record,1;Stock$
260     IF Stock$=Part$ then Pr
270   NEXT Record
280   PRINT "!!PART NOT ON FILE!!"
290   WAIT 2000
300   GOTO In
310   END
```

Storing and Retrieving Arrays

Entire arrays can be stored and retrieved by using the array identifier in PRINT# and READ# statements. The syntax for each statement, using direct access, is as follows:

PRINT# *file number* [,*record number*] ;*array name*(*) [,*array name*(*)][,END]

READ# *file number* [,*record number*] ;*array name*(*) [,*array name*(*)]

Arrays are stored and retrieved element by element without regard to dimensionality.

For example:

```
10   OPTION BASE 1
20   PRINTER IS 0
30   DIM A(3),B(3),C(6)
40   A(1)=A(2)=A(3)=65
50   B(1)=B(2)=B(3)=66
60   ASSIGN #1 TO ”TEN”
70   PRINT #1,1;A(*),B(*)
80   READ #1,1;C(*)
90   PRINT C(*);
100  END

     65   65   65   66   66   66
```

Also refer to the MAT PRINT# and MAT READ# statements, covered in page 297 .

Closing a File

The ASSIGN statement is also used to close a file. Any subsequent attempts to access that file number result in an error. Syntax of the ASSIGN statement to close a data file is as follows:

$$\text{ASSIGN}\begin{Bmatrix} \text{* TO \# }\textit{file number} \\ \text{\# }\textit{file number}\text{ TO *} \end{Bmatrix}$$

For example, this statement closes file number 1:

```
310  ASSIGN * TO #1
```

A file is also automatically closed by executing the following operations:

Table 16 **File Table Reset Conditions**

Operation	Files Closed (X) All	Files Closed (X) All Except COM
LOAD		X
GET		X
RUN		X
STOP		X
END		X
SCRATCH		X
SCRATCH V		X
SCRATCH P		X
SCRATCH A	X	
SCRATCH C	X	
Subprogram Return*		

* All files opened in subprogram, but not passed via COM or parameters, are closed.

Purging a File

The PURGE statement erases any DATA file by removing its name from the name table in the directory.

PURGE *file spec*

The disk space used by the file is then available for other uses. For example, the statement to follow purges a file:

```
320  PURGE "DATA"
```

NOTE: Use PURGE with care since all data in a purged file is lost.

NOTE: You can purge an opened file if it belongs to your own process.

The XPURGE statement purges a file of given type. Normally, using PURGE, to remove a file which is not of type DATA, you have to enter a statement like this:

```
COMMAND "!rm "&MAPVOL$("CODE")&"WILLI.PROG"
```

Using XPURGE you would simply enter:

```
XPURGE "WILLI,CODE";PROG
```

If you omit the *type*, it defaults to DATA. So

```
PURGE "WF1"
```

and

```
XPURGE "WF1"
```

are equivalent.

If *type* is OTHR, *no* extension will be added to the filename, e.g.

```
XPURGE "WILLI;OTHR
```

will purge file WILLI.

File Storage Functions

The TYP Function

The TYP (type) function is used to determine what type of data is to be accessed by READ#.

TYP (*file number*)

The possible values returned and their meanings are shown in the following table:

Table 17 **File Table Reset Conditions**

Value	Meaning
0	Unrecognized type.
1	Real-precision number.
2	Total string.
3	End-of-file (EOF) mark.
4	End-of-record (EOR) mark.
5	Integer-precision number.
6	Short-precision number.
7	Dinteger
8	First part of a string.
9	Intermediate part of a string.
10	Last part of a string.
11	HP-UX text file.

If the file number is negative, the record and word pointers are not advanced. If it is positive, however, the pointers move until positioned at something other than an EOR mark. In effect, a negative file number causes a direct read. A positive file number causes a serial read, ignoring EOR marks.

The next program is used to print (serially) various types of data on a new file named TYPE?:

```
10   CREATE "TYPE?",5
20   ASSIGN "TYPE?" TO #1
30   INTEGER Int
40   SHORT Short
50   Real=1E99
60   String$="STRING"
70   Int=12345
80   Short=654321
90   PRINT #1;Real,String$,Int,Short
100  END
```

Now run this program which uses the TYP function to identify each data item, and then stores it into the appropriate type of variable:

```
10         ASSIGN #1 TO "TYPE?"
20         INTEGER Int
30         SHORT Short
40         READ #1,1
50 Type:   !
60         ON TYP(-1) GOTO Real,String,Eof,Eor,Integer,Short
70 Real:   !
80         READ #1;Real
90         PRINT Real,"is a real-precision value."
100        GOTO Type
110 String:  !
120        READ #1;String$
130        PRINT String$,"is a string variable."
140        GOTO Type
150 Eof:   !
160        PRINT "EOF mark is next."
170        STOP
180 Eor:   !
190        PRINT "EOR mark is next."
200        STOP
210 Integer:  !
220        READ #1;Int
230        PRINT Int,"is an integer-precision value."
240        GOTO Type
250 Short:  !
260        READ #1;Short
270        PRINT Short,"is a short-precision value."
280        GOTO Type
290        END
```

Line 40 sets the record pointer to record 1 of file TYPE?. The computed GOTO statement (line 60) branches the program to one of six labels, depending upon the value returned by TYP(-1). This statement is executed before the READ# to determine which type of data is to be read next. Here is the printout:

```
1.00000000000E+99   is a real-precision value.
STRING              is a string variable.
 12345              is an integer-precision value.
 654321             is a short-precision value.
EOR mark is next.
```

Notice that if the record pointer had been set to any other record of file TYPE? (for example, READ#1,4) the TYP function would return 3, indicating an EOF mark. Remember that each record is filled with EOFs when it is CREATEd; the EOFs are replaced by data via PRINT# statements.

The SIZE Function

The SIZE function returns the size of the specified file.

SIZE (*file number*)

A positive file number, for all files except HP-UX text files, returns the file size expressed in logical records. For HP-UX text files (TYP 11), the file size is expressed in bytes.

A negative file number, for all files except HP-UX files, returns the logical record size in words (one word equals two bytes). For HP-UX text files, a negative file number returns the number 1.

The REC Function

The REC (record) function returns the current position of the data pointer within the specified file.

REC (*file number*)

A negative file number always returns 0.

The WRD Function

The WRD (word) function returns the current position of the word pointer for the specified file.

WRD (*file number*)

A value of 1 indicates the first word of the record.

NOTE: The WRD function cannot be used on an regular text file (TYP 11). Doing so will result in error 69 (”operation not allowed for this file type”).

NOTE: The AVAIL and HOLE functions are no longer valid.

The FNAME$ Function

The keyword

FNAME$ (*expr*)

returns the file name associated with the given file number.

For example:

```
ASSIGN #1 TO"foo,TMP"
DISP FNAME$(1)
```

```
-> /tmp/foo.DATA
```

An absolute path-name is returned, depending on the platform.

Trapping EOR and EOF Conditions

An error usually results from encountering one of two conditions—a logical or physical EOF during READ# or an EOR during direct READ#. The ON END# statement can detect those errors and cause a branching operation.

$$\text{ON END\# } \textit{file number} \begin{Bmatrix} \text{GOTO } \textit{line id} \\ \text{GOSUB } \textit{line id} \\ \text{CALL } \textit{subprogram name} \end{Bmatrix}$$

In some of the previous programs, for example, **ERROR 59** appears after the last item is accessed, telling you that the end of file has been reached. This error message can be avoided by including an ON END# statement in the program.

Here is a modified version of the example program used when “Repositioning the Record Pointer”. The program ends when the end of file is reached.

```
10   CREATE "DATA15",15
20   ASSIGN #1 TO "DATA15"
30   READ #1,8
40   ON END #1 GOTO Exit
50   FOR V=1 TO 50
60   PRINT #1;V
70   NEXT V
80 Exit:  ! end of file found.
90   PRINT "END OF DATA!"
100  END
```

As another example, the next program prints four data items into each record of files DATA.1 and DATA.2. When DATA.1 is filled, reading the EOF mark causes a branch to the subroutine Newfile, which opens file DATA.2 for the rest of the PRINT# routine.

```
10   CREATE "DATA.1",20
20   CREATE "DATA.2",20
30   ASSIGN #1 TO "DATA.1"
40   ON END #1 GOSUB Newfile
50   FOR R=1 TO 100
60     PRINT #1;R,R^2,R^3,R^4
70   NEXT R
80   STOP
90 Newfile:   !  open new file.
100  ASSIGN #1 TO "DATA.2"
110  RETURN
120  END
```

ON END is disabled during INPUT, LINPUT and EDIT statements. ON END can interrupt ON ERROR and ON KEY routines. ON END cannot be executed from the keyboard.

An ON END is deactivated with the OFF END# statement:

OFF END# *file number*

NOTE: Under the HP 260 environment the ON END # command could be used with the PRINT # statement to signify when a file was filled. In the HP-UX environment files are dynamic, rather than static, and printing will continue until the entire disk is full if a limit is not written into the program. Therefore, make sure you change any converted HP 260 programs that make use of ON END # and PRINT #, in this way.

Data Storage Requirements

When storing data, it is possible to optimize the use of your storage medium by minimizing the amount of unused space. The best way to do this is to create your files so they are suited to the amount of data you wish to store and are suited to storage medium capacities.

The following tables indicate how many bytes are needed to store each type of variable.

Simple Variables

Real precision	8 bytes.
Short precision	4 bytes.
Integer precision	4 bytes.
Dinteger	6 bytes.
String	1 byte per character (rounded to an even integer) + 4 bytes (+ 4 additional bytes each time string crosses into a new defined record).

Array Variables

Real precision	8 bytes x dimensioned number of elements.
Short precision	4 bytes x dimensioned number of elements.
Integer precision	4 bytes x dimensioned number of elements.
String	4 bytes per element + total needed for all strings as defined above.

By summing up how many bytes of storage your data requires, you can tailor your file and logical record lengths to suit your needs and minimize waste.

Multi-User File Protection

The LOCK# statement restricts data file access to the task which executes it.

LOCK# *file number* [,*wait variable*]

The optional wait parameter indicates whether the computer should wait for access to the specified file while it is already locked via another task. If the value of the wait parameter is 0, the program will wait until the file is available (unlocked). If the wait parameter is not 0, the LOCK# statement will be aborted if the file is already locked. The default value is 0.

The value of the wait variable changes to 0 when the lock operation is successfully completed. If the lock operation is aborted, the wait variable changes to 1.

To release a data file for use by other tasks, use the UNLOCK# statement:

UNLOCK# *file number*

Copying a File

The COPY statement is used to copy information from one DATA file into another file. Syntax for this statement is as follows:

COPY *source file spec* TO *destination file spec*

Execution of the COPY statement causes all records of a DATA file to be copied. A check of the name of the destination is made; an error is given if the name is already present. If not, a file of the same characteristics as the source file is created. The same directory can be both source and destination.

For example, this statement copies a file named DATA1 from the current volume to a new file named DATA2 on a volume labeled BACKUP:

```
160  COPY "DATA1" TO "DATA2,BACKUP"
```

The contents of a spool file, a file created by a previous printer assignment statement, are dumped to an output device, such as the display or a printer, by specifying only the *source file spec* in a COPY statement:

COPY *source file spec*

Use of spool files is covered in page 249 .

Renaming a File

The RENAME statement is used to give a DATA file a different name. Syntax for this statement is as follows:

$$\text{RENAME } \textit{old file spec} \text{ TO } \left\{ \begin{array}{l} \textit{new file name} \\ [\textit{new file name}] \end{array} \right\} , \textit{volume label}$$

For example:

```
RENAME "DATA1" TO "DATA2"
```

Renames the file DATA1 to DATA2.

```
RENAME "DATA1" TO "DATA2,TEST"
```

Renames DATA1 to DATA2 and moves it to the volume labelled TEST.

```
RENAME "DATA1" TO ",TEST"
```

Moves DATA1 to the volume labelled TEST.

9

Output Operations

This chapter covers audible, display, and printer output operations. How to format printed output is also covered. The following statements are discussed in this chapter:

PRINTER IS Defines the standard printer (the device used for all PRINT and PRINT USING output).

SYSTEM PRINTER

IS	Defines the output device for LIST, FETCH and CAT operations.
PRINT ALL IS	Defines the output device for all messages normally shown on the display.
REQUEST	Reserves use of a specified device (PORT) for one process.
RELEASE	Cancels any REQUEST for exclusive use of a peripheral device (PORT) by that process.
BEEP	Outputs an audible signal to the operator.
DISP*	Displays text and variables.
LDISP*	Like DISP, except an entire display line is used.
REFRESH ON/OFF	Redraws the display.
CURSOR*	Controls the display cursor and enhancements.
PRINT	Outputs text and variables on the standard printer. Also, it is an output control function used with DISP and PRINT.
LIN, PAGE, SPA and TAB	Output-control functions used with DISP and PRINT.
IMAGE	Lists specifiers controlling form of items output with PRINT USING and DISP USING.
PRINT USING	References image specifiers while outputting each item to the standard printer.
DISP USING*	Uses the same field specifiers mentioned above for display outputs.

* This statements are available on HP-UX platform, only.

Restrictions on the Use of ASCII Control Characters

NOTE: This function is only available on HP-UX platforms.

You can use DISPLAY FUNCTIONS to store non-printing ASCII control characters; however, be careful when outputting these control characters to a screen or printer, while in DISPLAY FUNCTIONS mode.

If you turn on DISPLAY FUNCTIONS and then press RETURN, the carriage-return/line-feed character is displayed on the screen and not executed. For example:

```
10 DISP "C/r L/f" ! Display carriage-return/line-feed
```

You can list this line to your terminal; however, the display will show the following:

```
10 DISP "~~"
```

ASCII control characters are displayed as a tilde (~), if you list your program.

Selecting Output Devices

Information output by Eloquence takes three forms—PRINT outputs, SYSTEM outputs, and PRINT ALL outputs. All output is automatically shown on the display screen after Eloquence is started. The display is called the default output device. Alternately, you may specify other devices to output each type of information by using the statements introduced here:

Table 18 **File Table Reset Conditions**

Output Type	Operations Affected	Controlled By
PRINT	PRINT, PRINT USING	PRINTER IS
Eloquence SYSTEM	LIST, CAT, TRACE, FETCH, COMMAND “!OS cmd”	SYSTEM PRINTER IS
PRINT ALL	All displayed information	PRINT ALL IS

Printer and Port Numbers Vs. Device Addresses

Each device connected to the computer responds to a unique address. The address is mapped to a number in a configuration file. For printers this number can be from –2 to 7 and from 11 to 99. For ports this number can be from 11 to 20. Three numbers are reserved, as shown in the following table; the table also shows what devices are associated with the reserved numbers. For more details on how to configure printer/ports, see the instructions in chapter 2.

Printer Number	Explanation
8	Display.
9	Null device (Bit Bucket) - all output data is ignored.
10	Local printer.

Printers

System Printer

The system printer has the following characteristics:

- Accessible from every task and user.
- Supports the standard ASCII characters in the range 32 through 126 and most National characters. (ASCII characters 32 through 126 are listed in page 391 .)

Eloquence supports the “Underline” enhancement and all “PRINT” statements. Individual printer features can be used with the aid of the “PRINTER ISTRANSPARENT” option, which allows all codes to be sent to the printer.

Local Printer

NOTE: Local Printers are available with a character oriented user interface, only.

The local printer has the following characteristics:

- Printer number is 10.
- Connection to a terminal.
- Accessible from primary tasks only.

You are also provided with the “PRINTER ISTRANSPARENT” option. However, some codes (for example, 0, 5, and 127 on the HP 700/92) are stripped by the terminal. The remaining features of the local printer are the same as for the system printer.

All UX printers are supported with Eloquence when connected to an HP 700/92, HP 700/94, or HP Vectra.

The PRINTER IS Statement

The PRINTER IS statement assigns the output device for PRINT and PRINT USING operations. Refer to page 254 for information on the syntax of the PRINTER IS statement.

The SYSTEM PRINTER IS Statement

The SYSTEM PRINTER IS statement assigns the output device for all successive SYSTEM outputs. This includes CAT, LIST, FETCH, COMMAND “!”, single-step, and TRACE operations. Refer to page 254 for information on the syntax of the SYSTEM PRINTER IS statement.

The PRINT ALL IS Statement

To obtain a permanent copy of all operator and system interactions, use the PRINT ALL IS statement to specify a device other than the display. Syntax for the statements related to output device assignment is as follows:

```
                          { printer no     }
{ PRINTER IS          }   { file specifier }
{ SYSTEM PRINTER IS   }   { STDOUT         }   [,WIDTH] [,TRANSPARENT]
{ PRINT ALL IS        }   { STDERR         }
                          { CONSOLE        }
                          { TTY            }
```

Printer number is an integer which states the number of a printer, port, or display screen. The possible printer numbers are –2 to 99. The printer number for the display screen is 8. The printer numbers for printers and ports are assigned in the configuration files.

Line width is a numeric expression (–1, or 20 to 264) which specifies the number of characters per line output by PRINT and PRINT USING. If omitted, line width assumes the previously set value. If no value was previously set or the system was turned off and on, the line width value defaults to 80 for terminals and 132 for other devices. Specifying –1 sets an infinite line width.

File specifier is a spool file, where all outputs are recorded in hp-ux format. For more details, see page 292 at the end of this chapter.

Each successive PRINTER IS statement cancels the previous one. Note, however, that the line width parameter of any future PRINT ALL IS or SYSTEM PRINTER IS statement will also determine the line width for the current PRINTER IS statement for that device. (This allows multiple users to each assign their own line width to the same printer.)

NOTE: If printer number is defined as FILE in the configuration file, it is locked into the specific eloqcore process to avoid printing conflicts.

The REQUEST Statement

The REQUEST statement requests the use of a printer or port. Syntax for this command is as follows:

REQUEST *printer/port number* [,*return variable*]

The *printer/port number* is an integer expression evaluating to the number of the requested printer or port. Valid printer/port numbers are listed in this chapter under page 252 . The parameter *return variable* should be replaced with a valid variable name, if this parameter is desired.

The REQUEST statement first checks to see if the requested device is a printer or a port. Printers and ports are defined in the user, group, and global configuration files using the PRINTER and PORT statements.

If the *return variable* is omitted and the device is already reserved by another program, **`ERROR 131`** results. If the requested device is not defined in the user, group, or global configuration file, **`ERROR 132`** results.

If the return variable is pre-set to 0, the request is carried out in WAIT mode: that is, if requesting an already reserved port, program execution will wait until the port becomes avaliable. If the return variable is pre-set to 1, there will be no wait and [1] is returned (see below).

If the requested device is a port, the corresponding HP-UX device file is locked, reserving it for your use.

The value returned to the *return variable*, if present, is based upon the following criteria:

- 0 returned if request a printer.
- 0 returned if request a port that is available.
- 1 returned if request a port that is already reserved.

NOTE: The REQUEST statement is different from the REQUEST # statement. REQUEST # has to do with multiple task programming.

The RELEASE Statement

The RELEASE statement cancels any previous REQUEST for the specified device. Note that it is not necessary to RELEASE a printer. This is because a printer is not locked at the time the REQUEST is started. Syntax for the RELEASE statement is as follows:

RELEASE *printer/port number*

NOTE: The RELEASE statement is different from the RELEASE # statement. RELEASE # has to do with multiple task programming.

Audible Output (BEEP)

The BEEP statement creates a brief audible tone which can be used in a number of ways. The syntax is as follows:

BEEP

BEEP can signal that a particular computation or program segment is complete. It can also be used to audibly indicate that the computer is ready for input, so that the operator does not have to remain at the keyboard.

Here is an example:

```
250  FOR I=1 TO 25
260    BEEP
270    INPUT "ENTER VALUE:";Value(I)
280  NEXT I
```

In this case, a beep signals the operator when the program is ready for input. Another practical use for BEEP is to signal the operator when a data-entry error occurs.

As another example, here is a musical sequence to signal the end of a program:

```
10   FOR Beep=1 TO 7
20     READ Delay
30     WAIT Delay*10
40     BEEP
50   NEXT Beep
60   DATA 80,40,15,15,40,80,40
70   END
```

Displayed Output (DISP)

NOTE: The DISP statement is available on HP-UX platforms, only.

The DISP (display) statement outputs text and variables on the display. Syntax is as follows:

DISP [display list]

The *display list* can contain one or more of the following:

- Variable names.
- Array identifiers.
- Numeric expressions.
- String expressions.
- User Defined Types.
- TAB, SPA, LIN and PAGE functions (covered later).

Each item must be separated by either a comma or semicolon.

Here are some examples:

```
10   X=3.5
20   E$="SQUARED EQUALS"
30   DISP "X ";E$[9];X,"X ";E$;X^2

X EQUALS 3.5                 X SQUARED EQUALS 12.25

40   DISP 1,2,3,4
50   DISP 1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18

1                   2                   3                   4
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18
```

The difference in spacing between numbers is controlled by use of commas and semicolons. When an item is followed by a comma, it is left-justified in a field 20 characters wide. Two or more commas after an item cause one or more character fields to be skipped. For example:

```
60   DISP 123456, ,654321

123456                                  654321
```

When an item is followed by a semicolon, no additional blanks are output. Remember that every number has a leading blank or minus sign and a trailing blank for spacing. For example:

```
10   STANDARD
20   GOSUB Disp
30   FIXED 3
40   GOSUB Disp
50   FLOAT 5
60   GOSUB Disp
70   STANDARD
80   GOTO 110
90 Disp:   DISP 123;.456;−789;−1.23E45
100        RETURN
```

```
123  .456 −789 −1.23e+45
123.000  .456 −789.000 −1.23E+45
1.23000E+02  4.56000E−01 −7.89000E+02 −1.23000E+45
```

When the display list ends with a comma or semicolon, any future DISP statement is appended to the current display line. For example:

```
110   INPUT "ENTER TODAY'S DATE:";Date$
120   DISP "TODAY'S DATE IS: ";
130   DISP Date$
```

```
TODAY'S DATE IS: Sept 22nd
```

If the output line is longer than the current display width, a carriage-return line-feed (CRLF) is normally output after every 80th character. This can be altered by assigning another line width via PRINTER IS. For example:

```
140   PRINTER IS 8,WIDTH(40)
```

sets the display line width to 40 characters. In this case, the new line width is in effect for successive DISP, PRINT, PRINT USING, and DISP USING output.

Below you find two examples concerning printing of User Defined Type variables.

In the example below, the Comment$ member variable from the Phone1 variable is displayed.

```
Phone1.Comment$="*Fancy Comment*"
DISP Phone1.Comment$
```

In addition to accessing single variables, you can specify the whole variable at once.

The example below displays all member variables of Phone1:

```
DISP STRUCT Phone1
```

The LDISP Statement

NOTE: The LDISP Statement is available on the HP-UX platform, only.

The LDISP (line display) statement outputs text and variables on the display, like DISP, but the output is placed on a complete display line, rather than in separate fields. Syntax for the LDISP statement is as follows:

LDISP [*display list*]

The display list can contain the same item types as with DISP. Each item must be separated by either a comma (open spacing) or a semicolon (pack items together). The LDISP operation skips over any protected display lines and outputs the list on the first unprotected line found. See page 274 later in this chapter.

Here is an example sequence which uses LDISP to replace a previously-output message with another:

```
110  DISP "C/H C/S"! Cursor Home/Clear Screen characters
120  INPUT "↓↓↓↓↓ENTER TODAY'S DATE:";Date$
130  DISP "↑↑"
140  LDISP "SELECT PROCEDURE BELOW:"
```

Line 110 outputs display-control characters (C/H C/S - the exact representation of these characters may vary from screen to screen) used to clear the display. Line 120 uses display control characters to position the cursor before displaying each string (↓↓↓↓↓). Line 130 then repositions the cursor so that line 140 will erase the input-prompt line and display the last message. Note that the CURSOR statement can be used in place of display-control characters to position the cursor. See page 268 later in this chapter.

The display-control characters are output in a DISP statement (line 130), rather than in the LDISP statement. This is because LDISP is a “line output” operation (like CATALOG and LIST), so it does not execute display-control characters; it displays the actual characters instead. This allows you to use these characters in displayed messages, as shown in the following example:

```
190  LDISP "TODAY'S DATE IS →ÆÆ";Date$
```

It also allows recalling the entire contents of a string variable containing display-control characters. For example, executing the line below clears the display and then displays ABC:

```
A$="C/H C/S ABC"
```

Executing an expression from the keyboard actually performs an “implied LDISP”. To recall the entire value of A$, do not execute DISP A$ (an item output operation), but execute LDISP A$ (a line output operation).

POPUP BOX

The POPUP BOX function will superimpose a box on your screen and wait for user response.

Syntax:

```
POPUP BOX [xpos,ypos,] Image_def [,Return]
```

```
Image_def: "[Heading] [textlines|...|...][button1|...]"
```

Example:

```
POPUP BOX "[Heading][Text line #1|Text line #2][YES|NO ]"
```

```
----------------
|   HEADING     |      <- header
|               |
| Text line #1  |      <- text
| Text line #2  |
|               |
| [YES] [NO ]   |      <- "buttons"
----------------
```

This box consists of 3 parts:

header	output in reverse
text	up to 19 text lines separated by "|" character (header and text together can be up to 20 characters.)
buttons	up to 5 buttons separated by "|" character

Header and text is optional and can be empty.

```
<xpos>
```

and

```
<ypos>
```

are optional. If the are omitted, the box is centered on the screen. If

```
<xpos>
```

or

```
<ypos>
```

is <= 0 then this value will be ignored and default will be used.

You have to specify at least one "button". The user can select a "button" in several ways:

- Press RETURN. This will select the “default” button (reverse one). By pressing backspace, space or using the cursor keys you can move the default button.
- Press the first character from the button text (in this example either ’y’ or ’n’).

If you specify the return variable it’s possible to preset the default button (0 = 1st, 1 = 1st, 2 = 2nd). The user’s choice will be returned in this variable.

You will get ERRORS 870 to 875 if you specify an invalid box position or an invalid box “image”.

NOTE: The screen buffer will be saved and restored. Program execution will be suspended until user selects a “button”. In BACKGROUND, the default button will be selected.

NOTE: When the ".driver" attribute is set, the POPUP BOX is mapped to a dialog window.

The REFRESH Statement

NOTE: The REFRESH statement is available on the HP-UX platform, only. It has an effect on character oriented user interfaces, only.

The REFRESH statement refreshes the display of the terminal on your system. Its syntax is as follows:

REFRESH [ON]

REFRESH OFF

REFRESH or REFRESH ON redraws the display of the terminal associated with the task executing the statement. The REFRESH statement is similar in effect to pressing CTRL L on the terminal's keyboard.

Screen refreshment is done in two steps. First the internal screen buffer will be maintained. Then terminal image will be updated from internal screen buffer.

REFRESH OFF will suppress updating of terminal image. So it's possible to save some i/o time and prevent terminal flicker.

REFRESH ON will update terminal image from internal screen buffer and re-establish automatic updating of terminal image. REFRESH ON is executed implied every time Eloquence waits for a user interaction (INPUT, LINPUT, WAIT, PAUSE), because the screen has to be up to date.

Output Functions

Four functions are available to increase output formatting capabilities—TAB, SPA, LIN, and PAGE. The functions can be used in DISP, LDISP, and PRINT statements. Each function must be followed by either a comma or a semicolon (it makes no difference here).

NOTE: The statements TAB, SPA, LIN, and PAGE work with DISP and LDISP on the

The TAB Function

The TAB function causes the next item in the list to be output beginning in the specified column. Syntax is as follows:

TAB (*charcter position*)

The character position is any numeric expression, and is rounded to an integer. If it is less than 1, it defaults to 1. For example:

```
160  DISP 1;2;3;TAB(10),"BEGINNING OF 10th COLUMN"
```

```
1 2 3          BEGINNING OF 10th COLUMN
```

If the specified column has already been filled, a CRLF is output, and then the TAB is completed. For example, if the line above is changed to:

```
170  DISP 1,TAB(10),"BEGINNING OF 10th COLUMN"
```

```
1
          BEGINNING OF 10th COLUMN
```

a CRLF would be output after 1 (notice that the comma causes it to be output in a 20-column field) and the text appears on the next line.

When the character position specified is greater than the number of columns in the standard printer, it is reduced by this formula:

character position MOD N

N being the number of columns specified as the standard printer width. If the *character position* is a multiple of the printer width, this formula returns a 0. In this case, the item is output in the last column which equals the printer width. For example, with a display width of 80:

```
180  DISP TAB(10),1;TAB(90),2;TAB(170),3
          1
          2
           3
```

The SPA Function

The SPA (space) function outputs the specified number of blank spaces, up to the end of the current line. Syntax is as follows:

SPA (*number of spaces*)

Here is an example:

```
190   DISP 1;SPA(10);2;SPA(10);3
200   DISP 1,SPA(10),2,SPA(10),3

1          2          3
1                      2                      3
```

The number of spaces is any positive numeric expression from 0 through 32767, and is rounded to an integer. If it specifies more blanks than remain in the line, the next item begins the next line. For example:

```
210   Ast$="**********"
220   DISP Ast$;TAB(70);Ast$;SPA(20);Ast$

**********                                                        **********
**********
```

The LIN Function

The LIN function causes the specified number of linefeeds to be output. Syntax is as follows:

LIN (*number of linefeeds*)

The number of linefeeds is any numeric expression, and is rounded to an integer. Its range is from –32768 through 32767.

Here is an example:

```
230   INPUT "DATE:";Date$
240   DISP "DATE:";Date$,LIN(5),"END OF PROGRAM",LIN(5)

DATE:April 23rd




END OF PROGRAM
```

When the number of linefeeds is positive, a carriage return is also output. When 0 is specified, only a carriage return is output.

When the number of linefeeds is negative, no carriage return is output and the number of linefeeds output equals the absolute value of the expression. For example:

```
10   DISP "TODAY";LIN(−2);"IS";LIN(−2);"FRIDAY"

TODAY

    IS

      FRIDAY
```

The PAGE Function

The PAGE function causes a form feed to be output, so further printing can begin on a new page or at the top of the next form on devices that respond to ASCII form feed (CHR$(12)). The syntax is as follows:

PAGE

Here is an example:

```
130 Heading:  ! print catalog header.
140   PRINT PAGE,Cat$(1)
150   PRINT Cat$(2);LIN(1)
```

When the standard printer is the display, PAGE scrolls the display buffer up to position the cursor at the top of the display.

Here is a short program which uses some of the output function to repeatedly output a string diagonally across the display:

```
10   DISP PAGE
20   Column=1
30   FOR Line=1 TO 10
40     DISP SPA(Column);"Eloquence"
50     Column=Column+3
60   NEXT Line
70   END

Eloquence
   Eloquence
      Eloquence
         Eloquence
            Eloquence
               Eloquence
                  Eloquence
                     Eloquence
                        Eloquence
                           Eloquence
```

Display Enhancements

NOTE: Display Enhancements are available on the HP-UX platform, only.

The CURSOR Statement

The CURSOR statement allows program control of the display cursor and modification of a block of displayed characters. The syntax is as follows:

CURSOR *item list*

Any combination of these control items can be used:

(*X pos*, *Y pos*)	Cursor position.
IV	Set inverse video.
BL	Blinking character.
UL	Underline field.
HB	Set half bright display.
RE	Reset field enhancement.
PL	Protect lines.
UPL	Unprotect lines.
IF	Specify input field.
RIF	Reset input field.
OF	Specify output field.
ROF	Reset output field.
PALL	Protects all lines of the display buffer.

Each CURSOR item must evaluate to an integer greater than 0. There now follows a description of each control item.

The Eloquence Forms software provides additional CURSOR control items to assign input/output field numbers. See the *Eloquence Forms Manual* for details.

Set Cursor Position

(*X pos*,*Y pos*)

(*X pos*)

(*Y pos*)

This item allows the cursor to be positioned anywhere within the display buffer. The X position, if specified, determines the character position within a display line. An X position greater than 80 will "wrap around" to the second row of the same line. (Remember, the display buffer width is initially 160 characters but can be changed by using PRINTER IS.) If the X position is not specified, the current cursor X position will be used.

The Y position, if specified, determines the number of lines down from the HOME position, which is the top of the display buffer. Any Y position below the last line will add new lines as required. If it is not specified, the current Y position is used.

Note that the cursor must remain on the display at all times, so the display may scroll up or down appropriately to satisfy this condition. The cursor may be repositioned many times in a single CURSOR statement, with other functions being performed at the intermediate positions.

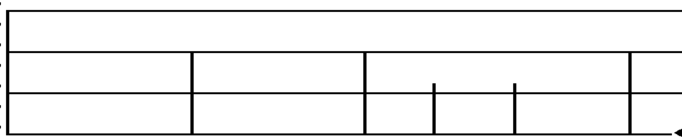
This program generates the same display as that illustrating the PAGE function, but CURSOR is used here in place of output functions:

```
10    CURSOR (1,1)
20    Column=1
30    FOR Line=1 TO 10
40      CURSOR (Column,Line)
50      DISP "Eloquence"
60      Column=Column+3
70    NEXT Line
80    END
```

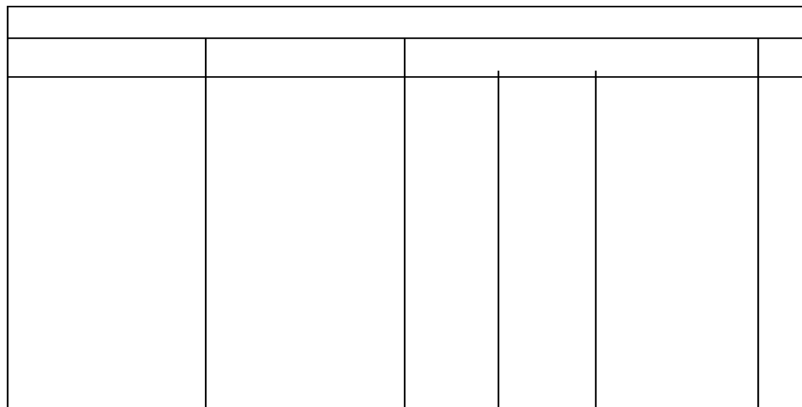
Line 10 positions the cursor to the top of the display buffer, whereas the PAGE function in the earlier program only sets the cursor to the top of the next page of the buffer.

Here is a more practical example of CURSOR use:

```
LIST
10       DIM Form$(10)[90]
20       Form$(1)="
30       Form$(2)="
40       Form$(3)="
50       Form$(4)="
60       Form$(6)="
70       Form$(7)="
80       Form$(8)="
90       Form$(5)=Form$(4)
100 Draw_form:!
110      DISP ""
120      FOR L=1 TO 6
130         CURSOR (10,L)
140         DISP Form$(L)
150      NEXT L
160      FOR L=7 TO 19
170         CURSOR (10,L)
180         DISP Form$(7)
190      NEXT L
200      CURSOR (10,20)
```



The Assign routine assigns line-drawing characters to elements of a string array, one line per string. Then the Draw routine displays the strings to create the form. Notice that the CURSOR statements reposition the cursor before each DISP. Here is the resulting display:



CURSOR can also be used to accurately position the display cursor while adding each title to the form:

```
210 Add_titles: !
```

```
220       CURSOR (31,2)
230       DISP "IN-STOCK REPORT"
240       CURSOR (13,4)
250       DISP "Part No.";
260       CURSOR (27,4)
270       DISP "Description";
280       CURSOR (49,4)
290       DISP "On Hand"
300       CURSOR (67,4)
310       DISP "On";
320       CURSOR (66,5)
330       DISP "Order";
340       CURSOR (12,5)
350       DISP "XXXXX-XXXXX";
360       CURSOR (41,5)
370       DISP "Qty"
380       CURSOR (48,5)
390       DISP "Cost"
400       CURSOR (56,5)
410       DISP "Tot. Cost";
```

Notice that a semicolon follows each DISP statement here, to suppress additional spaces which may print over the line-drawing characters. (Unless you meticulously plan the position of each line-drawing and title character in advance, creating forms of this kind is, at best, a trial and error experience.)

Now the form looks like this:

IN-STOCK REPORT					
Part No. XXXXX-XXXXX	Description		On Hand		On
		Qty	Cost	Tot. Cost	Order

The XPOS and YPOS Functions

XPOS

YPOS

These functions return numeric values according to the current cursor position. YPOS returns the current line number, relative to the first line of the display buffer (home). XPOS returns the character position of the cursor in the current line. Both values will always be greater than 0. These functions may be used anywhere in Eloquence expressions, but they are particularly useful in conjunction with the CURSOR statement for relative movement of the cursor or for monitoring user manipulation of the cursor.

Here is another version of the display program shown under the description of the PAGE function.

```
10    CURSOR (1,1)
20    FOR Line=1 TO 10
30      DISP "Eloquence";
40      CURSOR (XPOS-2,YPOS+1)
50    NEXT Line
60    END
```

Field Enhancements

$$\left\{\begin{bmatrix} \text{IV} \\ \text{BL} \\ \text{UL} \\ \text{HB} \\ \text{RE} \end{bmatrix}\right\} \textit{field length}$$

These items affect the characters starting at the current cursor position. The parameter *field length* specifies the number of characters to be enhanced. The cursor will be left in its original position at the first enhanced character. The RE item cancels any previous enhancement items for the specified display field.

The characters used by the computer to control the display enhancements are supplied in page 383 .

Here is a program that demonstrates all of the field enhancements:

```
10          DISP "Cr/H Cl/S"  ! Cursor home and clear display.
20 Inverse: ! fill screen with inverse video fields.
30          FOR Line=1 TO 24
40          CURSOR (1,Line),IV(80)
```

```
50          DISP "INVERSE VIDEO"
60          NEXT Line
70 Halfbright: ! add fields in halfbright video.
80          WAIT 1000
90          FOR Line=2 TO 24 STEP 2
100         CURSOR (1,Line),HB(80)
110         DISP "HALF BRIGHT"
120         NEXT Line
130         WAIT 1000
140 Blinking: ! add alternate blinking fields.
150         FOR Line=2 TO 24 STEP 2
160         CURSOR (40,Line),BL(40)
170         DISP "BLINKING FIELD"
180         NEXT Line
190         WAIT 1000
200 Underline: ! add underlined fields.
210         FOR Line=1 TO 23 STEP 2
220         CURSOR (40,Line),UL(40)
230         DISP "UNDERLINED FIELD"
240         NEXT Line
250         WAIT 5000
260 Reset: ! reset all display enhancements.
270         FOR Line=1 TO 24
280         CURSOR (1,Line),RE(80)
290         NEXT Line
300         END
```

Line 10 sets the cursor at buffer home and clears the buffer. The Inverse routine fills the display with inverse video fields and labels each line. The Halfbright routine then fills every other line with half bright fields. The next two routines fill the right half of the display with blinking and underlined labels.

The display now looks like the following:

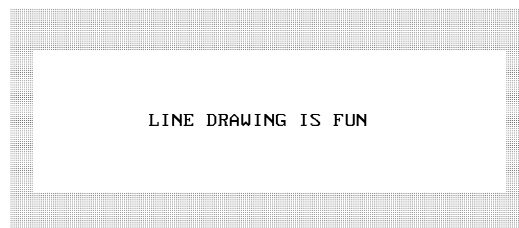
```
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
HALF BRIGHT                                  BLINKING FIELD
INVERSE VIDEO                                UNDERLINE FILED
```

Finally, the last routine resets (clears) all field enhancements, leaving only the displayed labels.

The next program uses CURSOR to draw a box of inversed-video characters. Within the box is a blinking, underlined message.

```
10   DISP "Cr/H Cl/S"! Cursor home and clear screen
20   CURSOR (18,5),IV(44)
30   CURSOR (18,6),IV(44)
40   CURSOR (18,7),IV(2),(60),IV(2)
50   CURSOR (18,8),IV(2),(60),IV(2)
60   CURSOR (18,9),IV(2),(60),IV(2)
70   CURSOR (18,10),IV(2),(30),UL(20),BL(20),(60),IV(2),(30)
80   DISP "LINE DRAWING IS FUN!"
90   CURSOR (18,11),IV(2),(60),IV(2)
100  CURSOR (18,12),IV(2),(60),IV(2)
110  CURSOR (18,13),IV(2),(60),IV(2)
120  CURSOR (18,14),IV(44)
130  CURSOR (18,15),IV(44)
140  END
```

Here is the resulting screen:



Protect and Unprotect Lines

CURSOR (*x,y*) PL (*number of lines*)

CURSOR (*x,y*) UPL (*number of lines*)

CURSOR (*x,y*) PALL

CURSOR (*x,y*) UPALL

To prevent input or output fields on the screen from being overwritten, PL and PALL allow lines of the screen to be declared as unmodifiable, or protected. UPL and UPALL allow "unprotecting" screen lines. An unprotected line could be modified by the keyboard, a program, or the system. CURSOR PL(x) protects x number of lines starting with the current cursor position. PALL protects all lines in the display buffer.

CURSOR (x,y) PL(z) unprotects z number of lines starting with the current cursor position (x,y). CURSOR (x,y) UPALL unprotects lines in the current display buffer.

Note that protected lines will not be stripped from the top of the display buffer. This can be of use in fixing the HOME position if an absolute reference is desired for the cursor position function. However, the lines must then be explicitly unprotected to allow the system to recover unnecessary display areas.

Input and Output Fields

CURSOR (*x,y*) IF (*field width*)

CURSOR (*x,y*) OF (*field width*)

CURSOR (*x,y*) RIF (*field width*)

CURSOR (*x,y*) ROF (*field width*)

Fields within a line can be declared as input-only or output-only. IF, OF, RIF, and ROF work on lines that have been protected. (For more information on protected lines, refer to page 274 in this chapter.)

An input field can receive input from the keyboard only. Your program must not attempt to print in an input field. In the statement CURSOR (x,y) IF(z) there is an input field of length z starting at cursor position (x,y). In an INPUT or ENTER statement, one input variable is assigned to each input field. When entering the data, press TAB to move the cursor to the first character of the next input field. Press SHIFT TAB to move the cursor to the previous input field.

An output field can only receive output from the computer. Your program must not expect input from an output field. Each sequential output item goes to the next sequential output field or the next unprotected line, whichever comes first. After output to an output field, the cursor rests at the next character position until moved by another CURSOR, input, or output operation.

Output fields only receive output, such as from DISP, PRINT, and the prompt from an INPUT. Line output operations such as LIST, LDISP or CAT do not alter protected lines. The line output occurs on the next unprotected line.

The next example is part of a program to initiate new customer accounts. The New Customer form is first displayed; then underscores are used to fill the input fields. Then the input fields and one output field are defined. Line 210 protects the entire display page. The input-data routine suspends program execution to permit user entry and then enters data from each field using ENTERs. An account number is then assigned and placed in the output field with a DISP. The remainder of the program sets up the customer account.

```
10       DISP Name$[35],Address$[50],Zip$[5],Phone$[11]
20 Display_form:  !
30       DISP "                NEW CUSTOMER FORM    "
40       DISP "     NAME       ____________     ,    ____________     ,
            "
____________
50       DISP "           last                first             midd
le"
60       DISP "     ADDRESS      ____________   "
70       DISP "                  street"
80       DISP "                  ____________  ,  ____________   "
90       DISP "                  city              state"
100      DISP "                  ____________     (  ____________    )
            "
____________
110      DISP "                  zip      telephone"
120      DISP "    ACCOUNT NO."
130      DISP " ____________
140      DISP "ENTER INFORMATION INTO EACH BLANK FIELD. USE THE "
150      DISP "TAB KEY TO MOVE TO THE NEXT FIELD. PRESS ENTER WHEN
 COMPLETE"
160 Define_fields: !
170      CURSOR (11,5),IF(15),(29,5),IF(15),(47,5),IF(2)
180      CURSOR (14,8),IF(19),(14,11),IF(14),(31,11),IF(5)
190      CURSOR (14,14),IF(5),(23,14),IF(3),(28,14),IF(8)
200      CURSOR (19,19),OF(20)
210      CURSOR (1,1),PL(24)
220 Input_data: !
230      CURSOR (5,5)  ! Set cursor to first field.
240      INPUT         ! Allow operator to fill fields.
250      CURSOR (5,5)
260      ENTER Name$[1,15],Name$[16,30],Name$[31,32]
270      ENTER Address$[1,19]
280      ENTER Address$[20,34],Address$[35,39]
290      ENTER Zip$,Phone$[1,3],Phone$[4,11]
300      CURSOR (1,1),UPL(24)
310 Assign_account: !
320      READ #1;Account_no
330      CURSOR (19,19)! Set cursor at output fields.
340      DISP Account_no+1
350      PRINT #1;Account_no+1
360      CURSOR (1,1),UPL(24)
370 Open_file: ! Open new file on customer.
```

Notice that the cursor must be positioned at the beginning of each display input or output operation. See line 230, 250, and 330. Be sure to unprotect the display (line 300) after entering data to allow use of the display lines by other operations.

The form generated by this program is shown here:

```
                NEW CUSTOMER FORM


     NAME ____________________ , _______________ , _____
          last                   first             middle

     ADDRESS _____________________________
              street

             ____________________, ____
              city                 state

             _____   (___)________
              zip      telephone


    ACCOUNT NO.
________________________________________________________________________________
ENTER INFORMATION INTO EACH BLANK FIELD. USE THE
TAB KEY TO MOVE TO THE NEXT FIELD.  PRESS ENTER WHEN COMPLETE.
```

The PRINT Statement

The PRINT statement allows text and variables to be output on the standard printer. Syntax for this statement is as follows:

PRINT [*print list*]

The print list can contain one or more of the following:

- Variable names.
- Array identifiers.
- Numeric expressions.
- String expressions.
- User Defined Types.
- TAB, SPA, LIN, and PAGE functions.

As with DISP, each item must be separated by a comma or semicolon.

Here are some examples:

```
10   FOR I=1 TO 5
20     PRINT "I EQUALS ";I
30   NEXT I

I EQUALS 1
I EQUALS 2
I EQUALS 3
I EQUALS 4
I EQUALS 5

40   PRINT "11111";"22222";"33333";"44444"
50   PRINT "55555";"66666";"77777";"88888"

11111222223333344444
55555               66666               77777               88888
```

Notice that commas and semicolons have the same effect in PRINT as in DISP: A comma after an item causes an item to be output left-justified, in a 20-character field. A semicolon after an item suppresses additional blanks. A comma or semicolon after the last item in the list allows a future print list to be appended by suppressing the CRLF. A CRLF is automatically output when the WIDTH is exceeded.

The current numeric output form (FIXED, FLOAT, or STANDARD) determines how a number is output with DISP and PRINT.

For example:

```
60   STANDARD
70   GOSUB Print
80   FIXED 2
90   GOSUB Print
100  FLOAT 4
110  GOSUB Print
120  STANDARD
130  STOP
140 Print: PRINT 123;.4560;−78910;−1.235E47
150        RETURN
```

```
123  .456 −78910 −1.235e+47
123.00  .46 −78910.00 −1.235E+47
1.2300E+02  4.5600E−01 −7.8910E+04 −1.2350E+47
```

NOTE: To print the " character, type PRINT CHR$(34).

Below you find two examples concerning printing of User Defined Type variables.

In the example below, the Comment$ member variable from the Phone1 variable is PRINTed.

```
Phone1.Comment$="*Fancy Comment*"
```

```
PRINT Phone1.Comment$
```

In addition to accessing single variables, you can specify the whole variable at once.

The example below prints all member variables of Phone1:

```
PRINT STRUCT Phone1
```

Formatted Output

The PRINT USING and IMAGE statements provide complete control output format by referencing a list of specifiers called an image string. The image string can be listed in an IMAGE statement and then referenced by stating the IMAGE statement line id in a PRINT USING statement:

PRINT USING *line id* [;*print-using list*]

IMAGE "*format string*"

Or, the image string can be contained in a string expression which is stated in place of the line id in PRINT USING:

PRINT USING *string expression* [;*print-using list*]

The string expression must be a valid image string at the time of execution.

The image string is a list of output specifiers, each separated by a delimiter. Each specifier creates a part of the output format, such as numeric and string fields, blanks, and carriage control. Each numeric or string field specifier corresponds to an equivalent item in the print-using list, and indicates how that item is to be output. The image specifiers and delimiters to be described are summarized in the next table.

An IMAGE statement must be used when literals are to be included in an image string. For example, either of these sequences is allowed:

```
300   IMAGE 30X,"Title"
310   PRINT USING 300
350   PRINT USING "30X,5A";"Title"
```

but this sequence is not:

```
400   PRINT USING"30X","Title""
```

The print-using list can contain one or more of the following:

- Variable names.
- Array identifiers.
- Numeric expressions.
- String expressions.

The items are separated by either commas or semicolons. Unlike PRINT or DISP, the delimiter used has no effect on the printout. The output is totally controlled by the image string.

PRINT USING output is directed to the standard printer, the device specified by PRINTER IS. To ensure that formatted output will be directed to the display, use the DISP USING keyword in place of PRINT USING.

Table 19 **Summary of Image Symbols**

Image Symbol	Symbol Replication	Purpose	Comments
X	Yes	Blank	Can go anywhere
" "		Text	Can go anywhere
D	Yes	Digit	Fill = blanks
Z	Yes	Digit	Fill = zeros
*	Yes	Digit	Fill = asterisks
S		Sign	"+" or "–"
M		Sign	"·" or "–"
.		Radix	Output "."
C		Comma	Conditional number separator
R		Radix	Output ","
P		Decimal Point	Conditional number separator
A	Yes	Characters	Strings
( )	Yes	Replicate	For specifiers, not symbols
#		Carriage control	Suppress CRLF
+		Carriage control	Suppress LF
–		Carriage control	Suppress CR
K		Compact	Strings or numerics
,		Delimiter	
/	Yes	Delimiter	Output CRLF
@		Delimiter	Output FF

"·" indicates a blank space.

Delimiters

Three delimiters are used to separate field specs:

, A comma is used only to separate two specifiers.

/ A slash causes output of a CRLF. When using display output fields (CURSOR), use the / to advance to the next output field. A slash can also be used to separate two specifiers.

@ The @ sign outputs a top-of-form (FF) signal, starting a new page of output. It can also be used to separate two specifiers.

/ and @ can also be used as specifiers by themselves; that is, they can be separated from other specs by a comma. Only the / can be directly replicated, however, as explained later.

Blank Spaces

A blank space is specified with X; *n*X specifies *n* blanks. Any X spec can be embedded within any other field spec.

String Specifications

Text can be specified in two ways:

" " A literal spec is text enclosed in quotes. This spec may be embedded within any other field spec.

A The character A is used to specify a single string character. nA specifies n characters. The length of the string spec is determined by the number of As that are specified between delimiters; this corresponds to one item in the print using list.

For example:

```
10   IMAGE "*****",4X"RESTART"4X,"*****"
20   PRINT USING 10

30   Res$="RESTART"
40   IMAGE "*****"4X7A4X"*****"
50   PRINT USING 40;Res$

60   PRINT USING "5A4X7A4X5A";"*****RESTART*****"
```

Each of the sequences causes the same output:

```
*****    RESTART    *****
```

If the string item in the print using list is longer than the number of characters specified, the string is truncated. For example:

```
70   PRINT USING "4A";"RESTART"
```

```
REST
```

If the item is shorter, the rest of the field is filled with blanks.

Display Enhancements and Alternate Character Sets

As shown earlier in the manual, use of display enhancements (inverse video, etc.) and alternate character strings adds unseen characters, or control bytes, to the apparent string length. For example, this statement A$="ABCDE" actually assigns seven characters to A$, the five visible underlined characters and one control byte at each end of the string. Although these control bytes must be taken into account during string operations (dimensions, substrings, string functions, etc.), they need not be considered with IMAGE/PRINT USING operations. So the statement PRINT USING "5A"; "ABCDE" will output the entire enhanced string.

Numeric Specifications

Numeric specs can be made up of digit symbols, sign symbols, radix symbols, separator symbols and an exponent symbol. All these symbols are discussed on the following pages.

Digit Symbols

D Specifies a digit position. *n*D specifies *n* digit positions. Leading zeros are replaced with a blank space as a fill character.

Z Specifies a digit position; *n*Z specifies *n* digit positions. Leading zeros are used as a fill character.

***** Specifies a digit position; *n** specifies *n* digit positions. Leading zeros are replaced with * as a fill character.

For example:

```
80   PRINT USING "DDDDD,2X,DD";250,45
90   PRINT USING "ZZZZZ,2X,DDDDD";251,321
100  PRINT USING "*****,2X,ZZZZZ,2X,DDDDD";99,88,77
```

```
  250    45
00251      321
***99    00088     77
```

Only the symbol D is allowed to the right of any radix symbol (see page 285). Any digit symbol can be used to specify the integer portion of any number, but they cannot be mixed. (For example, if D is used they must all be D.) Thus, the following example shows an invalid image and would cause an **IMPROPER PRINT USING STATEMENT** message:

```
110   PRINT USING "DDDZZ,2X,ZDZ";123,456
```

Note that there is one exception to this rule. The exception is that the digit symbol specifying the one's place can be a Z regardless of the other symbols.

Radix Symbols

A radix symbol separates the integer part of a number from the fractional part. In the United States, this is customarily the decimal point, as in 34.7. In Europe, this is frequently the comma, as in 34,7. Only one radix symbol can appear in a numeric specifier.

.	Specifies a decimal point in that position.
R	Specifies a comma in that position.

Here are some examples:

```
120   PRINT USING "DDD.DD,2X,**Z.DDD,2X,ZZZRDD";123.4,56.789,98,7
130   IMAGE DDZ.DDD,4X,ZZZ.DD
140   PRINT USING 130;.111,22.33

123.40  *56.789  098,00  7.00
  0.111    022.33
```

Sign Symbols

Two sign symbols control the output of the sign characters + and –. Only one sign symbol can appear in each numeric spec.

S	Specifies output of a + sign if the number is positive, – if the number is negative.
M	Specifies output of a – sign if it is negative, a blank if it is positive.

If the sign symbol appears before all digit symbols in a numeric spec, it floats to the left of the leftmost significant digit.

When no sign symbol is specified, any – sign occupies a digit position.

Here is an example:

```
150   PRINT USING "MDD.DD,2X,DDSZ,DD";-34.5,-67

-34.50   -67
```

Digit Separator Symbols

Digit separators are used to break large numbers into groups of digits (generally three digits per group) for greater readability. In the United States, the comma is customarily used; in Europe, the period is commonly used.

C Specifies a comma as a separator in the specified position.

P Specifies a period as a separator in the specified position.

The digit separator symbol is output only if a digit in that item has already been output; the separator must appear between two digits. When leading zeros are generated by the Z symbol, they are considered digits and will contain any separators.

Here are some examples:

```
10   N=12345.67
20   PRINT USING ”DDDDD.DD”;N
30   PRINT USING ”DDCDDD.DD”;N
40   PRINT USING ”2DC3D.2D”;N
50   PRINT USING ”2D3DR2D”;N
60   PRINT USING ”ZZZCZZZ.2D”;N
70   PRINT USING ”6Z.2D”;N
```

```
12345.67
12,345.67
12,345.67
12345,67
012,345.67
012345.67
```

Floating Specifiers

The sign specs, S and M, or text in quotes that precede all digit specifiers in a numeric spec will “float” past blanks to the leftmost digit of the number or to the radix indicator. Here are some examples:

```
200   IMAGE ”(”DDD.DD”)”
210   PRINT USING 200;1.11,22.22
```

```
(1,11) (22.22)
```

```
10   IMAGE ”$”DCDDDCDDD.DD
20   FOR I=1 TO 6
30     READ Amt
40     PRINT USING 10;Amt
50   NEXT I
60   DATA .12,12.34,1234.56,123456.78,1234567.89,12345678.90
```

```
         $.12
       $12.34
    $1,234.56
  $123,456.78
$1,234,567.89
$$$$$$$$$$$$$
```

The field of dollar signs indicates that the last item in the print-using list overflowed the image string.

Sign symbols and text that are embedded between digit symbols do not float. Here are some examples of floating and non-floating specifiers ("·" indicates a blank space):

Table 20 Summary of Image Symbols

IMAGE	Output -17	Output +17
M4D	··-17	···17
M4Z	-0017	·0017
M4*	-**17	·**17
S4D	··-17	··¬+17
'T'4D	·T-17	··T17
'T'M4D	··T-17	··T·17
DMDD	·-17	··17
DDMD	·1-7	·1·7
DDDDS	··17-	·17+

X, S, M or text embedded in a numeric stops the floating field.

Symbol Replication

Many of the symbols used to make up image specifiers can be replicated (repeated) by placing an integer (from 1 through 32767) in front of the symbol. For instance, the following images all specify the same image string:

```
200   IMAGE DDDDDD.DD
210   IMAGE 2DD3D.2D
220   IMAGE 6D.2D
```

Placing an integer before a symbol works exactly like having multiple adjacent characters. The X, D, Z, $\ast$, A, and / symbols can be replicated directly. For example:

```
230   PRINT USING "5(DX)";1,2,3,4,5

1 2 3 4 5
```

In addition to symbol replication, an entire specifier or group of specifiers can be replicated by enclosing it in parentheses and placing an integer before the parentheses. For example:

```
10   IMAGE 3(K)
20   IMAGE DD.D,6(DDD.DD)
30   IMAGE D.D,2(DDD.DD),3(D.DDD)
40   IMAGE D,4(4X,DD.DD,"LABEL2",2X,DD)
50   IMAGE 4Z.D,4(2X,4*Z.D,(2X,D))
```

In this manner, both K and @ can be repeated:

```
70   IMAGE 4(K),2(@)
```

Up to four levels of nested parentheses can be used for replication.

Compacted Specifier

A single symbol, K, is used to define an entire field for either numeric or string output. If the corresponding print-using item is a string, the entire string is output. If it is a numeric, it is output in standard form. K outputs no leading or trailing blanks in numeric fields. For example:

```
80   IMAGE K,2X,K,K,K
90   PRINT USING 80;"ABC",123,"DEF",456
```

```
ABC   123DEF456
```

Carriage Control

A CRLF is normally output when the print-using list is exhausted. This can be altered by using a carriage control symbol as the first item in an image string; a comma must separate it from the next item.

#	Suppresses both the carriage return and linefeed.
+	Suppresses the linefeed.
–	Suppresses the carriage return.

For example:

```
30   IMAGE #,4(A,2X)
40   IMAGE K
50   PRINT USING 30;"A","B","C","D"
60   PRINT USING 40;"***"
```

```
A  B  C  D  ***
```

Notice that PRINT USING "+" is equivalent to PRINT LIN(0); and PRINT USING "–" is equivalent to PRINT LIN(–1).

Here is a short program which uses carriage control and symbol replication in a DISP USING image to print a table of ASCII characters and decimal values.

```
10    !     Display ASCII character set.
20    DISP ”Cr/H Cl/S”,SPA(25);”ASCII CHARACTER SET”,LIN(1)
30    FOR Char=32 TO 127
40      DISP USING 60;CHR$(Char),Char
50    NEXT Char
60    IMAGE #,AX,”(”,3D,”)”,3X
70    END
```

The program above creates the following output:

```
                         ASCII CHARACTER SET

  ( 32)    ! ( 33)    " ( 34)    # ( 35)    $ ( 36)    % ( 37)    & ( 38)
' ( 39)
( ( 40)    ) ( 41)    * ( 42)    + ( 43)    , ( 44)    - ( 45)    . ( 46)
/ ( 47)
0 ( 48)    1 ( 49)    2 ( 50)    3 ( 51)    4 ( 52)    5 ( 53)    6 ( 54)
7 ( 55)
8 ( 56)    9 ( 57)    : ( 58)    ; ( 59)    < ( 60)    = ( 61)    > ( 62)
? ( 63)
@ ( 64)    A ( 65)    B ( 66)    C ( 67)    D ( 68)    E ( 69)    F ( 70)
G ( 71)
H ( 72)    I ( 73)    J ( 74)    K ( 75)    L ( 76)    M ( 77)    N ( 78)
O ( 79)
P ( 80)    Q ( 81)    R ( 82)    S ( 83)    T ( 84)    U ( 85)    V ( 86)
W ( 87)
X ( 88)    Y ( 89)    Z ( 90)    [ ( 91)   \\ ( 92)    ] ( 93)    ^ ( 94)
_ ( 95)
  ( 96)    a ( 97)    b ( 98)    c ( 99)    d (100)    e (101)    f (102)
g (103)
h (104)    i (105)    j (106)    k (107)    l (108)    m (109)    n (110)
o (111)
p (112)    q (113)    r (114)    s (115)    t (116)    u (117)    v (118)
w (119)
x (120)    y (121)    z (122)    { (123)    | (124)    } (125)    ~ (126)
(127)
```

A DISP statement is used to output the table header, since display control characters cannot be output within a DISP USING statement.

A similar program is used to display a table of line drawing characters. In line 40, a non-ASCII character (decimal value 147) is concatenated with each character in the series to set the line drawing mode. Lines 50 through 80 insert a CRLF after each nine data sets displayed.

```
10    !     Display LINE DRAWING character set.
20    DISP ”   ”,SPA(25);”LINE DRAWING CHARACTER SET”,LIN(1)
30    FOR Char=161 TO 246
40      DISP USING 100;CHR$(147)CHR$(Char),Char
50      I=I+1
60      IF I
```

```
9 THEN 90
70      DISP
80      I=0
90    NEXT Char
100   IMAGE #,AX,"(",3D,")",2X
110   END
```

NOTE: The actual line drawing character set displayed depends on the type of workstation you are using.

```
                    LINE DRAWING CHARACTER SET

┣(161)  ┫(162)  ┳(163)  ┻(164)  ╟(165)  ╢(166)  ╤(167)  ╧(168)  ║(169)
╂(170)  ┿(171)  ─(172)  ⌊(173)  │(174)  ┼(175)  ╋(176)  ┣(177)  ┫(178)
┳(179)  ┻(180)  ├(181)  ┤(182)  ┬(183)  ┴(184)  ═(185)  ┃(186)  ━(187)
╫(188)  ⌈(189)  ╪(190)  ╬(191)  ╞(192)  ┗(193)  ╉(194)  █(195)  ▌(196)
┃(197)  └(198)  ┘(199)  ┐(200)  ](201)  ┬(202)  ╥(203)  ╦(204)  ╈(205)
╀(206)  ╡(207)  ⌋(208)  ┏(209)  ┌(210)  ┛(211)  ┐(212)  ┤(213)  ╇(214)
┓(215)  ▄(216)  ⌉(217)  ▂(218)  ╡(219)  ╌(220)  ╨(221)  -(222)  π(223)
```

Reusing the Image String

An image string is reused from the beginning if it is exhausted before the print-using list. A CRLF is not normally output until the list is exhausted. For example:

```
70    PRINT USING "DDD.DD";25.11,99,9
```

```
25.11 99.00   9.00
```

Field Overflow

If a numeric item requires more digits than the field spec provides, an overflow condition occurs. When this happens, the item which causes the overflow is replaced with a field of dollar signs. Then the rest of the print-using list is output. For example:

```
105  IMAGE 3(DD.D)
110  PRINT USING 105;111.11,2,33.3
120  PRINT USING 105,12.3,123,1234.56
130  PRINT USING 105;12.3,–1.2,–12.3
```

```
$$$$ 2.033.3
12.3$$$$$$$$
12.3–1.2$$$$
```

Remember that a minus sign not explicitly specified with S or M requires a digit position. For instance:

```
140   PRINT USING ”2(DD.D)”;12.3,–45.6
```

```
12.3$$$$
```

Spool Files

When lack of a printer prevents running a program, spool files can be used in place of a printer. SPOOL (Simultaneous Peripheral Operations On Line) files allow rapid, temporary storage of printer output. Spool operations, however, do not relieve the CPU from eventually outputting the data to the printer. Using spool files reduces program execution time, since printer output is stored directly on a disk data file. The data can then be dumped to a printer or the display at your convenience.

Creating Spool Files

The three printer-control statements (PRINTER IS, PRINT ALL IS and SYSTEM PRINTER IS) are used to create a unique data file which holds all output of that type. The general syntax is as follows:

$$\left\{\begin{array}{l}\text{PRINTER IS}\\\text{SYSTEM PRINTER IS}\\\text{PRINT ALL IS}\end{array}\right\}\textit{file specifier}\ [\text{,WIDTH }\textit{line width}]$$

The optional WIDTH parameter specifies where to insert CRLFs in the output, as described at the start of this chapter. Once created, each spool file must not be accessed by other file storage operations (COPY, RENAME, PRINT#, etc.) until you have completed spooling into the file. Then reassign the printer-control function to another device (for example, PRINTER IS 8) to close the spool file.

Here is a program sequence which creates three spool files for later use:

```
10   F$="SPOOL"
20   D$=",SCR_PAD"
30   PRINTER IS F$,WIDTH(80)
40   SYSTEM PRINTER IS F$"1",WIDTH(132)
50   PRINT ALL IS "SPOOLA"D$
60   END
```

Recording into Spool Files

Once created, each spool file is automatically accessed by successive output operations of the appropriate type—PRINTER, SYSTEM PRINTER, or PRINT ALL. Only the significant characters of each line of output are recorded. CRLFs are automatically inserted according to the WIDTH parameter.

Dumping Spool Files

An abbreviated form of the COPY statement is used to output the contents of each spool file:

COPY *file specifier*

The contents of the spool file are dumped to the currently-specified printer of the same type assigned to the spool file. The current line width of the printer is ignored during the spool file dump.

Here is an example program which generates a 65-character-wide “ripple print-out” of alphanumeric characters. The output is first spooled into a file called Ripple then dumped to the printer.

```
10 Makeripple:  !  output ripple to spool file.
20   PRINTER IS ”Ripple”,WIDTH(65)
30   PRINT LIN(5),SPA(25);”RIPPLE PRINT”,LIN(1)
40   C1=32
50   FOR Line=1 TO 32
60     FOR Char=C1 TO C1+64
70       PRINT CHR$(Char);
80     NEXT Char
90     C1=C1+1
100  NEXT Line
110 Dumpripple:  ! copy spool file to printer.
120  PRINTER IS 0,WIDTH(200)
130  COPY ”Ripple”
140  END
```

Notice that the output can be duplicated as many times as needed by simply repeating the COPY statement.

```
                         RIPPLE PRINT
 !”#$%&’()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
!”#$%&’()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_a
”#$%&’()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_ab
#$%&’()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abc
$%&’()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcd
%&’()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcde
&’()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdef
’()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefg
()*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefgh
)*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghi
*+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghij
+,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghijk
,-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghijkl
-./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghijklm
./0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghijklmn
/0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghijklmno
0123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghijklmnop
123456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghijklmnopq
23456789:;%<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_abcdefghijklmnopqr
```

Appending to Spool Files

When an already-existing file is specified as a spool file, the computer attempts to append the spool file output onto the existing file.

Spool File Errors

Error codes 140 through 149 are reserved for spool file operations (refer to the error list in page 415). When one of these errors occurs, the spooling operation is terminated, and the display is automatically reassigned as the appropriate printing device (printer, print all or system printer).

Printer Control Functions

Many printers have a set of control functions which are accessed via unique single or multi-character control sequences. These ASCII character sequences can be sent separately to initialize particular functions such as bell, back spacing, or formfeed. The control sequences can also be embedded in text to be printed, or a combination of both methods can be used.

The CHR$ string function is most often used to send control sequences to a printer. For example, most printers respond to an ASCII BS character by back-spacing one character. The BS character has an ASCII-decimal value of 8, allowing this sequence to backspace and underline text.

```
10    PRINTER IS 0
20    PRINT "HERE'S HOW TO BACKSPACE AND UNDERLINE.";
30    FOR Char=1 TO 24
40      PRINT CHR$(8);
50    NEXT Char
60    PRINT " ____________    ____________    ____________   "
70    END
```

```
HERE'S HOW TO BACKSPACE AND UNDERLINE.
```

Use of the ASCII ESC (escape) character allows more complex functions to be controlled on the printer. For example, the sequence ESC z initiates the printer's self-test routine. This statement sends those two ASCII characters to the printer:

```
10  PRINT CHR$(27)&"z"
```

Use of escape sequences allows turning on and off various modes. This statement shows how to control the printer's underline mode:

```
100  PRINT "HERE'S ANOTHER WAY TO ";CHR$(27)&"dD";
"UNDERLINE";CHR$(27)&"dA";"."
```

```
HERE'S ANOTHER WAY TO UNDERLINE
```

Consult the printer's documentation for more information on printer control codes, escape sequences, and printer control.

10

Matrix Operations

This chapter describes the operations which can be performed on entire arrays. Some of the operations are used exclusively with matrix or vector arrays.

A matrix is a two-dimensional array. A vector is a one-dimensional array. Matrices and vectors are defined the same as other arrays.

These matrix statements and functions are described in this chapter:

MAT READ and MAT READ#	Copy values to the array.
MAT PRINT and MAT PRINT#	Write values from the array.
MAT INPUT, MAT . . . CON, MAT . . . ZER and MAT-initialize	Assign values to the array.
MAT	Arithmetic operations are performed on the entire array.
SUM	Returns the sum of all elements of the array.
ROW	Returns the number of rows in the array.
COL	Returns the number of columns in the array.
MAT . . . CSUM	Finds the sum of all columns of the matrix.
MAT . . . RSUM	Finds the sum of all rows of the matrix.

Redimensioning Arrays

Many matrix array operations allow the working size of the array to be changed prior to the actual execution of the operation. This is done when redimensioning subscripts are included in the syntax. Arrays may also be redimensioned by the REDIM statement which is described in page 73 .

There are two things to remember when redimensioning an array—the number of dimensions cannot change and the number of elements in the redimensioned array must be less than or equal to the number of elements in the original size.

Reading and Printing Arrays

MAT READ

To assign values to an array from within a program, the DATA statement is used with MAT READ. Syntax for the MAT READ statement is as follows:

$$\text{MAT READ } \textit{array 1} \quad [(\textit{redim subscripts})] \begin{bmatrix} ,\textit{array 2} \\ (\textit{redim subscripts}) \end{bmatrix}$$

The MAT READ statement specifies entire arrays. Array elements are read in order with the right-most subscript varying fastest.

For example:

```
10   OPTION BASE 1
20   INTEGER A(2,2,2)
30   DATA 1,2,3,4,5,6,7,8
40   MAT READ A
50   MAT PRINT A
60   END
```

```
 1                   2

 3                   4

 5                   6

 7                   8
```

Values are read in the following order:
A(1,1,1),A(1,1,2),A(1,2,1),A(1,2,2),A(2,1,1),A(2,1,2),A(2,2,1),A(2,2,2)

The following two statements are equivalent:

```
MAT READ A                                    READ A(*)
```

The MAT READ statement is programmable only; it *cannot* be executed from the keyboard.

MAT PRINT

The MAT PRINT statement allows the entire array to be output on the standard printer. Syntax for this statement is as follows:

$$\text{MAT PRINT array}_1 \quad \left[\begin{Bmatrix} ; \\ , \end{Bmatrix}\left[\text{array}_2\left[\begin{Bmatrix} ; \\ , \end{Bmatrix}\ldots\right]\right]\right]$$

The comma or semicolon following the array name specifies open or closed spacing between elements. A comma causes each element to be output left justified, in a 20-character field. A semicolon suppresses additional blanks. For example:

```
MAT PRINT A
 5              5              5              5
 5              5              5              5
 5              5              5              5
 5              5              5              5
MAT PRINT A,B
 5              5              5              5
 5              5              5              5
 5              5              5              5
 5              5              5              5
 2              2              2
 2              2              2
 2              2              2

MAT PRINT B;
 2  2  2
 2  2  2
 2  2  2
```

When an array has more than two dimensions, the last subscript varies fastest and defines the length of a row. For example:

```
10   OPTION BASE 1
20   INTEGER C(2,3,4)
30   FOR I=1 TO 2
40     FOR J=1 TO 3
50       FOR K=1 TO 4
60         C(I,J,K)=X
70         X=X+1
80       NEXT K
90     NEXT J
100  NEXT I
110  MAT PRINT C;
120  END

  0  1  2  3
  4  5  6  7
  8  9  10  11
  12  13  14  15
  16  17  18  19
  20  21  22  23
```

C(2,3,4) is treated as two matrices, each 3 by 4 for output or input.

File Input/Output

Entire arrays can be stored and retrieved by use of the MAT PRINT # and MAT READ # statements. Syntax for these statements are as follows:

MAT PRINT # *file number* [,*record number*]; $array_1$ [,$array_2$] [,END]

MAT READ # *file number* [,*record number*]; $array_1$ [,$array_2$] [,END]

Arrays are stored and retrieved, element-by-element, without regard to dimensionality. The last subscript varies fastest.

When the END parameter is specified in the MAT PRINT # statement, an end-of-file mark is printed at the end of the data; otherwise, an end-of-record mark is printed. For more details on PRINT #, READ #, and the other file storage operations, refer to page 195 .

Assigning Values to Arrays

MAT INPUT

The MAT INPUT statement suspends program execution, allowing values in the form of expressions to be assigned to elements of arrays from the keyboard. Syntax for this statement is as follows:

MAT INPUT *array*$_1$ [(*redim subscripts*)] [,*array*$_2$] [(*redim subscripts*)]

When MAT INPUT is executed, a question mark (?) appears in the display line. Values in the form of numeric expressions can be assigned individually or in groups (separate each value with a comma). Values are stored by pressing RETURN. The ? is redisplayed after RETURN is pressed until all values are input.

Pressing RETURN without entering a value causes execution to continue with the next element in the array. Elements not assigned values retain their previous value. For example:

```
10   OPTION BASE 1
20   INTEGER A(2,2)
30   MAT INPUT A
```

Responding to the MAT INPUT statement by typing 2, typing 4, pressing RETURN, and typing 8, assigns the following values:

```
A(1,1)=2
A(1,2)=4
A(2,1)=0    Note that the initial value, 0, is kept
A(2,2)=8
```

The following statements are equivalent:

```
MAT INPUT A                     INPUT A(*)
```

The MAT INPUT statement is programmable only; it *cannot* be executed from the keyboard.

MAT . . . CON

Use the MAT . . . CON statement to assign the constant 1 to all elements of an array. Syntax for this statement is as follows:

MAT array = CON [(redim subscripts)]

Since 1 has a logical value of “true”, the constant matrix is useful for logical initialization.

The following two sequences are equivalent:

```
10    DINTEGER A(10,10)
20    MAT A=CON
```

```
10    OPTION BASE 1
20    DINTEGER A(10,10)
30    FOR I=1 TO 10
40      FOR J=1 TO 10
50        A(I,J)=1
60      NEXT I
70    NEXT J
```

MATZER

Use the MATZER statement to assign the value 0 to all elements of an array. Syntax for this statement is as follows:

MAT array = ZER [(redim subscripts)]

Since 0 has a logical value of “false”, the zero matrix is useful for logic initialization.

The following two sequences are equivalent:

```
10   REAL A(12,12,12)
20   MAT A=ZER
```

```
10    REAL A(12,12,12)
20    FOR I=1 TO 12
30      FOR J=1 TO 12
40        FOR K=1 TO 12
50          A(I,J,K)=0
60        NEXT K
70      NEXT J
80    NEXT I
```

MAT-initialize

The MAT-initialize statement assigns the same value to every element in an array. Syntax for this statement is as follows:

MAT array = (numeric expression or string expression)

The numeric expression is evaluated once; it is converted to the numeric type of the array, if necessary. For example:

```
150    INTEGER X(4,4)
160    MAT X=(PI)
170    MAT PRINT X;
       3  3  3  3
       3  3  3  3
       3  3  3  3
       3  3  3  3
```

```
150    DIM A$(4)
160    MAT A$=("TEST")
170    MAT PRINT A$;
          TEST
          TEST
          TEST
          TEST
```

Arithmetic Operations

Copying an Array

To copy the value of each element of an array into the corresponding element of the result array, use the MAT-copy statement. Syntax for this statement is as follows:

MAT *result array* = *operand array*

The two arrays must have the same number of dimensions. The number of elements in the result array must be greater than or equal to the number of elements in the operand array. For example:

```
10   INTEGER C(4,4),D(3,3),E(4,2)
20   MAT D=(2.5)
30   MAT C=D
```

Each element of array **D** is set to 2.5, then the values of array **D** are copied into the elements of array **C**, then the working size of array **C** is redimensioned to be a 3 by 3 array.

```
40   MAT E=C
```

If the above statement was included, an error would occur because the array **E** has only 2 elements in the second dimension while the operand array, **C**, has 3 elements.

Example for string array:

```
50   DIM  A$(5),B$(5)
....

100  MAT  A$=B$
```

Scalar Operations

The scalar operations allow an arithmetic or relational operation to be performed with each element of an array using a constant scalar. The result of the operation becomes the value of the corresponding element of the result array. Either of the following syntax statements may be used to perform scalar operations:

MAT *result array* = *operand array operator* (*scalar*)

MAT *result array* = (*scalar*) *operand array operator*

For example, the following program line multiplies each element in array **C** by 4 and stores the result in the corresponding elements of array **B**:

```
30    MAT B=C*(4)
```

To further explain, take the following program line:

```
150    MAT B=C>(100)
```

If any elements of array **C** have a value greater than 100, a value of 1 is entered in the corresponding elements of array **B**. A 0 is entered if the value of the element in array **C** is less than or equal to 100.

The following operators are allowed:

```
+, -, *, /
=, < > or #, >, <, >=, <=
```

The two arrays must have the same number of dimensions. The number of elements in the result array must be greater than or equal to the number of dimensions in the operand array. The result array is redimensioned to the working size of the operand array after the operation.

Matrix Arithmetic Operations

The MAT-arithmetic statement allows an arithmetic or relational operation to be performed with corresponding elements of two arrays. The result becomes the value of the corresponding element in the result array. Syntax for this statement is as follows:

MAT *result array* = *operand array*$_1$ *operator* *operand array*$_2$

For example, the following program line multiplies corresponding elements of arrays Hours and Rate and stores them in array Pay:

```
200    MAT Pay=Hours.Rate
```

Note that a period (.) is used for multiplication, not an asterisk ($\ast$).

To further explain, take the following example:

```
30    MAT A=B>C
```

If the value in the element of array **B** is greater than the corresponding element of array **C**, a value of 1 is stored in array **A**.

The following operators are allowed:

```
+, -, /
=, < > or #, >, <, >=, <=
```

The result and operand arrays must have the same number of dimensions. The operand arrays must have the same number of elements in each dimension; the result array cannot have less.

Array Functions

System Functions

The function statement causes each element in the operand array to be evaluated by the specified function. The result becomes the corresponding element of the result array. Syntax for this statement is as follows:

MAT *result array* = *function* (*operand array*)

The function must be a single argument system function like ABS or SQR. User defined functions are not allowed.

For example, the following program line assigns the square root of each element in array **A** to the corresponding element in array **B**:

```
100   MAT B=SQR(A)
```

The SUM Function

The SUM function returns the sum of all elements in an array.

SUM *operand array*

The following two sequences are equivalent:

```
10 M A(10,10)=SUM A
20 I=SUM A
```

```
10    FOR J=1 TO 10
20      FOR K=1 TO 10
30        I=I+A(J,K)
40      NEXT K
50    NEXT J
```

The ROW Function

The ROW function returns the number of rows in the array according to its current working size. The number of rows corresponds to the subscript which is second from the right.

ROW *operand array*

The COL Function

The column function returns the number of columns in the array according to its current working size. The number of columns corresponds to the rightmost subscript.

COL *operand array*

Matrix Operations

The statements described next use either matrix arrays or vectors.

MAT . . . CSUM

The sums of all the columns of a matrix can be found by using the MATCSUM statement. Syntax for this statement is as follows:

MAT *result vector* = CSUM *operand matrix*

Each element in the result vector is the sum of the corresponding column of the operand matrix.

For example:

```
           | 2 5 7 |
Matrix A =
           | 9 8 1 |

MAT B = CSUM A

Vector B = | 11  13  8 |
```

The result vector is redimensioned.

MATRSUM

The sums of all the rows of a matrix can be found by using the MATRSUM statement. Syntax for this statement is as follows:

MAT *result vector* = RSUM *operand matrix*

Each element in the result vector is the sum of the corresponding row of the operand matrix.

For example:

```
           | 2 4 6 8 |
Matrix A =
           | 1 3 5 7 |

MAT C = RSUM A

           | 20 |
Vector C =
           | 16 |
```

The result vector is redimensioned.

11

System Clock

The clock can be used to read the date, time of day, or the seconds elapsed since 1 January 1970. It can also be used to trigger events in an Eloquence program.

NOTE: The date and time of the system clock is set by the system administrator through HP-UX commands. If you are transferring programs from an HP 260 environment and they contain the SET DATE TO or SET TIME TO statements, you will receive Error 302 (date and time has already been set).

Returning the Current System Time and Date

There are two functions provided as part of the Eloquence language which return the current time and date. They are TIME$ and DATE$.

The TIME$ Function

The TIME$ function returns the current system time. Syntax for this function is as follows:

TIME$

TIME$ (*format string*)

When TIME$ is executed without an argument, it returns the configured value defined in the Eloquence configuration file "eloqd.cfg".

In addition, you can now specify an arbitrary format. The format depends on the operating system where Eloquence is running. On HP-UX see date(1) and strftime(3) for more information on the time format.

Example:

```
TIME$("%H:%M:%S")     --> returns 11:35:45
```

The DATE$ Function

The DATE$ function returns the current system date. Syntax for this function is as follows:

DATE$

DATE$(*format string*)

When DATE$ is executed without an argument, it returns the configured value defined in the Eloquence configuration file "eloqd.cfg".

In addition, you can now specify an arbitrary format. The format depends on the operating system where Eloquence is running. On HP-UX see date(1) and strftime(3) for more information on the time format.

Example:

```
DATE$("%Y-%m-%d %H:%M")   --> returns 1997-01-14 14:37.
```

Measuring Elapsed Time

The CLOCK Function

The CLOCK function is ideal for measuring the time elapsed between two external events. It returns the number of milliseconds elapsed since a fixed date in the past (currently 1 January 1970). Syntax for this function is as follows:

CLOCK

Programmed Delays

The WAIT Statement

The WAIT statement delays program execution a specified number of milliseconds before continuing. Syntax for this statement is as follows:

WAIT [*numeric expression*]

The *numeric expression* can range from -2147483648 through 2147483647 (about 596 hours); a negative number defaults to 0. The wait can be interrupted by pressing BREAK or a user-defined special function key (softkey or SFK).

Consider the following program extract:

```
2010 WAIT 8000
```

The WAIT command with a time interval (in milliseconds) provides the specified delay. However, this approach has the following potential difficulties:

1. The resolution of the timed WAIT command is one millisecond.
2. The delay of a timed WAIT command is terminated when any software interrupt condition becomes true and the specified action command occurs. Software interrupt conditions include softkeys, BREAK key, etc.
3. Although software interrupts may be inhibited by means of the DISABLE command, this programming approach is not the friendliest to the end-user. Disabling software interrupts means that the Eloquence program is unable to detect conditions, such as input available, during the delay.

NOTE: Although the CLOCK function can be used to establish an arbitrary programmed delay, total computer system performance may suffer. Consider the following program extract:

```
2010 Start=CLOCK
2020 WHILE CLOCK-Start%<8000
2030 END WHILE
```

NOTE: This code results in an eight second programmed delay; however, the delay is a busy wait, meaning that the program demands processor execution time for the entire period of the delay. On a multiple user configuration, another process would effectively execute at half speed during the eight seconds.

The SLEEP Statement

The SLEEP statement behaves like WAIT unless executing in BACKGROUND. BACKGROUND is defined as redirecting stdout or setting -b flag on command-line. If executing in BACKGROUND the WAIT statement behaves differently:

- timed delays are ignored;
- unconditional WAIT results in reading the next key from stdin.

This behaviour is acceptable for most programs. But if a program is dealing with external devices (e.g. modems or BDE) this may make it impossible to execute such programs in BACKGROUND.

Sample code:

```
   ON KEY#8:"EXIT" GOTO E
   REQUEST #11
   PRINTER IS 11
   ON INPUT #11 GOSUB P
   WAIT
E: STOP

P: A$=AREAD$(11)
   ...
   PRINT ">"
   WAIT 500
   PRINT "<"
   RETURN
```

This program works well and may be used to get data from PORT #11 until KEY#8 is pressed. Now we are in BACKGROUND - what happens is as follows: The WAIT statement will read the next line from stdin. If there is no more line on stdin the program will be terminated at once. The only valid input is “:KEY #8” which will terminate the program. The timed DELAY in the subprogram will be ignored. The program will not work.

The solution for the above problem is the SLEEP statement. If you replace the WAIT statement with a SLEEP statement, this program will behave in back-ground the same as in foreground.

NOTE: Don’t exchange WAIT with SLEEP statement without further investigation: SLEEP will not read stdin, so it’s impossible to trigger a key “press” via stdin.

Event Scheduling

Eloquence provides a mechanism to do efficient and straightforward scheduling of future events without using programmed delays.

The ON DELAY Statement

The ON DELAY command schedules a software interrupt after a specified number of tenths of a second have elapsed.

$$\text{ON DELAY } \textit{delay spec} \begin{Bmatrix} \text{GOTO } \textit{line id} \\ \text{GOSUB } \textit{line id} \\ \text{CALL } \textit{subprogram name} \end{Bmatrix}$$

The specified interval must be an integer number of milliseconds greater than 100 and less than or equal to $2^{31}-1$. This value provides a time in milliseconds that is in the range 100 to $2^{31}-1$. The resolution of software interrupts is one millisecond. This means that although the elapsed time from the execution of the ON DELAY command to the first software interrupt may be less than one second, the operating system will provide (assuming the Eloquence program does not fall behind) subsequent software interrupts at the precise interval specified.

For example, if you wanted to schedule a software interrupt every 5.8 seconds, you would use:

```
10  ON DELAY 5800 GOTO ...
```

Note that the ON DELAY statement is active every 5.8 seconds (not for just one 5.8 second interval). You must use the OFF DELAY statement to de-activate the ON DELAY statement.

An interrupt triggered by an ON DELAY statement sets the value of CURKEY to 55.

The OFF DELAY Statement

The OFF DELAY command cancels the scheduled future software interrupts.

OFF DELAY

12

Multiple Task Programming

Eloqence provides the ability to run multiple programs concurrently from the same terminal. This ability is referred to as “multi-tasking.” Multi-tasking is controlled by five statements—REQUEST #, ATTACH #, DETACH, ATTACH, and RELEASE #. This chapter describes multi-tasking as well as how to control the shared use of peripherals, files, and databases.

NOTE: This feature is not available on the Windows NT and Linux platform.

Primary and Secondary Tasks

A primary task is the process or session that is started when you enter Eloquence. From a primary task, any Eloquence operation can be performed (for example, running and editing programs, printing output, or communicating with plotters or bar code readers).

A secondary task, like a primary task, is a process or session; however, a secondary task is started *from* a primary task using the REQUEST # and ATTACH # statements (discussed later in this chapter). In other words, a secondary task is owned by a primary task. There can be more than one secondary task per primary task. The maximum number of secondary tasks allowed depends upon what has been defined by the system administrator in the global configuration file. One primary task can have a maximum of nine secondary tasks. Each secondary task is assigned a TASKID (or USRID), just as with primary tasks. Each secondary task is referenced by its unique TASKID.

Different types of programs can run in secondary tasks; however, any job that runs in a secondary task must require very little interaction with a user. Typical examples are updating databases from transaction files, generating large records, and generating reports.

A primary task can attach to any secondary task, as long as the secondary task is not currently owned by another primary task. Ownership allows “terminal attachment” by the primary task and, at the same time, inhibits access to the secondary task by other users.

Configuration Requirements

To use the TASK feature you should enter a number larger than zero into the NTASKS line in the global configuration file. For each secondary task required, raise the number next to NTASKS by one. Make sure that the number of NUSERS is large enough, as NUSERS represents the sum of primary tasks and secondary tasks.

Multi-Tasking Statements

The five statements that control multi-tasking are summarized below. All statements can be executed from the keyboard or within a program. They provide the capability for a primary task and one or more secondary tasks to share a terminal by allowing the primary task to attach the terminal to those tasks that require attention from the user.

NOTE: To execute a program that contains the REQUEST# and ATTACH # statements, start Eloquence by typing **`eloq`** and then run the program from within Eloquence. If you start Eloquence by typing **`eloq`** ***`program name`*** when you have a program that contains these statements, the program sets up a secondary task and then the same program that started the secondary task runs in the secondary task. In other words, the same program running in the primary task runs in the secondary task.

The REQUEST # Statement

The REQUEST # statement can only be executed from a primary task. Its syntax is as follows:

REQUEST # *taskid* [,*result*]

It requests the ownership of a secondary task whose *taskid* is specified. The optional *result* parameter indicates the outcome of the request. Omitting this parameter will cause an execution error if the request is unsuccessful. This statement must be executed successfully before a subsequent ATTACH # statement can be executed for the corresponding *taskid*.

The ATTACH # Statement

The ATTACH # statement can only be executed from a primary task. Its syntax is as follows:

ATTACH # *taskid* [,*result*]

Its function is to switch the terminal from the executing primary task to the designated secondary task whose *taskid* is specified. The optional *result* parameter indicates the outcome of the statement. Omitting this parameter will cause an execution error if the statement is unsuccessful. The secondary task must have previously been REQUESTed by the primary task. Note that when you attach to a secondary task, the primary task continues to operate.

The ATTACH Statement

Its syntax is as follows:

ATTACH

Operator control (that is, the terminal) is passed to the DETACHed task executing the ATTACH statement. The ATTACH statement can be executed in primary or secondary task. However, if executed in a secondary task and the terminal is assigned to another secondary task (another ATTACH #), it will be ignored.

Example:

```
10 DETACH
20 WAIT 5000
30 DISP “Done!”
40 ATTACH
50 END
```

ATTACH to a secondary task; run the program. The terminal will switch back to primary task if executing DETACH statement. After 5 seconds the terminal will switch back to secondary task.

The DETACH Statement

The DETACH statement can only be executed from a secondary task. Its syntax is as follows:

DETACH

Operator control (that is, the terminal), is passed from the secondary task to the primary task that owns it.

The DETACH statement can also be executed by pressing CTRL D. This can be used when the user is unable to enter the DETACH statement because the keyboard is not enabled.

The RELEASE # Statement

The RELEASE # statement can only be executed from a primary task. Its syntax is as follows:

RELEASE #*taskid*

Its function is to terminate the ownership of the specified secondary task. After this statement is issued, another primary task can then REQUEST that secondary task number.

NOTE: If this statement is issued while a program is still running in the secondary task, that program will abort.

Example Program Using TASK

If you have read and understood the previous chapters, programming with TASK is simple and straight-forward. The following is an example of a simple task that generates a report in a secondary task (assuming that the Gen_report routine does the actual report generation).

```
10  DIM File_name$[6]
20  INPUT "Please enter first file name ";File_name$
30  WHILE File_name$ <> "STOPIT"
40  CALL Gen_report(File_name$)
50  INPUT "Please enter next file name ";File_name$
60  END WHILE
70  END
```

This program can be initiated by explicitly requesting a secondary task using the REQUEST and ATTACH commands, entering the program, and executing RUN. The operator can then press CTRL D to return to the primary task.

The primary task can interact with the secondary task (taskid #2 in this example) in the following manner:

```
ATTACH #2
```

Once the secondary task is obtained, the file name prompt will be displayed:

```
Please enter first file name
```

Now enter the file name desired. Once the file name is entered, the secondary task will be underway. Now enter the DETACH statement. This will cause the terminal to switch back to the primary task. A return from subroutine Gen_report will cause the secondary task to wait for another file name to be input. The operator of the terminal can now repeat the process to resume the secondary task. Note that this program does not provide a way to inform the primary task when one file is processed.

Error Codes

The error codes have different meanings for the REQUEST #, ATTACH #, and DETACH statements. Please see chapter , TASK Errors, in Appendix C for information about these error codes.

HP-UX Background Processing

Besides the multi-tasking features offered inside Eloquence, you have the possiblity to use HP-UX background processes. Within the HP-UX environment there are two options. The first option is to run your Eloquence program in HP-UX background. The second option is to run your Eloquence program in HP-UX foreground and then switch it to background. Each option is described below. All commands discussed are executed from the HP-UX prompt.

Option One

Eloquence distinguishes between two major operating modes: Interactive mode and background mode. Interactive mode is assumed when stdout is connected to a tty device (e.g. your terminal). Background mode is assumed when stdout is connected to anything else but a tty device (e.g. pipe, file). In background mode, the screen buffer is not updated to your terminal. All screen output is put only in the internal screen buffer.

You may run Eloquence from the HP-UX prompt using the following syntax:

eloqcore [-b] *program name* [*outfile*] [*infile*] [&]

If you redirect stdout to file or pipe, Eloquence will automatically run in background mode. Specifying the

-b[ackground]

switch in the command line will put Eloquence in background mode independently of the output device.

If you execute the PRINTER IS STDOUT statement in your program, you can redirect printed output into file or pipe. The BACKGROUND function returns 1 if operating in background mode.

Example:

```
! SAMPLE.PROG
IF BACKGROUND THEN PRINTER IS STDOUT
FOR I=1 TO 10
   PRINT "THIS IS LINE #";I
NEXT I
PRINTER IS 8

eloq SAMPLE | lp &
```

NOTE: If run from cron, at or batch, Eloquence will operate in background mode. If output from an Eloquence program is piped to another command (for example, "eloq TEST | lp &", terminal output is automatically suppressed. The -b is thus not necessary, but the PRINTER IS STDOUT *is*.

Option Two

Under this option, you start your Eloquence program from HP-UX foreground and then switch it to background.

The first step in this process is to define a suspend character. The suspend character is used to suspend the execution of a program so that it can be put into background. To define a suspend character, execute the following command sequence:

stty susp *susp char*

Replace *susp char* by pressing any key combination. For example, **stty susp CTRL Z**. The CTRL Z displays as ^Z.

Once a suspend character is defined, start the program in foreground by issuing the command:

eloqcore *program name*

Replace *program name* with the name of an Eloquence program. You do *not* have to specify the extension .PROG.

The next step is to switch the program to HP-UX background processing. To make this switch perform the following steps:

1 Suspend the program, using the pre-defined suspend character (for example, CTRL Z). When the suspend character is pressed, the HP-UX prompt appears.

2 Execute the command **bg [%*process number*]**. If bg is entered by itself, the program currently suspended will be put in background in the first available process. If the parameter %*process number* is added, the program currently suspended will be put in background in the specified process. Once the program is in background, it resumes execution. While the program is executing in background, you can do anything in foreground except logout. You can have several programs running in background.

To switch a program in background to foreground, use the following command:

fg [%*process number*]

The optional %*process number* is only needed if more than one program is running in background. The *process number* indicates in which background process the program is running.

Starting Eloquence from an Eloquence program.

It is also possible to start Eloquence from an Eloquence program. This Eloquence will terminate if the program is terminated.

```
1000 ! RE-STORE "FG,TEST"
1010 ! This program is intended to control FG,TEST in
1020 ! background.
1030 !
1040 ! Run eloqcore process, wait until finished
1050 !
1060        Info$="'PASS #1'"
1070     COMMAND "!INFO="&Info$&";export INFO;eloqcore BG,TEST 2>&-"
1080        COPY "BG,TEST"
1090 !
1100 ! Start eloqcore process, run asynchroneously
1110 !
1120        Info$="'PASS #2'"
1130         COMMAND "!INFO="&Info$&"; export INFO;eloqcore BG,TEST
</dev/null >/dev/null 2>&1 &"
1140        END

1000 ! RE-STORE "BG,TEST"
1010 ! This program is intended to run in the background
1020 ! controlled by a foreground eloquence (FG,TEST)
1030 !
1040 ! Purge file and recreate
1050        ON ERROR GOTO E
1060        PURGE "BG,TEST"
1070 E:     OFF ERROR
1080        CREATE "BG,TEST",0
1090        ASSIGN #1 TO "BG,TEST"
1100 !
1110 ! Put some data into file
1120        PRINT #1;"DATE = "&DATE$
1130        PRINT #1;"TIME = "&TIME$
1140        PRINT #1;"PID  = "&VAL$(PID)
1150        PRINT #1;"TASK = "&VAL$(TASKID)
1160        PRINT #1;"INFO = "&GETENV$("INFO")
1170 !
1180 ! Do some work
1190        Start=CLOCK
1200        FOR I=1 TO 10000
1210        NEXT I
1220        Seconds=(CLOCK-Start)/1000
1230        PRINT #1;"USED = "&VAL$(Seconds)
1240 !
1250 ! Done
1260        ASSIGN * TO #1
1270        END
```

Programming Considerations

When developing or modifying programs to be run on a multiple-user system, consideration should be given to resource management. Such questions should be asked as: should databases be shared or is exclusive access needed? How can a program guarantee that its output is not interrupted by another program?

Once the resource questions are answered, performance needs to be considered. For example, when is instant printer output needed and when can spooling be done? Should the database be updated or will a transaction file be kept?

Resource Management

Resources such as output devices, files, and databases are generally shared. A program requests exclusive access to a resource using one of the Eloquence statements described next. Until the program releases the resource, no other program can use it. If a program requests a resource and the resource is being used exclusively by another program, the request can be queued.

A problem could arise if one program is getting exclusive access to several resources and does not allow other programs to access it. The other programs have to wait. Therefore, a resource should only be requested when it is immediately needed and then should be released after being used.

Another problem is called “deadlock”. This occurs when one program has exclusive access to one resource, and then requests access to another. If the other resource is being used exclusively by a second program that then requests the resource locked by the first program, both programs will halt indefinitely.

Output Device Control

Output directed to a CRT is displayed on the terminal from which the program is run.

For example:

```
100  PRINTER IS 8
110  PRINT "HELLO"
120  END
```

When this sequence is run from the terminal, it prints HELLO on the terminal display.

A printer connected to a terminal (a “local printer”) has an address of 10. A printer connected to one terminal cannot be accessed by another terminal. Any terminal can access its own printer and any printers connected directly to the computer.

If several terminals use the same printer *and* that printer is *not* spooled, confusion could result if each terminal outputs a line or two. To avoid the confusion, use the REQUEST and RELEASE statements, or use the printer as a spooled device. These statements are fully described in page 249 .

File Access

If more than one program is going to access a file, some handshaking must be done before one program changes data that another program is reading. The ASSIGN statement’s class list parameter defines how the file is accessed by each program. ASSIGN is fully described in page 195 . The syntax is listed here for convenience.

ASSIGN # *file number* TO *file spec* [,*return variable*] [;*class list*]

The access keyword can be EXCLUSIVE, UPDATE or READONLY. If EXCLUSIVE access is requested, only the requesting program can use the file. If another program is already using the file, EXCLUSIVE access is not granted. In UPDATE access, several programs can access the file, but a LOCK must be performed before any writes are allowed to the file. In READONLY access, several programs can access the file but none may update it.

The LOCK and UNLOCK statements are described in page 195 . The syntax and an example using ASSIGN and LOCK are shown here for convenience.

LOCK # *file number* [,*wait variable*]

UNLOCK # *file number*

```
100  ASSIGN #1 TO “Accnt”,Return;UPDATE
110  IF Return=4 THEN GOTO Queue
120  IF Return<>0 THEN GOTO Error
130  Wait=1
140  LOCK #1,Wait
150  ! Read then modify
 .
 .
 .
200  UNLOCK #1
```

Database Locking

A database may be shared by more than one program, or individual data sets or data items may be locked for exclusive access. (The entire database can also be locked.) All the database statements are described in the *Eloquence DBMS Manual*. They are summarized here with references to multiple users.

DBOPEN

A database may be opened in one of these modes—shared access, exclusive access and read-only access. In shared access mode (mode = 1), multiple users can read from the database. They can modify the database only after locking the data set or data record to be modified. In exclusive access mode (mode = 3), only one user can read from or write to the database. In read-only access mode (mode = 8), many users can read from the database, but none may modify it. Syntax for the DBOPEN statement is as follows:

DBOPEN (*base$*,*pass$*,*mode*,*status(*)*)

LOCKING

The three statements—PREDICATE, DBLOCK and DBUNLOCK—are used together to gain exclusive access of the database or data sets and data items.

The PREDICATE statement is used when more than one data set is to be locked when data items are to be locked. It predefines the data sets and statements to be used by a DBLOCK. These sets and items are referenced by a later DBLOCK by quoting the *predicate$* parameter in the DBLOCK statement. Syntax for the PREDICATE statement is as follows:

PREDICATE *predicate$* FROM $set_1\$$ [,$item_1\$$ [,*relop$*,*value*]] [;$set_2\$$ . . . [;*setn$* . . .]]

The DBLOCK statement uses the predicate string defined in the PREDICATE statement or a data set name to lock the desired data sets and data items.

$$\text{DBLOCK } (base\$, \begin{Bmatrix} set \\ set\$ \\ predicate\$ \end{Bmatrix}, mode, status(*))$$

The DBUNLOCK statement relinquishes all locks on a database. Syntax for this statement is as follows:

$$\text{DBUNLOCK }(\mathit{base\$}, \left\{\begin{array}{l}\mathit{set}\\ \mathit{set\$}\\ \mathit{predicate\$}\end{array}\right\}, \mathit{mode}, \mathit{status}(*))$$

Under the current version of Eloquence, the parameters *set*, *set$*, and *predicate$* are ignored and the entire database is unlocked. These parameters are reserved for future use.

For example:

```
100   Base$="DBASE"
110   Pass$="USER"
120   DBOPEN (Base$,Pass$,1,Status(*))
 .
 .
 .
180   Item$="MFG"
190   Set$="COST"
200   PREDICATE Predicate$ FROM Set$,Item$,"<",100
 .
 .
 .
300   DBLOCK (Base$,Predicate$,1,Status(*))
 .
 .
 .
400   DBUNLOCK (Base$,P$,1,Status(*))
```

Performance Considerations

When deciding how to manage system resources, some thought needs to be given to performance. For example, should a program wait for a printer to become free or should it spool its output? If waiting is a frequent occurrence, should another printer be added to the computer? Should a printer be added to a terminal?

Database Performance

When multiple users are accessing a database, only the part which is to be modified should be locked. This method allows access to other parts of the database without waiting for each program to complete its processing.

Another method is to modify database information indirectly. Any changes are put in a transaction data set or file. Then, at the end of the day, one program locks the entire database and processes the transaction. This method allows more rapid access to the database during the day and ensures that the data remains constant.

Output Performance

If an output bottleneck is caused by having only one system printer, there are four options available. The first option is to use the HP-UX spooler. For this you must define the printer as type PIPE in the global configuration file eloq.config and pipe the output to the HP-UX command lp, with appropriate parameters. You can pipe the output to any HP-UX command, but lp is usually used. Of all the four options available, this option is preferable.

The second option is for each program to send its output to a spool file. Then a program can output the spool file contents to the printer at the end of the day or at any specified time. This ensures that other programs will not tie up the printer when output is needed immediately. This is accomplished by using a file name as the device specifier in the PRINTER IS statement. Spool files are described in page 249 . Syntax for the PRINTER IS statement is as follows:

PRINTER IS "*file name*"

To output the file, use the COPY statement.

```
PRINTER IS 0
COPY "file name"
```

The third and fourth options include adding a printer to the system. The printer can be added to the system for all terminals to use, or it can be added to a terminal. If one terminal is continually using a printer but the other terminals only use a printer occasionally, add the printer to the busy terminal. If all terminals are keeping the printer busy, add a new printer to the system.

Functions for Task Control

The TSTAT Function

TSTAT (*taskid*)

The TSTAT function returns the status of the specified task.

Table 21 **Summary of Image Symbols**

Return Value	Description
0	Task is in idle state (that is, not in any of the other states).
1	Task is in input state.
2	Task is in wait state.
3	Task is executing, but blocked for I/O.
4	Task is in running state.

The OWNID Function

OWNID

The OWNID function returns the USRID of the owner of the executing task. A zero is returned if the executing task is unowned.

The XOWNID Function

The XOWNID function returns the USRID of a specified task. A zero is returned if it is a primary task or not REQUESTed

Example:

```
XOWNID (15)
```

returns the owner of TASK 15.

The SHOWTASK Function

The SHOWTASK function outputs information about eloquence tasks to the SYSTEM PRINTER.

```
TASK OWNER STAT UID   NAME      PID
---- ----- ---- ----- --------- -----
  12     0   0 Y   102 mike      27624
  15    12   0 N   102 mike      27625
  16    12   0 N   102 mike      27626
```

TASK: Task number.
OWNER: Owner if secondary task or 0.
STAT: TSTAT of taskid and attached flag (Y=attached to terminal).
UID: HP-UX user id.
NAME: HP-UX user name.
PID: eloqcore process id.

The SIGNAL statements

Eloquence can use the USR1 signal to communicate with either HP-UX processes or another Eloquence process.

ON SIGNAL Branches to a specified program sequence when USR1 signal is caught

OFF SIGNAL Cancels previous ON SIGNAL

SEND SIGNAL # Send SIGUSR1 signal to specified taskid

The ON SIGNAL Statement

The ON SIGNAL statement sets up the branching condition which will occur if a USR1 signal is caught.

$$\text{ON SIGNAL} \left\{ \begin{array}{l} \text{GOTO line id} \\ \text{GOSUB line id} \\ \text{CALL subprogram name} \end{array} \right\}$$

The branch occurs immediately after the current program line is executed.

Here is an example sequence which checks for the USR1 signal and branches to a routine to output some debug information.

```
100  ON SIGNAL GOSUB Signal
110  ON HALT GOTO Stop
120  LOOP
130   I=I+1
140  END LOOP
150 Stop:!
160  DISP "I=";I
```

```
170  END
180 Signal:!
190  DISP "I=";I
200  RETURN
```

The ON SIGNAL condition is cancelled after SCRATCH, STOP, END or RUN.

To cancel any previous ON SIGNAL condition, use the OFF SIGNAL statement:

```
OFF SIGNAL
```

The SEND SIGNAL # Statement

The SEND SIGNAL # statement will send a USR1 signal to the specified taskid.

SEND SIGNAL #*taskid*

It is also possible to send USR1 signal from shell (or using the COMMAND statement) but this way you don't you have to know the process id of the destination process.

NOTE: The HP-UX protection scheme prohibits sending signals to a process of another user.

13

Asynchronous Devices

In order to address an asynchronous port with TIO statements, an appropriate entry must be made in the global, group, or user configuration file. The entry must have the following format:

PORT *port# device file*

The terminal input/output (TIO) statements provide a means to connect RS-232 asynchronous devices to your computer system. The devices may be terminals, printers, or computers.

This chapter describes the TIO statements and provides programming approaches and tips. Note that TIO error codes are listed in page 427 .

An application program can be written to send output to and receive input from asynchronous devices. TIO statements allow the application program to determine the state of each asynchronous port. The program uses PRINTER IS, PRINT ALL IS or SYSTEM PRINTER IS (coupled with the device address) to direct output to the remote device.

The port number can be between 11 and 20, but *cannot* be the same as any device address. The device file must have read and write capabilities. Here is an example PORT entry on HP-UX:

```
PORT 15 /dev/tty1p4
```

The interface parameters (such as baud rate) can be adjusted with the HP-UX stty command.

NOTE: This manual assumes that you are familiar with your computer system and that you have a good understanding of Eloquence. It is also assumes that you have read the manuals which accompany the remote printers and terminals and that you understand the devices.

TIO Statements

The ON INPUT # Statement

The ON INPUT # statement is similar to the ON KEY # statement, in that the program continues execution until a preset condition occurs. When that condition occurs, the program branches to a specified routine.

Before an ON INPUT # statement is executed, the program must have exclusive use of the device. This is done with the REQUEST statement which is described later in this chapter.

The ON INPUT # statement explicitly enables input from a terminal and informs the system of the desired action to be taken when a complete input line is received. Syntax for this statement is as follows:

ON INPUT #*port no* [*branching statement*]

The *branching statement* can be a GOTO, GOSUB, or CALL statement. Here is a description of each:

GOTO When an interrupt occurs, the program branches to the specified line and continues execution. The program cannot return to the line where the interrupt occurred. The GOTO statement does not store a return address, while the GOSUB and CALL statements do.

GOSUB When an interrupt occurs, the program branches to the subroutine. After the subroutine has been executed, the program returns to the line where the interrupt occurred.

CALL When an interrupt occurs, the program branches to the subprogram. After the subprogram has been executed, the program returns to the line where the interrupt occurred. Parameters cannot be passed.

Subprograms create a new environment during their execution. If an interrupt condition occurs during a subprogram and the action is another CALL, then the interrupt is serviced. If the action is a GOTO or GOSUB, the interrupt is not serviced until the subprogram exits with a SUBEXIT or SUBEND.

An interrupt generally occurs after execution of the current line. However, if an interrupt occurs during execution of a WAIT or INPUT statement, execution of the statement is suspended while the interrupt is serviced.

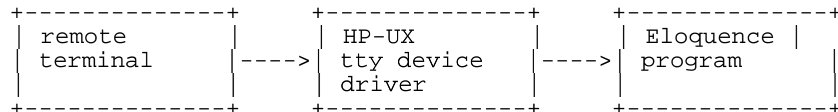
To prevent interrupts during critical portions of a program, use DISABLE statements to enclose the lines. Once the DISABLE statement is executed, no interrupts occur until the ENABLE statement is executed.

The ON INPUT # statement remains in effect until input is received or an OFF INPUT # statement is executed.

If the *branching statement* is omitted, TIO assumes that an ON INPUT # statement specifying a branch was previously executed.

For example, assume the subroutine Txy is called whenever input is available. The first ON INPUT # statement would have a branching statement of GOSUB Txy. When input is received from the terminal an interrupt occurs and program execution continues in the Txy subroutine. Before the subroutine is ended with a RETURN, the ON INPUT # statement with no branch is executed. When input is received from the terminal, an interrupt occurs and the action statement in the previous ON INPUT # statement, GOSUB Txy, is executed.

Handling of input data:



HP-UX tty device driver collects data from the remote terminal. The behaviour and transmission parameters (e.g. baud rate, echo) depend on the configuration of tty device driver (via stty). Whenever the tty device driver signals that data is available, Eloquence will read data into an internal 512 byte buffer. If ON INPUT # is active an interrupt will be generated.

It is possible to configure the HP-UX device driver characteristics from an Eloquence program. MAPPNTR$ (*port number*) will return the name of the HP-UX device file which is mapped to that port number.

(For more detailed information see references to *stty* and *termio*(7).)

```
Port = 15
REQUEST Port
COMMAND "!stty 2400 icanon icrnl <"&MAPPNTR$(Port)
...
RELEASE Port
```

The AREAD$ Function

The string function AREAD$ transfers data from the input buffer to a string variable. Syntax is as follows:

*Variable$* = AREAD$(*port number*)

If the AREAD$ function is attempted when no terminal input line is present in the input buffer, an empty string is returned. The carriage return character is not transferred to the string variable.

When the program has completed processing the input, the program explicitly re-enables further terminal input by means of the ON INPUT # statement.

Examples:

```
             REQUEST 12,Status
             IF Status=1 THEN Queue
             ON INPUT #12,3 GOSUB In_12
             .
             .
        In_12:  Next_line$=AREAD$(12)
             .
             .
             RELEASE 12
             RETURN
        Queue:  !
             REQUEST 12
             ON INPUT #12 GOSUB Input
             .
             .
        Input:  Next_line$=AREAD$(12)
             IF Next_line$="EXIT" THEN GOTO Stop_now
             .
             .
530          ON INPUT #12
540          RETURN
550   Stop_now:  RELEASE 12
560          RETURN
```

The OFF INPUT # Statement

This cancels a previous ON INPUT # statement. The syntax is as follows:

OFF INPUT #*port number*

Data remains in the buffer and can be read with AREAD$. No further interrupts are created if data arrives from that port.

Examples:

```
OFF INPUT #12
.
.
.
Device=12
.
.
OFF INPUT #Device
```

The AOVFL function

AOVFL (*port number*)

returns 1 if the TIO buffer has overflowed, otherwise 0. AREAD$ resets the buffer flag.

Eloquence Statements Used With TIO

Four Eloquence statements and one function are used with TIO—REQUEST, RELEASE, ENABLE, DISABLE, and CURKEY. The REQUEST and RELEASE statements are described fully in page 249 , while ENABLE, DISABLE, and CURKEY are in page 151 . They are briefly described here for your convenience.

The REQUEST and RELEASE Statements

Before an ON INPUT # statement is executed, the program must have exclusive access to the required device.

Successful execution of a REQUEST statement addressing a terminal causes TIO to implicitly execute the OFF INPUT # statement. Syntax of the REQUEST statement is as follows:

REQUEST *port number* [,*return variable*]

The REQUEST statement first checks to see if the requested device is a printer or a port. Printers and ports are defined in the user, group, and global configuration files using the PRINTER and PORT statements.

If the *return variable* is omitted and the device is already reserved by another program, **`ERROR 131`** results. If the requested device is not defined in the user, group, or global configuration file, **`ERROR 132`** results. If the user has no access rights or if there are currently more than 4 ports in use, **`ERROR 313 - Can't access port`** results.

If the requested device is a printer, it then checks if it is spooled or not. Spooled printers are indicated by “PIPE” in the PRINTER statement in a configuration file. The word “FILE” in a PRINTER statement indicates a non-spooled printer. If the requested printer is not spooled, the corresponding HP-UX device file is locked, thereby reserving the device for your use. This is not necessary for a spooled printer.

If the requested device is a port, the corresponding HP-UX device file is locked, reserving it for your use. The user must have read and write permission to the port, e.g. **`crw-rw-rw- 1 root bin 58 0x000005 Aug 29 14:36 /dev/tty0p5`**

The value returned to the *return variable*, if present, is based upon the following criteria:

- 0 returned if request a spooled printer.

- 1 returned if request a port or non-spooled printer that is already reserved.
- 0 returned if request a port or non-spooled printer that is available.

Reserved access to a port or non-spooled printer is relinquished by the RELEASE statement. Note that it is not necessary to RELEASE a spooled printer. This is because a spooled printer is not a directly accessed device. Syntax for the RELEASE statement is as follows:

RELEASE *port number*

The DISABLE and ENABLE Statements

The DISABLE statement inhibits all interrupts (including ON KEY # interrupts); interrupts are still recorded. When the ENABLE statement is given, interrupts are serviced in sequence. Syntax for these statements is as follows:

DISABLE

ENABLE

The CURKEY Function

CURKEY is a function which returns a number indicating the source of an ON condition interrupt. Syntax for this function is as follows:

CURKEY *numeric variable*

The values CURKEY returns are shown in the following table:

Table 22 **Summary of Image Symbols**

Value	Condition
0	No interrupts have occurred
1–24	Softkeys 1 through 24
25–27	Port 11
28–30	Port 12
31–33	Port 13
34–36	Port 14
37–39	Port 15
40–42	Port 16

Table 22 **Summary of Image Symbols**

Value	Condition
43–45	Port 17
46–48	Port 18
49–51	Port 19
52–54	Port 20
55	ON DELAY

Three values are allocated for each port. An ON INPUT # interrupt returns the first value (25 for port 11, 28 for port 12, and so on). The second and third values (26 and 27 for port 11, 29 and 30 for port 12, and so on) are reserved for future use.

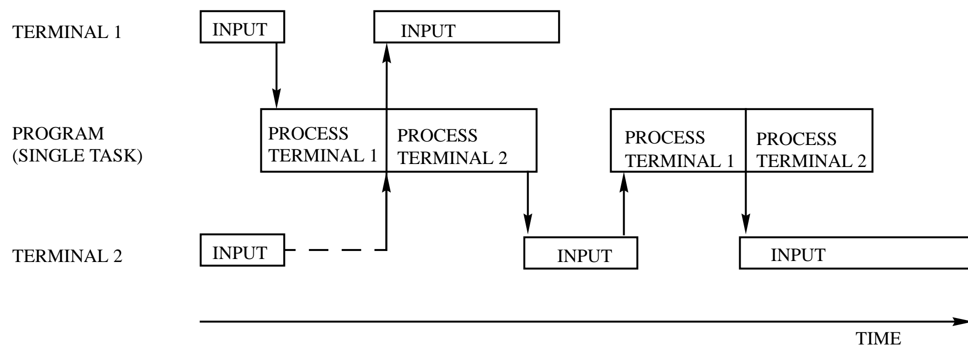
Programming with TIO

Programming Overview

TIO application programs can control remote terminals which are dedicated to the application. Since the terminal is not a remote console, the user is not confused by system messages or error codes. The application program can be tailored to the terminal user. Passwords and security features embedded in the programs can control access to sensitive information.

The ON-condition statements overlap I/O and processing. For example, instead of waiting for a terminal to respond to a prompt, the program does other processing until a carriage return is received from the terminal. If the input is not received before processing is finished, the program uses the WAIT statement to explicitly wait for the input.

Overlapping I/O and processing of other tasks is particularly useful in applications where several terminals are serviced by a single program. For example:



Programming Tips

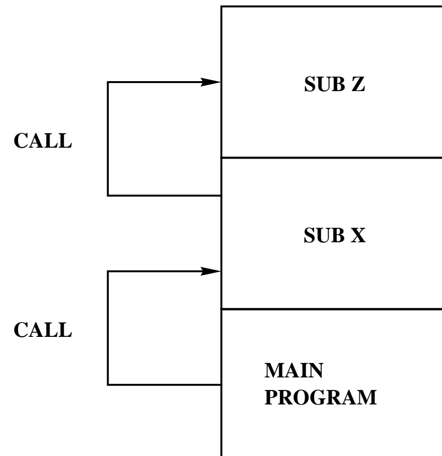
The action and consequences of the CALL branching statement as well as not specifying a branching statement should be understood.

CALL

Program execution in the current environment is suspended when the interrupt condition is satisfied. A new environment is created and remains in effect until a SUBEXIT or SUBEND is encountered or another CALL statement is executed.

The CALL statement is recognized in all successive environments including the one containing the statement.

```
10    REQUEST 11
20    CALL X
 *
 *
 *
100   SUB X
110   ON INPUT #11 CALL Y
120   CALL Z
 *
 *
 *
200   SUB Z
 *
 *
 *
```



This program is a good example of how *not* to program with ON-condition statements. When subprogram X is called, the ON INPUT # statement is executed. The interrupt is defined in subprogram X and in subprogram Z. When both X and Z are exited, however, the interrupt is no longer defined.

No Branching Statement

Not specifying a branching statement informs the system of the programmer's intent to re-establish an interrupt condition in a previous environment. This serves to turn on the input or output interrupt. It is useful when switching environments or when changing a port's input/output state.

```
10      REQUEST 12
20      ON INPUT #12 CALL  X
30      WAIT

100     SUB X
110     A$=AREAD$(12)

170     ON INPUT #12
180     SUBEXIT
```

Using OFF INPUT # clears the branching statement of the ON INPUT # statement which was executed in the same environment. Therefore, an ON INPUT # with no branching statement will subsequently be ineffective.

```
10      REQUEST 12
20      ON INPUT # 12 CALL X
30      CALL Y
```

```
100     SUB X
110     OFF INPUT #12
120     PRINTER IS 12          The OFF INPUT #12 and ON INPUT #12
130     PRINT ”...”            statements are executed in a different
                               environment than the initial
180     ON INPUT #12           ON INPUT #12 statement.
190     SUBEXIT
```

Programming Approaches

Once TIO statements and the concepts are understood, you are ready to begin programming. The most useful program approaches are introduced next.

Straight Line Approach

The following program communicates with one terminal. It demonstrates the ease of programming for one remote device. Later, the same program will be expanded for multiple terminals.

```
5       OPTION BASE 1
10      DIM A$[254] ,B$[254]
20      Port=11
30      REQUEST Port
40      PRINTER IS Port ,WIDTH(-1)
50      PRINT ”Please enter your name:”;
60      ON INPUT #Port GOTO Ini
70      WAIT
80     Ini:  A$=AREAD$(Port)                  *  Input data from the
90      PRINT ”What’s your street address”&A$&”?”* remote terminal;
100     ON INPUT #Port GOTO In2              *  equivalent to LINPUT
110     WAIT                                 *  A$ directed to the
120     In2: B$=AREAD$(Port)                 *  main console.
  .
  .
  .
```

Modular Approach

The modular approach to programming is useful when input from the terminal will determine which task is to be done.

In the following example, the program accepts a command from the terminal and the FNInterp function determines the task (X1, X2, and so on) to be done.

```
5       OPTION BASE 1
10      DIM Commd$[254]
20      Port=11
30      REQUEST Port
40      PRINTER IS Port ,WIDTH(-1)    ! This is the default width
for all TIO devices.
50      PRINT “Please enter a command”;LIN(1);”:”;
60      ON INPUT #Port GOSUB Service
70      WAIT
80      END
100    Service: Commd$=AREAD$(Port)
```

```
110    ON FNInterp(Commd$)+1 GOTO Cmd_err;Call_x1;Call_x2
120  Cmd_err: PRINT "ERROR: COMMAND NOT RECOGNIZED."
130    GOTO Print_lbl
140  Call_x1: CALL X1(Commd$)
150    GOTO Print_lbl
160  Call_x2: CALL X2(Commd$)
170    GOTO Print_lbl
 .
 .
200  Print_lbl: PRINT ";";
210    ON INPUT #Port
220    RETURN
```

Array Addressing Mode

Expanding a program from accepting input from one terminal to accepting input from several terminals can be accomplished with little difficulty if the initial program was designed properly.

In the following example, the initial program is shown in the straight line approach example:

```
10     OPTION BASE 1
20     DIM A$(5)[254], B$(5)[254]
30     DISABLE
40     FOR Port=11 TO 14
50        REQUEST Port
60        PRINTER IS Port ,WIDTH(-1)
70        PRINT "Please enter your name:";
80        ON INPUT #Port GOTO Ini
90     NEXT Port
100    ENABLE
110    WAIT
200  Ini:  DISABLE
210    Port=(CURKEY-25)/3+11    !CALCULATE PORT NUMBER
220    A$(Port-10)=AREAD$(Port)
230    PRINTER IS Port,WIDTH(-1)
240    PRINT "What's your street address "&A$"?"
250    ON INPUT #Port GOTO In2
260    ENABLE
270    WAIT
300  In2: DISABLE
310    Port=(CURKEY-25)/3+11
320    B$(Port-10)=AREAD$(Port)
```

The DISABLE and ENABLE statements are used to protect critical sections of code from GOTO interrupts.

This program can communicate with four terminals because Eloquence always knows where to go when it gets an input line from terminal. Having the system keep track of program state flow in this manner is called an Implicit State Machine. This is further discussed later in this chapter.

Executive Mode

In the following example the Service subroutine is used as an input/output traffic manager. It is referred to as an Executive Routine.

```
5       OPTION BASE 1
10      DIM A$[254], B$[254], Input$[254]
20      DIM Buff$(5)[750]
25      State=1
30         DISABLE
40         FOR Port=11 TO 14
50           PACK USING P1;Buff$(Port-10)
60           REQUEST Port
70           PRINTER IS Port
75           PRINT "Please enter your name:";
80           ON INPUT #Port GOSUB Service
90         NEXT Port
100        ENABLE
110        WAIT
120   P1:    PACKFMT State, A$,B$
130    !
200   Service:DISABLE
205        Port=(CURKEY-25)/3+11
210        Input$=AREAD$(Port)
220        UNPACK USING P1 ;Buff$(Port-10)
230        PRINTER IS Port
240        GOSUB X
250        PACK USING P1;Buff$(Port-10)
260        ON INPUT #Port
265        ENABLE
270        RETURN
280    !
300   X:    ON State GOTO S1,S2,S3
310   S1:    A$=Input$
320          PRINT "What's your street address "&A$&"?"
330          State=2
340          RETURN
350    !
360   S2:    B$=Input$
```

This program demonstrates how PACK and UNPACK can be used to swap variables into and out of a common area. This program also illustrates how to use the Explicit State Machine approach.

Structured Programming

When the application program addresses one remote device with one task, you may only need to add a few statements to the current non-TIO program to drive that device. For example, assume a report is written at the end of the working day. Instead of outputting the report to the local (default) printer, the report is now to go to a remote printer. If this is the only remote I/O function the program performs, the only modifications needed are changing the device address in the PRINTER IS statements in the appropriate places.

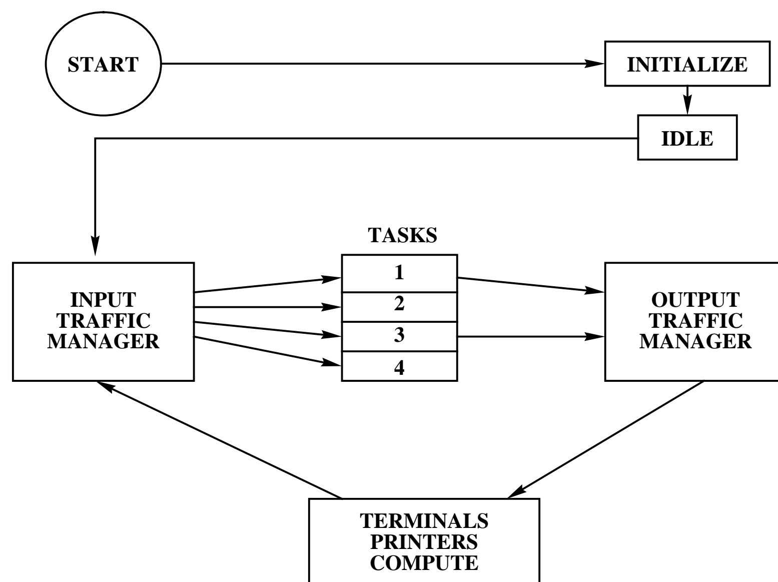
The creation of a more complex TIO application program, however, usually consists of many tasks which may be executing concurrently. In the TIO environment, each remote device may be associated with a task which controls the processing and input/output associated with the others. If interaction is desired among tasks, however, you must ensure that it happens efficiently.

The multi-tasking environment is more complex than that of sequential programming for one task. The problems demand a highly organized and structured approach. Without such an approach, you may have a program containing persistent (but unrepeatable, under debugging conditions) interference problems among the tasks.

Structured programming is concerned with improving the programming process through better program organization. Structured programming techniques, such as constructive use of subroutines and subprograms, ensure that a program is understandable, easily modified and documented, and easier to debug.

Basic Structural Flow

The following diagram can be used for many TIO applications.



In regard to the above diagram, the following remarks should be made:

Initialize The logic flow begins with the devices being initialized.

Wait Once the initialization is complete, the program waits for input from one or more of the devices.

Input Traffic Manager Input goes to the Input Traffic Manager routine which initiates a task.

Tasks Each task is processed according to its priority.

Output Traffic Manager If output is required, the Output Traffic Manager initiates and completes it to the appropriate device.

Terminals, Printers, Computer The program waits in an idle state until the next input from the devices or until one of the traffic managers begins the next task.

Example Program

The program used to demonstrate the Executive Mode of programming was designed according to the preceding structural flow diagram.

```
5      OPTION BASE 1
10     DIM A$[254], B$[254], Input$[254]
20     DIM Buff$(5)[750]
25     State=1
30        DISABLE
40        FOR Port=11 TO 14
50          PACK USING P1;Buff$(Port-10)
60          REQUEST Port
70          PRINTER IS Port
75          PRINT "Please enter your name:";
80          ON INPUT #Port GOSUB Service
90        NEXT Port
100       ENABLE
110       WAIT
120   P1:     PACKFMT State,A$,B$
130    !
200   Service:  DISABLE
205       Port=(CURKEY-25)/3+11
210       Input$=AREAD$(Port)
220       UNPACK USING P1;Buff$(Port-10)
230       PRINTER IS Port
240       GOSUB X
250       PACK USING P1;Buff$(Port-10)
260       ON INPUT #Port
266       ENABLE
270       RETURN
280    !
300   X:    ON State GOTO S1,S2,S3
310   S1:    A$=Input$
320          PRINT "What's your street address "&A$&"?"
330          State=2
340          RETURN
```

```
350   !
360  S2:    B$=Input$
 .
 .
```

Transaction Driven Applications

The design techniques described here are suitable for the creation of application programs driven by external events, such as remote terminals controlled by users. The application program and each user exchange data in an interactive fashion. The program usually supplies prompts to facilitate communication. The user might supply commands consisting of keywords and perhaps parameters to direct the program into specific operating modes and input data when requested by the current operating mode. In response to user commands (or perhaps by default), the program may display CRT forms to facilitate data entry and may generate and transmit reports either directly to the user's remote terminal or to some other output device. Finally, in response to user commands which cannot be fulfilled, the program generates messages which give the user the proper course of action.

A transaction consists of a logically complete interchange of prompts, commands, processing, input data and output reports. A transaction may be as simple as typing in a single command, in which case the transaction is just the action performed by that command. Transactions should be kept as simple as possible, otherwise the operational requirements (user training, program reliability, etc.) become extremely demanding.

Transactions may be categorized as follows:

- Security and overhead operations such as user sign on and sign off.
- Data retrieval operations accessing a database or a normal file in read-only mode. The objective of the data retrieval may be either quick "on-line" access to information or a printed report.
- Batch data-entry operations resulting in an intermediate or transaction file which is not the final end product of the application. The transaction file is later used as input in batch mode to a program which creates or updates the final end product (usually a database).
- Interactive data-entry operations in which the database is updated immediately, making the updated information available to other users as soon as the transaction is complete. If this technique is chosen, the parallel maintenance of a transaction file, as in batch data entry, should be strongly considered to allow backup and recovery from errors.

Generally, an application program offering interactive data entry transactions must ensure that multiple users and their associated tasks are protected from one another. In the worst case, the designer may have to prevent any other access to

the database while an update transaction is in progress. This includes data retrieval access in read-only mode since reporting of partially updated data may be unacceptable to the application.

State Machine Model

The State Machine Model is a conceptual framework for designing a TIO application package. It is used to keep track of where you are in the program, and how you got there. For example, in the array addressing mode example, the variable Port is used to inform the system of which terminal supplied input and where the input is to be stored.

In the executive mode example, the variable State is given a value when a routine has completed processing. When input is received from a terminal, State is used to determine the next task to perform. This method is called the Explicit State Machine Approach because the variable is explicitly assigned a value.

Controlling Your Application

The definition and priorities of the softkeys on the terminal you use to control the terminal(s) used in your application can have a strong influence on the performance of the application.

Therefore you must be aware of the relative priorities of the softkeys and the operations in your application. With this knowledge you will avoid unexpected results when running your application.

For example, if you decide that pressing a softkey on the terminal should never be allowed to interrupt your application, you should set the priority of all of the softkeys to a value lower than the priorities specified in your application program. This is done using the ON KEY # statement (refer to page 151 for the details of this statement).

14

Integrating C Functions (DLL)

DLL (Dynamic Loadable Library) is a mechanism for extending the functionality of Eloquence by calling your own functions, written in C. Using this functionality you are able to integrate specialized solutions into Eloquence.

This chapter is divided into two parts. The first part describes how to use DLL from inside Eloquence. The second part describes the technical realization and how to generate your own DLL. The example described in this chapter, and two more examples, are stored in the directory **`/opt/eloquence/share/example`**.

Using DLL in Eloquence

You can use a DLL in Eloquence like a subprogram. You load it using LOAD DLL, call it with CALL DLL and terminate it with DEL DLL.

Example:

Subprogram:	DLL:
LOAD SUB “SUBPG”	LOAD DLL Subpg, 1024
CALL Numeric(A$,A)	CALL DLL Subpg(”Numeric”, A$,A)
DEL SUB Numeric TO END	DEL DLL Subpg

The following table illustrates the differences and similarities between a subprogram and a DLL.

	Subprogram	DLL
Programming Language:	Eloquence	C
Procedures	Yes	Yes
Functions	Yes	No
COM variables	Yes	No
Module-specific variables	No	Yes
File arguments	Yes	No
Variable arguments	No	Yes
Call to Eloquence	Yes	No
Input/Output to screen	Yes	No

The DLL is started from within Eloquence as a separate process.

Communication between Eloquence and the DLL process is realized using shared memory, which is a storage area that can be accessed simultaneously by several processes.

LOAD DLL starts the DLL process and initializes the shared memory. The DLL process now waits for a signal from Eloquence.

CALL DLL copies the arguments into shared memory and signals the DLL process that the arguments have been passed. Now control is passed to the DLL.

After the DLL process has finished processing, control is passed back to Eloquence.

DEL DLL terminates the DLL process and releases the shared memory.

The LOAD DLL Statement

The LOAD DLL statement starts a DLL process and assigns it to the given name.

The syntax is as follows:

LOAD DLL Name [,filename$], size

Name the name the DLL will be referenced with. Note that this name must have an initial upper-case letter, in accordance with Eloquence syntax.

filename$ (optional). If present, specifies the file name of the DLL. If not present the file **`/opt/eloquence/dll/`***Name* will be loaded.

size specifies the size of the shared memory. The size of the shared memory depends on the number and size of the arguments.

Examples:

```
LOAD DLL Sample, 512
LOAD DLL Test, "TEST,SYSTEM", 1024
```

The first example starts the DLL **`/opt/eloquence/dll/Sample`** and creates a shared memory of 512 Bytes.

The second example starts the DLL **`Test`**, which is located in the file TEST in the directory specified by the volume name **`SYSTEM`** and creates a shared memory of 1024 Bytes.

A maximum of five DLL processes can be loaded.

Possible Error Messages:

56 File name or directory undefined or inaccessible

600 Unable to load DLL

601 Improper DLL memory size

The DEL DLL Statement

The DEL DLL statement terminates the DLL process specified by *Name*.

The syntax is as follows:

DEL DLL *Name*

Name same name as used in the LOAD DLL statement. No runtime error occurs if this DLL doesn't exist.

Example:

```
DEL DLL Sample
```

This terminates the DLL loaded with the name **Sample**.

The CALL DLL Statement

The CALL DLL statement starts the specified procedure of the DLL process.

The syntax is as follows:

CALL DLL *Name*(*Proc$* [,*arguments*])

Name same name as used in the LOAD DLL statement.

Proc$ name of the procedure

arguments list of arguments to be passed to the DLL process. The arguments are passsed as if to a subprogramm. Files cannot be passed. Maximum number of arguments is 20.

Examples:

```
CALL DLL Sample(P$,A$)
CALL DLL Test(”check”, 1,”TEST”,A$,A$(*),I,I(*))
```

The first example calls the procedure whose name is stored in **P$** in the DLL **Sample** using the argument **A$**. The second example calls the procedure **check** of the DLL **Test** with the arguments **1, "TEST", A$, A$(*), I,I(*).**

Possible Error Messages:

602 DLL not loaded

603 DLL memory overflow

604	DLL process not found
605	DLL return area destroyed
606	Number of arguments exceeds maximum

Shared Memory

The shared memory is required to exchange data between Eloquence and the DLL process.

The size of the area depends on the number and type of the arguments:

36 Bytes + 24 Bytes per argument.

Additional requirements per argument:

STRING	4 Bytes + String length rounded up to next 4 byte boundary
INTEGER	4 Bytes
DINTEGER	4 Bytes
SHORT	12 Bytes
REAL	12 Bytes

Arrays require the above memory capacity per element

Generating a DLL

This section presumes you have experience with Eloquence and C, and that you have previously worked with HP-UX tools.

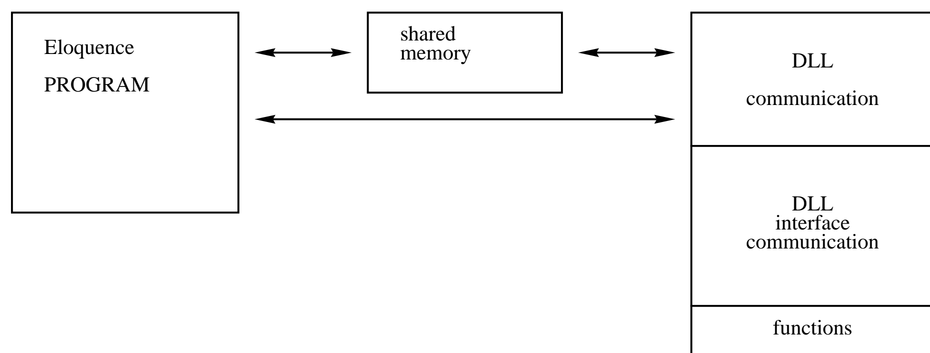
First write the C functions you want to call from your Eloquence application.

Then start the program **`/usr/eloquence/dllcc`**. This program analyzes the C functions and determines the function names and arguments which have to be used in the CALL DLL statement. This information is stored in a second C source file.

Now compile all C source files using the ANSI-C option and link them with the library **`libeloq.a`**. LOAD DLL starts the DLL process and initializes the shared memory.

The DLL program is started as a separate process from within Eloquence to prevent mutual interference.

The DLL process now waits for a signal from Eloquence. CALL DLL copies the arguments into shared memory and signals the DLL process that the arguments have been passed. Now control is passed to your C function. The DLL process converts the passed arguments into the format specified in the C functions. If the return value of the C function is non-zero, this value is used to force an Eloquence error. If no error has occured the DLL function transfers the arguments passed as "passed by reference" back to the shared memory segment and signals the Eloquence process that the DLL function has finished.



NOTE: ANSI-C functionality is not supported by the C-compiler provided as part of the standard HP-UX operating sytem. It is supported by the C-compiler sold as an optional, separate product. See the example in the directory **/usr/eloquence/example** of how to compile a DLL without the optional ANSI-C compiler. The example DLL described in this manual is generated using the ANSI-C compiler.

DLLCC

dllcc [*options*] {- | *files* }

Options:

-help Gives usage

-l Output source listing

-o *file* Output file name, default is **stdout**

Specifying input file '-' will force reading from **stdin**

Examples:

```
dllcc -o sampleif.c sample1.c sample2.c
cat test.c | dllcc - >testif.c
```

Each source file containing functions to interface with Eloquence must include the file **dllif.h**.

Each user function you want to call from Eloquence must be of type EqDLL.

The function arguments must be of the following type:

EqVoid this function has no arguments

EqInt argument is of type Integer (long)

EqReal argument is of type Real (double)

EqChar argument is of type String, terminated by a NULL-character (char *)

EqString argument is of type String (EqSring *)

... the function has a variable number of arguments

The DLLCC program supports the following syntax:

C comments (/*...*/) are recognized and ignored.

The C preprocessor statements

```
#if 0 ... #endif
```

can be used to deactivate whole segments, but they must not be nested.

The function syntax must follow the ANSI-C convention:

```
EqDLL functionname ( EqVoid )
EqDLL functionname ( Argument [, Argument] )
EqDLL functionname ( Argument [, Argument [,...]] , ...)
```

Argument types:

```
EqInt ArgumentName
```

This is an Integer argument which cannot be modified (”pass by value”). Valid argument types are INTEGER, DINTEGER, SHORT, REAL

```
EqInt *ArgumentName
```

This is an Integer argument which can be modified or an Integer array (”pass by reference”). Valid argument types are INTEGER, DINTEGER

```
EqInt ArgumentName size
```

This is an Integer array (”pass by reference”). If size is specified, this array must at least have size elements. Valid argument types are INTEGER, DINTEGER.

```
EqReal ArgumentName
```

This is a Real argument which cannot be modified (”pass by value”). Valid argument types are INTEGER, DINTEGER, SHORT, REAL.

```
EqReal *ArgumentName
```

This is a Real argument which can be modified, or a Real array (”pass by reference”). Valid argument types are SHORT, REAL.

```
EqReal ArgumentName size
```

This is a Real array (”pass by reference”). If size is specified, this array must at least have size elements. Valid argument types are SHORT, REAL

```
EqChar *ArgumentName
```

This is a pointer to a zero-terminated character array (”pass by value”). Valid argument type is STRING.

```
EqString *ArgumentName
```

This is a String argument which can be modified or a String array (”pass by reference”). Valid argument type is STRING.

```
EqString ArgumentName size
```

This is a String array (”pass by reference”). If size is specified, this array must have at least size elements. Valid argument type is STRING

```
...
```

This is a variable argument (”pass by reference”). Type depends on the Eloquence data type passed. A maximum of 20 arguments can be passed.

Example

This example shows a C program with the definition of two functions that could be called from an Eloquence application, and the output file generated by the **dllcc** program, the makefile and the corresponding Eloquence program.

Example:

```
/* num.c */

#include "dllif.h"

EqDLL int_add( EqInt int1, EqInt int2, EqInt *result )
{
   *result = int1 + int2;
   return(0);
}

EqDLL int_div( EqInt int1, EqInt int2, EqInt *result )
{
   if(int2 == 0)
      return(31);           /* division by zero */
   *result = int1 / int2;
   return(0);
}
```

The call

```
   dllcc -o num.if.c num.c
```

causes dllcc to generate the file num.if.c with the following content:

Example:

```
/*
   num.if.c
   THIS FILE IS GENERATED AUTOMATICALLY BY DLLCC
   DON'T CHANGE MANUALLY
*/

#include <dllif.h>

/*
 int_add( long int1, long int2, long *result )
 int_div( long int1, long int2, long *result )
*/

struct EqFuncList EqFuncList[] = {
 { "int_add",   0, 3 },
 { "int_div",   3, 3 },
 { 0L }
```

```
};

struct EqArgList EqArgList[] = {
 { "int1",      EqArg_IntByValue        , 0 },
 { "int2",      EqArg_IntByValue        , 0 },
 { "result",    EqArg_IntByReference    , 0 },
 { "int1",      EqArg_IntByValue        , 0 },
 { "int2",      EqArg_IntByValue        , 0 },
 { "result",    EqArg_IntByReference    , 0 },
};

int EqCall(fn, av)
int fn; void *av[];
{
 switch(fn) {
 case 0:
    return  int_add(*(EqInt  *)(av[0]),  *(EqInt  *)(av[1]),  (EqInt
*)(av[2]));
 case 1:
    return  int_div(*(EqInt  *)(av[0]),  *(EqInt  *)(av[1]),  (EqInt
*)(av[2]));
 default:
  return -1;
 }
}
```

This is the sample ‘makefile’ to build the above example.

Example:

```
# compiler flags
CFLAGS = -Aa +OV

Num: num.o num.if.o
        $(CC) -o $@ $(CFLAGS) num.if.o num.o -leloq -lmalloc

num.if.o: num.c
        /usr/eloquence/dllcc -o num.if.c num.c
        $(CC) $(CFLAGS) -c -o $@ num.if.c
        rm -f num.if.c
```

The corresponding Eloquence application is NUM.PROG.

Example:

```
1000 ! RE-STORE "NUM,EXAMPLE"
1010 !
1020     INTEGER Res
1030 !
1040     ON ERROR GOSUB Error
1050 !
1060     LOAD DLL Num,"Num,EXAMPLE",1024
1070 !
1080     CALL DLL Num("int_add",1,1,Res)
1090     PRINT Res
1100     CALL DLL Num("int_div",5,2,Res)
1110     PRINT Res
1120     CALL DLL Num("int_div",1,0,Res)
1130     PRINT Res                    ! Value unchanged due to error
1140 !
1150     DEL DLL Num
1160     END
1170 !
1180 Error:!
1190     PRINT ERRM$
1200     PRINT ERRMSG$(ERRN)
1210     RETURN
```

Programming Guidelines

DLL interface level

All source files using DLL interface level functions must include the file **dllif.h**. It defines the data types and the function prototypes.

```
#include dllif.h
```

DLL interface level functions

```
struct EqFuncInfo *EqFuncInfo( void )
```

This function returns information about the current function.

Type EqFuncInfo is defined in dllif.h.

```
struct EqFuncInfo {
   char *name;
   int arg_cnt;
};
```

name contains a pointer to the function name.

arg_cnt contains the number of arguments.

This data must not be modified.

```
struct EqArgInfo *EqArgInfo( void *arg )
```

This function returns information about a function argument. If the argument is a pointer, it can be used to find out further information on this argument. Alternatively, the order (beginning at 0) can be passed.

If the argument is invalid, NULL is returned.

Type EqArgInfo is defined in dllif.h.

```
struct EqArgInfo {
   enum EqArgType type;
   char *name;
   int elcnt;
};
```

type specifies the argument type.

name contains a pointer to the name of the argument. For a variable argument this is "...".

elcnt contains the number of elements. Unless the argument is an array, this is 1.

This data must not be modified.

int EqStrlen (EqString *s);

Returns current string length of Eloquence string

int EqMaxStrlen(EqString *s);

Returns maximum string length of Eloquence string

int EqStrcat(EqString *s1, EqString *s2);

Appends a copy of string s2 to the end of string s1. Returns zero on success or 18 on string overflow.

int EqStrcpy(EqString *s1, EqString *s2);

Copies string s2 to string s1. Returns zero on success or 18 on string overflow.

int EqStrcmp(EqString *s1, EqString *s2);

Compares its arguments and returns an integer less than, equal to, or greater than zero, depending on whether string s1 is lexicographically less than, equal to, or greater than string s2.

EqString *EqSubstr(EqString *s, int start, int len);

Extracts substring of Eloquence string.

start is the position of the first character starting at 1

len is the number of characters to be extracted.

Returns new Eloquence string descriptor, or NULL pointer if arguments are invalid.

NOTE: The return value points to a static area which will be overwritten on next call. The string value is not moved; the descriptor will point into original string.

int EqStr2str(EqString *eq, void *s);

Converts Eloquence string into zero terminated string. Returns string length without trailing zero.

NOTE: Target string must provide enough space to hold string value and terminating \0 character.

int str2EqStr(void *s, EqString *eq);

Converts zero terminated string into Eloquence string. Returns zero on success, or 18 on string overflow.

EqString *EqMkstr(int maxl);

Allocate new string using malloc(). Returns NULL on memory allocation failure.

NOTE: The return value points to a static area and will be overwritten on next call.

Setup/cleanup hooks

The functions dll_setup() and dll_cleanup() are optional. If specified by the programmer, they are called when the DLL process starts, or before it terminates.

```
void dll_setup( void )
void dll_cleanup( void )
```

The DLL process can be terminated in dll_setup() or dll_cleanup() using exit().

To terminate the process while inside a function called by Eloquence, the process should send itself a SIGTERM signal (e.g. raise(SIGTERM);). This is necessary to clean up properly.

Signals

The following signals are used internally and MUST NOT be modified:

SIGINT ignored

SIGTERM is rerouted to the DLL termination handler

SIGUSR2 is rerouted to the DLL manager

The signal handling is internally realized using the *sigvector* functionality.

Environment

When a DLL is called, all files with the exception of stderr are closed.

Stderr is linked to the terminal but not integrated into the Eloquence screen buffer.

If you reroute stderr when starting Eloquence, for example for the purpose of external tracing, all output onto stderr will also be rerouted into this file.

Debugging

If you set the global variable dll_debug, you can print out the call, the arguments and return value of the function called into stderr.

DLL debug levels

0 = no debugging

1 = function call and return value

2 = function call, argument values, return value

3 = internal

xdb

To debug a DLL process with xdb, follow the procedure below:

Make sure that the file **`.xdbrc`** is present in your HOME directory, and that it contains the following:

```
z 17 rs
```

All modules of the DLL process must be compiled and linked using the debug option (-g).

You will need 2 terminals or 2 windows. No further user can have loaded the DLL.

Now start the Eloquence program on terminal 1. After the LOAD DLL statement has been executed, you can look up the process number of the DLL process using the ‘ps’ command.

Now start xdb on terminal 2 using the following options:

```
xdb -P {DLL process number} {DLL filename}
```

Insert a breakpoint in the routine(s) you want to test and continue the DLL process with xdb ‘c’ command.

Now continue the Eloquence program execution on the first terminal. After the CALL DLL statement you are in debug mode.

DLL communication level

NOTE: Do not use the DLL communication level unless absolutely necessary. Using the DLL communication level is more trouble, more prone to errors and generally not so flexible. Using the DLL communication level is not possible in conjunction with the DLL interface level.

3 functions are available:

```
int dll_main( void )
```

dll_main() is called in a CALL DLL. The return code is interpreted as an Eloquence runtime error number.

```
void dll_init( void )
void dll_exit( void )
```

These functions are called when the DLL process is started or terminated.

DLL communication level utilities

When using DLL communication level, there are 3 functions available:

```
int u_get_argc( void )
```

Returns the number of Eloquence arguments.

```
t_DLL_arg u_get_arg(int idx)
```

Returns the description of an Eloquence argument.

```
u_unref_arg(int idx)
```

Flags an argument passed as “pass by reference” as not modified (for TRACE).

Example:

```
/*
   This is a sample DLL communication level routine
   It will force eloquence runtime error with the error
   number given as argument.
*/

#include <dll.h>

int dll_main( void )
{
   int argc;
   t_DLL_arg arg;

   /* get argument count */
   argc = u_get_argc();
   if(argc != 1)
      return(9);

   /* check for integer argument */
   arg = u_get_arg(0);
   if(arg.type != DLL_INTEGER)
      return(8);
   if(arg.elcnt != 1)
      return(8);

   /* return error number */
   return(*(int *)arg.ptr);
}
```

Error Messages

600 Unable to load DLL

5 DLL processes have already been started.

The DLL shared memory cannot be assigned.

The DLL process cannot be started.

601 Improper DLL memory size

602 DLL not loaded

603 DLL memory overflow

604 DLL process not found

There is an error in the “C” function, causing the program to abort, or the DLL process has been killed.

605 DLL return area destroyed

Usually a pointer problem in C function.

606 Number of arguments exceeds maximum.

15
Statement Flow Analyser

The SFA (Statement Flow Analyser) gives you the ability to analyse your application. You can find out which parts of your application are executed and how often, and how much time the execution took. So on the one hand you can check what parts of your application were executed in a test, on the other hand you can receive information about run time behaviour of your application.

The SFA consists of two programs: sfagen and sfarpt.

The sfagen program must be run first. It generates empty SFA files for all program files you want to analyse. Then start eloq -sfa with the required options, and the files will be filled.

The first analysis type is a *file* report. This lists what program was started, how often it was executed and how much time it used. The analysis can be done using wildcards, so that it is possible to analyse a specific application (e.g. Fibu).

To use an SFA analysis you must start Eloquence with the -sfa option and the name of the program. An analysis is not possible in the development version.

Example:

```
sfarpt -f AC*
```

All programs beginning with AC are to be analysed.

From all programs selected a histogram is created which shows what portion of the elapsed time each program took.

The second analysis type is the *segment* report. This lists which segments (parts of program) were executed and how much elpased time was used.

Example:

```
sfarpt -s AC* YSUBR.SFA
```

All programs beginning with AC, plus the program named YSUBR are to be analysed.

The third type is a *line* report. This analyses *one* program a line at a time. It shows how often this line was executed and how much time was used for it. The program distinguishes between individual segments. A segment can be a program, a SUB or an FN. Here too a histogram is created showing which program line in the segment used the most time.

Example:

```
sfarpt -l 0.5 YSUBR.SFA
```

YSUBR is to be analysed. Lines whose portion of the elapsed time is less than 0.5% are to be ignored.

Example:

```
sfarpt -g 0 YSUBR.SFA
```

List all lines which were not executed.

sfagen

The **sfagen** program generates a SFA file out of a program file. In the SFA file is stored all information about how often each line was executed and how much elpsed time was used in the execution. This SFA file can be optionally updated every time the program starts.

Syntax

sfagen [-v] *name* prog

Example

sfagen AC* YSUBR

A file with extension .SFA is created. If an SFA file already exists, it will be overwritten. This way SFA files which are already full of data can be set to zero. This would be required to obtain a new reading after a program modification.

NOTE: Every time a program is modified a new SFA file must be created, otherwise the program is excluded from the analysis. This is achieved by a comparison of the date of the last modification of .PROG file and .SFA file.

NOTE: The SFA files are *not* ASCII files.

NOTE: Waiting time is *not* included when using INPUT, WAIT, LINPUT etc. With FN and SUB calls the call line contains the accumulated elapsed time.

sfarpt

The sfarpt program analyses SFA files and produces the results on stdout.

Syntax

sfarpt [-help | -f | -s | -l *percent* | -g *percent* | -b *line* | -e *line* | -n] *file*

Options

-f	A file report is to be created. All other options to be ignored.
-s	A segment report is to be created.
-l percent	All lines with elapsed time *less* than *percent* are to be ignored (default = 0%).
-g percent	All lines with elapsed time *greater* than *percent* are to be ignored (default = 100%)
-b line	The analysis is to begin at *line number* (line report only).
-e line	The analysis is to end at *line number*. (line report only).
-n	No page header to be printed.

Example listing sfarpt DBMAP1.SFA

```
GENERIC SFARPT  (C) COPYRIGHT MARXMEIER SOFTWARE AG 2002 (A.03.00)


-------------------------------------------------------------------
------


   STATEMENT FLOW ANALYZER - LINE REPORT
```

```
   File name                : DBMAP1
   Ignored less than        :   2.00 % execution time
   Ignored greater than     : 100.00 % execution time
   Beginning line number    : 0
   End line number          : 32767

   Total execution time     : 1.54 Seconds
   Number of lines          : 48
   Executed lines           : 79.17 %

------------------------------------------------------------------
------



------------------------------------------------------------------
------
Segment               : #1, main
Times loaded          : 1
Eecution time         : 1.64 sec (106.49%)
Number lines          : 48
Executed lines        : 38

LINE                              EXECUTION
NO     HISTOGRAM                  SECONDS    % RATE COUNT  AVG
----- ------------------------------ --------- ------ ------ ------
-
 1110  **************************  0.57       37.01      1    0.57
 1230  ***********                0.25       16.23     26    0.00
 1260  *                          0.04        2.60     25    0.00
 1280  ******************************0.64       41.56     25    0.02
 1380  *                          0.04        2.60      1    0.04
```

Example listing sfarpt -l2.0 DBMAP1.SFA

```
GENERIC SFARPT  (C) COPYRIGHT MARXMEIER SOFTWARE AG 2002 (A.03.00)


------------------------------------------------------------------
------
```

```
   STATEMENT FLOW ANALYZER - LINE REPORT

   File name               :  DBMAP1
   Ignored less than       :    0.00 % execution time
   Ignored greater than    :  100.00 % execution time
   Beginning line number   :  0
   End line number         :  32767

   Total execution time    :  1.64 Seconds
   Number of lines         :  48
   Executed lines          :  79.17 %

-------------------------------------------------------------------
------



-------------------------------------------------------------------
------
Segment             : #1, main
Times loaded        : 1
Eecution time       : 1.64 sec (100.00%)
Number lines        : 48
Executed lines      : 38

LINE                                 EXECUTION
NO    HISTOGRAM                        SECONDS    % RATE  COUNT  AVG
----- ------------------------------     --------- ------ ------ --
-----
 1000                                      0.00    0.00      1    0.00
 1010                                      0.00    0.00      1    0.00
 1020                                      0.00    0.00      1    0.00
 1030                                      0.00    0.00      1    0.00
 1040                                      0.00    0.00      1    0.00
 1045                                      0.00    0.00      1    0.00
 1050                                      0.00    0.00      1    0.00
 1060                                      0.00    0.00      1    0.00
 1070                                      0.00    0.00      1    0.00
```

```
1080                                    0.01   0.61     1   0.01
1090                                    0.00   0.00     1   0.00
1100                                    0.00   0.00     1   0.00
1110 **************************          0.57  37.76     1   0.57
1120                                    0.00   0.00     1   0.00
1130                                    0.00   0.00     1   0.00
1140                                    0.00   0.00     1   0.00
1150                                    0.00   0.00     1   0.00
1160                                    0.00   0.00     1   0.00
1170                                    0.00   0.00     1   0.00
1171                                    0.01   0.61     1   0.01
1180                                    0.01   0.61     1   0.01
1190                                    0.02   1.22     1   0.02
1200                                    0.00   0.00     1   0.00
1210                                    0.00   0.00     1   0.00
1220                                    0.00   0.00     1   0.00
1230 ***********                        0.25  16.23    26   0.00
1240                                  0.02   1.22    26   0.00
1250                                   0.01   0.61    25   0.00
1260 *                                  0.04   2.44    25   0.00
1270                                    0.00   0.00    25   0.00
1280 ******************************      0.64  39.02    25   0.02
1290                                    0.00   0.00    25   0.00
1300                                    0.00   0.00    25   0.00
1310                                    0.00   0.00    25   0.00
1320                                    0.00   0.00    25   0.00
1330                                    0.02   1.22    25   0.00
1340                                    0.00   0.00    25   0.00
1350                                    0.00   0.00     0   0.00
1360                                   0.00   0.00     0   0.00
1370                                    0.00   0.00     0   0.00
1380 *                                  0.04   2.44     1   0.04
1390                                    0.00   0.00     0   0.00
1400                                    0.00   0.00     0   0.00
1410                                    0.00   0.00     0   0.00
1420                                    0.00   0.00     0   0.00
1430                                    0.00   0.00     0   0.00
```

```
 1440                                            0.00    0.00       0    0.00
 1450                                            0.00    0.00       0    0.00
```

Example listing sfarpt -b1070 -e1230 DBMAP1.SFA

```
GENERIC SFARPT  (C) COPYRIGHT MARXMEIER SOFTWARE AG 2002 (A.03.00)

------------------------------------------------------------------
------

   STATEMENT FLOW ANALYZER - LINE REPORT

   File name             : DBMAP1
   Ignored less than     :   0.00 % execution time
   Ignored greater than : 100.00 % execution time
   Beginning line number: 1070
   End line number       : 1230

   Total execution time : 0.87 Seconds
   Number of lines       : 48
   Executed lines        : 79.17 %

------------------------------------------------------------------
------


------------------------------------------------------------------
------
Segment        : #1, main
Times loaded   : 1
Execution time : 1.64 sec (188.51%)
Number lines   : 48
Executed lines : 38

LINE                                 EXECUTION
NO    HISTOGRAM                      SECONDS    % RATE COUNT  AVG
----- ---------------------------- --------- ------ ------ -------
 1070                                   0.00    0.00       1    0.00
 1080                                   0.01    1.15       1    0.01
```

```
1090                                      0.00   0.00     1    0.00
1100                                      0.00   0.00     1    0.00
1110 ******************************       0.57  65.52     1    0.57
1120                                      0.00   0.00     1    0.00
1130                                      0.00   0.00     1    0.00
1140                                      0.00   0.00     1    0.00
1150                                      0.00   0.00     1    0.00
1160                                      0.00   0.00     1    0.00
1170                                      0.00   0.00     1    0.00
1171                                      0.01   1.15     1    0.01
1180                                      0.01   1.15     1    0.01
1190 *                                    0.02   2.30     1    0.02
1200                                      0.00   0.00     1    0.00
1210                                      0.00   0.00     1    0.00
1220                                      0.00   0.00     1    0.00
1230 *************                        0.23  28.74    26    0.00
```

Example listing sfarpt -f DBMAP*

```
GERERIC SFARPT  (C) COPYRIGHT MARXMEIER SOFTWARE AG 2002 (A.03.00)

------------------------------------------------------------------

   STATEMENT FLOW ANALYZER - FILE REPORT

   Ignored less than     :   0.00 % execution time
   Ignored greater than : 100.00 % execution time

   Number of files       : 5
   Total execution time : 11.09 Seconds
   Number of lines       : 375
   Executed lines        : 66.13 %

------------------------------------------------------------------

FILE                                   EXECUTION        LINES
NAME     HISTOGRAM                     SECONDS  % RATE EXEC  % RATE
-------- ----------------------------  --------- ------ ----- ------
```

```
DBMAP1   ********                       1.64  14.79   38  79.17
DBMAP2   ***************                2.88  25.97   61  84.72
DBMAP3   ******************************  5.58  50.32   71  70.30
DBMAP4   **                             0.52   4.69   43  59.72
DBMAP5   **                             0.47   4.24   35  42.68
```

Example listing sfarpt -s DBMAP*

```
GENERIC SFARPT  (C) COPYRIGHT MARXMEIER SOFTWARE AG 2002 (A.03.00)

-----------------------------------------------------------------

   STATEMENT FLOW ANALYZER - FILE REPORT

   Ignored less than     :   0.00 % execution time
   Ignored greater than  : 100.00 % execution time

   Number of files       : 5
   Number of segments    : 5
   Total execution time  : 11.09 Seconds
   Number of lines       : 375
   Executed lines        : 66.13 %

------------------------------------------------------------------

FILE/SEGMENT                                EXECUTION        LINES
NAME                  HISTOGRAM          SECONDS   % RATE EXEC  % RATE
----------------    ----------------    -------   ----- ----  -----

DBMAP1
main                 *****                1.64     14.79   38   79.17

DBMAP2
main                 **********           2.88     25.97   61   84.72

DBMAP3
main                 ********************  5.58    50.32   71   70.30
```

```
DBMAP4
main                *                     0.52    4.69    43   59.72

DBMAP5
main                *                     0.47    4.24    35   42.68
```

A
Reference Tables

System Reset Conditions

Table 23 **(R indicates resetting to default conditions)**

	Default Setting1	SCRATCH A P C V	SCRATCH	RUN	END STOP	HALT	CONT
Variables	none	R R R 3	R	R	-	-	-
Eloquence Programs	none	R R - -	R	-	-	-	-
Program Execution	halted	R R 4 4	R	-	R	R	-
Standard Printer	display2	R - - -	-	-	-	-	-
System Printer	display2	R - - -	-	-	-	-	-
Printall Printer	display 2	R - - -	-	-	-	-	-
Subroutine Return Pointers	none	R R - -	R	R	-	-	-
Angular Units	RAD	R R - -	R	R	-	-	-
Numeric Output Format	Standard	R R - -	R	R	-	-	-
Files Table	files closed	R 5 R 5	5	5	5	-	-
DATA Pointers	none	R R - -	R	R	-	-	-
ERRL, ERRN	0,0	R R R R	R	R	-	-	-
ON Declaratives	none	R R R R	R	R	R	-	-
TRACE Operations	none	R R R R	R	-	-	-	-
Device REQUESTs	none	R R R R	R	-	R	-	-

1 Setting or value at power-up or after pressing SCRATCH ALL.

2 The device address for the display is 8.

3 Resets all variables except those declared in COM.

4 Halts program only if executed while in a subprogram.

5 Also caused by LOAD and GET.

TYP Function Return Values

Table 24

Value	Meaning
0	Unrecognized type.
1	Real-precision number.
2	Total string.
3	End-of-file (EOF) mark.
4	End-of-record (EOR) mark.
5	Integer-precision number.
6	Short-precision number.
7	Double integer precision number.
8	First part of a string.
9	Intermediate part of a string.
10	Last part of a string.
11	HP-UX text file.

ASSIGN Statement Return Variable

Table 25

Return Variable	Meaning
0	File available, assignment complete.
1	File not found (same as error 56).
4	Access error (error 91 or 93).
5	Other error.

File Types

Table 26

Type	Description
PROG	Program file, machine-coded.
DATA	Data file, ASCII-coded with or without header.
FORM	Eloquence FORMS file.

IMAGE Formatting Symbols

Table 27

Image Symbol	Symbol Replication	Purpose	Comments
X	Yes	Blank	Can go anywhere
" "		Text	Can go anywhere
D	Yes	Digit	Fill = blanks
Z	Yes	Digit	Fill = zeros
*	Yes	Digit	Fill = asterisks
S		Sign	"+" or "-"
M		Sign	"·" or "-"
.		Radix	Output "."
C		Comma	Conditional number separator
R		Radix	Output ","
P		Decimal point	Conditional number separator
A	Yes	Characters	Strings
( )	Yes	Replicate	For specifiers, not symbols
#		Carriage control	Suppress CRLF
+		Carriage control	Suppress LF
-		Carriage control	Suppress CR
K		Compact	Strings or numerics
,		Delimiter	
/	Yes	Delimiter	Output CRLF
@		Delimiter	Output FF

Storage Requirements

Table 28

Variable Type	Space Needed for Read/Write Memory	Space Needed for DATA files
Simple:		
Real	16 bytes	8 bytes
Short	16 bytes	4 bytes
Integer	8 bytes	4 bytes
Double integer	8 bytes	6 bytes
String	12 bytes + 1 byte per char.[1]	4 bytes +1 byte per char + 4 bytes per defined record.[1]
Array:		
Real	8 bytes + 8 bytes/element[2]	8 bytes x no. of elements
Short	8 bytes + 8 bytes/element[2]	4 bytes x no. of elements
Integer	8 bytes + 4 bytes/element[2]	4 bytes x no. of elements
Double integer	8 bytes + 4 bytes/element[2]	6 bytes x no. of elements
String	12 bytes + 4 bytes/element[2] + 1 byte per char.[1]	4 bytes per element + total needed for each string (see above)[1]

[1] Rounded up, if needed, to a 4 byte boundary.

[2] Plus 8 bytes per dimension.

Display Enhancement Codes/Character Set Switching Codes

Table 29

Decimal Code	Hexadecimal Code	Function
128	80	Change to No Video Enhancements.
129	81	Change to Inverse Video.
130	82	Change to Blinking.
131	83	Change to Blinking and Inverse Video.
132	84	Change to Underline.
133	85	Change to Underline and Inverse Video.
134	86	Change to Underline and Blinking.
135	87	Change to Underline, Blinking, and Inverse Video.
136	88	Change to Half Bright.
137	89	Change to Half Bright and Inverse Video.
138	8A	Change to Half Bright and Blinking.
139	8B	Change to Half Bright, Blinking, and Inverse Video.
140	8C	Change to Half Bright and Underline.
141	8D	Change to Half Bright, Underline, and Inverse Video.
142	8E	Change to Half Bright, Underline, and Blinking.
143	8F	Change to Half Bright, Underline, Blinking, and Inverse Video.
146	92	Sys. Control
147	93	Line Drawing
148	94	Roman 8

ASCII Character Codes

Table 30

Graphic	Binary Equivalent	Decimal Character Equivalent	Description
	00000000	0	NULL (null)
	00000001	1	SOH (start of heading)
	00000010	2	STX (start of text)
	00000011	3	ETX (end of text)
	00000100	4	EOT (end of transmission)
	00000101	5	ENQ (enquiry)
	00000110	6	ACK (acknowledge)
	00000111	7	BEL (bell)
	00001000	8	BS (backspace)
	00001001	9	HT (horizontal tabulation)
	00001010	10	LF (line feed)
	00001011	11	VT (vertical tabulation)
	00001100	12	FF (form feed)
	00001101	13	CR (carriage return)
	00001110	14	SO (shift out)
	00001111	15	SI (shift in)
	00010000	16	DLE (data link escape)
	00010001	17	DC1 (device control 1 or X-ON)
	00010010	18	DC2 (device control 2)

Table 30

Graphic	Binary Equivalent	Decimal Character Equivalent	Description
	00010011	19	DC3 (device control 3 or X-OFF)
	00010100	20	DC4 (device control 4)
	00010101	21	NAK (negative acknowledge)
	00010110	22	SYN (synchronous idle)
	00010111	23	ETB (end of transmission block)
	00011000	24	CAN (cancel)
	00011001	25	EM (end of medium)
	00011010	26	SUB (substitute)
	00011011	27	ESC (escape)
	00011100	28	FS (file separator)
	00011101	29	GS (group separator)
	00011110	30	RS (record separator)
	00011111	31	US (unit separator)
	00100000	32	Space
!	00100001	33	Exclamation point
	00100010	34	Quotation mark
#	00100011	35	Number sign (hash mark)
$	00100100	36	Dollar sign
%	00100101	37	Percent sign
&	00100110	38	Ampersand
’	00100111	39	Apostrophe (closing single quote)
(	00101000	40	Opening parenthesis

Table 30

Graphic	Binary Equivalent	Decimal Character Equivalent	Description
)	00101001	41	Closing parenthesis
*	00101010	42	Asterisk
+	00101011	43	Plus
,	00101100	44	Comma
-	00101101	45	Hyphen (minus)
.	00101110	46	Period (point)
/	00101111	47	Slant (solidus)
0	00110000	48	Zero
1	00110001	49	One
2	00110010	50	Two
3	00110011	51	Three
4	00110100	52	Four
5	00110101	53	Five
6	00110110	54	Six
7	00110111	55	Seven
8	00111000	56	Eight
9	00111001	57	Nine
:	00111010	58	Colon
;	00111011	59	Semicolon
<	00111100	60	Less than sign
=	00111101	61	Equal sign
>	00111110	62	Greater than sign

Table 30

Graphic	Binary Equivalent	Decimal Character Equivalent	Description
?	00111111	63	Question mark
@	01000000	64	Commercial at
A	01000001	65	Uppercase A
B	01000010	66	Uppercase B
C	01000011	67	Uppercase C
D	01000100	68	Uppercase D
E	01000101	69	Uppercase E
F	01000110	70	Uppercase F
G	01000111	71	Uppercase G
H	01001000	72	Uppercase H
I	01001001	73	Uppercase I
J	01001010	74	Uppercase J
K	01001011	75	Uppercase K
L	01001100	76	Uppercase L
M	01001101	77	Uppercase M
N	01001110	78	Uppercase N
O	01001111	79	Uppercase O
P	01010000	80	Uppercase P
Q	01010001	81	Uppercase Q
R	01010010	82	Uppercase R
S	01010011	83	Uppercase S
T	01010100	84	Uppercase T

Table 30

Graphic	Binary Equivalent	Decimal Character Equivalent	Description
U	01010101	85	Uppercase U
V	01010110	86	Uppercase V
W	01010111	87	Uppercase W
X	01011000	88	Uppercase X
Y	01011001	89	Uppercase Y
Z	01011010	90	Uppercase Z
[	01011011	91	Opening square bracket
\	01011100	92	Reverse slant
]	01011101	93	Closing square bracket
^	01011110	94	Caret (circumflex)
_	01011111	95	Underscore (low line)
‘	01100000	96	Opening single quote
a	01100001	97	Lowercase a
b	01100010	98	Lowercase b
c	01100011	99	Lowercase c
d	01100100	100	Lowercase d
e	01100101	101	Lowercase e
f	01100110	102	Lowercase f
g	01100111	103	Lowercase g
h	01101000	104	Lowercase h
i	01101001	105	Lowercase i

Table 30

Graphic	Binary Equivalent	Decimal Character Equivalent	Description
j	01101010	106	Lowercase j
k	01101011	107	Lowercase k
l	01101100	108	Lowercase l
m	01101101	109	Lowercase m
n	01101110	110	Lowercase n
o	01101111	111	Lowercase o
p	01110000	112	Lowercase p
q	01110001	113	Lowercase q
r	01110010	114	Lowercase r
s	01110011	115	Lowercase s
t	01110100	116	Lowercase t
u	01110101	117	Lowercase u
v	01110110	118	Lowercase v
w	01110111	119	Lowercase w
x	01111000	120	Lowercase x
y	01111001	121	Lowercase y
z	01111010	122	Lowercase z
{	01111011	123	Opening brace (curly bracket)
		01111100	124	Vertical line
}	01111101	125	Closing brace (curly bracket)
~	01111110	126	Tilde
	01111111	127	Delete (rubout)

B
Eloquence Syntax

Introduction

The Eloquence language consists of statements, functions, operators and commands. Operators and functions are used with variables and numbers in creating numeric and string expressions. Expressions can be included in statements and executed from the keyboard. Each statement can also be preceded by a line number and stored as a program line. Commands can only be executed from the keyboard; they are not programmable.

This appendix alphabetically lists the majority of statements, functions, and commands available with the Eloquence language. Refer to the appropriate system software manual for the syntax of additional Eloquence language enhancements. For example, for database commands, refer to the *Eloquence DBMS* manual.

Obsolete keywords and options which are only handled for HP260 compatibility are no longer included here (they still work as before though).

Syntax List

A

ABS (*numeric expression*)

ACCEPT *string variable*

ACS (*numeric expression*)

ASN (*numeric expression*)

ASSIGN { *file spec* TO# *file number* | # *file number* TO *file spec* } [,*return variable* [;*class list*]]

ASSIGN { * TO# *file number* | # *file number* TO * }

ATN (*numeric expression*)

ATTACH

ATTACH# *taskid* [,*result*]

AUTO [*beginning line number* [,*increment value*]]

B

BACKGROUND

BEEP

BINEOR (*numeric expression*, *numeric expression*)

BINIOR (*numeric expression*, *numeric expression*)

BINAND (*numeric expression*, *numeric expression*)

BINCMP (*numeric expression*)

BIT (*numeric expression*, *numeric expression*)

C

CALL *subprogram name* [(*pass parameter list*)]

CASE $\left[\begin{Bmatrix} < \\ > \end{Bmatrix} \right] \begin{Bmatrix} \textit{constant} \\ \textit{"string"} \end{Bmatrix}$ [, ...]

CASE $\begin{Bmatrix} \textit{constant} \\ \textit{"string"} \end{Bmatrix}$ TO $\begin{Bmatrix} \textit{constant} \\ \textit{"string"} \end{Bmatrix}$ [, ...]

CASE ELSE

$\begin{Bmatrix} \text{CATALOG} \\ \text{CAT} \end{Bmatrix}$ ["*catalog spec*] [,*volume spec*"] [,*file type*]

CATOPEN *filename*

CATCLOSE

CATGETMSG$ [*msg_set*] *msg_num;string_var$*

CHR$ (*numeric expression*)

CLOCK

COL (*operand array*)

COM *item* [,*item* ...]

COMMAND *string expression* [,*return variable*]

$\begin{Bmatrix} \text{CONTINUE} \\ \text{CONT} \end{Bmatrix}$ [*line id*]

COPY *source file spec* [TO *destination file spec*]

COS (*numeric expression*)

CREATE *file spec*,*number of defined records* [,*record length*]

CRTCOLS

CRTCOLS=*columns*

CRTLINES

CURKEY

CURSOR *item.* [,*item* ...]

Cursor Items

Set Cursor position

([*Xposition*] [(,*Yposition*)])	Set Curssor Position

Dislay enhancements

RE (*columns*)	Reset
IV (*columns*)	Set Inverse Video
BL (*columns*)	Set Blinking
UL (*columns*)	Set Underline
HB (*columns*)	Set Half bright display

Fields

PL (*lines*)	Protect number of lines
UPL (*lines*)	Unprotect number of lines
UPALL	Unprotect all lines
IF (*columns*)	Specify Input Field
RIF (*columns*)	Reset Input Field
OF (*columns*)	Specify Output Field
ROF (*columns*)	Reset Output Field

D

DATA *constant or text* [,*constant or text* ...]

DATE$ [*format*]

DEFAULT { ON | OFF }

DEF FN $\left\{\begin{matrix} function\ name \\ function\ name\ \$ \end{matrix}\right\}$ [(*formal parameter list*)] = *expression*

DEF FN $\left\{\begin{matrix} function\ name \\ function\ name\ \$ \end{matrix}\right\}$ [(*formal parameter list*)]

DEG

DEL *first line id* [,*second line id*]

DEL SUB *subprogram name* [TO END]

DEL FN *function name* [$] [TO END]

DETACH

DIM *item* [,*item* ...]

DINTEGER *Variable* [(dim)]

DISABLE

DISP [*display list*]

DISP USING $\left\{\begin{matrix} image\ format\ string \\ line\ id \end{matrix}\right\}$ [;*print using list*]

DROUND (*numeric expression*,*number of significant digits*)

DVP

E

EDIT $\left["prompt" \left\{\begin{matrix} , \\ ; \end{matrix}\right\} \right]$ *string variable*

ELSE

ENABLE

END

END IF

END LOOP

END SELECT

END WHILE

ENTER *variable name 1* [,*variable name 2 ...*]

ERRL

ERRM$

ERRMSG$ (*message number*)

ERRN

EXIT IF *conditional expression*

EXP (*numeric expression*)

F

FETCH [*line id*]

FIXED *number of digits*

FLOAT *number of digits*

FNEND

FOR *loop counter = initial value* TO *final value* [STEP *increment*]

FORCE ERROR *numeric expression*

FRACT (*numeric expression*)

G

GET *file spec* [,*first line id* [,*execution line id*]]

GETENV$ *string expression*

GOSUB *line id*

GOSUB *numeric expression* OF *line id list*

GOTO *line id*

GOTO *numeric expression* OF *line id list*

GRAD

H

HOP [*line id*]

I

IF *numeric expression* THEN $\begin{bmatrix} \textit{line id} \\ \textit{executable statement} \end{bmatrix}$

IMAGE *format* [, *format* ...]

Format Elements:

D	specifies a digit position. The fill character is a blank.
nD	specifies n digit positions
Z	specifies a digit position. The fill character is a zero.
nZ	specifies n digit positions.
*	specifies a digit position. The fill character is an asterisk.
n*	specifies n digit positions
X	causes a blank to be printed.
nX	causes n blanks to be printed.
A	specifies a single string character position.
nA	specifies n string characters
.	indicates placement of a decimal point radix indicator. There may be only one radix indicator per numeric specifier.
R	indicates placement of a comma radix indicator. There may only be one radix indicator per numeric specifier.
C	indicates placement of a comma in a numeric specification. It is a conditional character and is output only if there is a digit to its left.
P	indicates placement of a period in a numeric specification. It is a conditional character and is output only if there is a digit to its left.
S	indicates a sign position for a + or -. The sign floats to the left of the leftmost significant digit if S appears before all digit symbols.

M indicates a sign position. + is replaced by a blank. The sign floats to the left of the leftmost significant digit if M appears before all digit symbols.

K specifies an entire string or numeric field. A numeric is output in standard format, except that no leading or trailing blanks are output. The current value of a string is output.

INDENT *starting column*,*increment*

INPUT ["*prompt*" { ; | , }] *variable name* [,...]

INT (*numeric expression*)

L

LASTLINE

LDISP [*display list*]

LDSPEC$

LENTER *string variable*

LEN (*string expression*)

LEX (*string expression, string expression*)

LGT (*numeric expression*)

LIN (*number of line feeds*)

LINK *file spec* [,*first line id* [,*execution line id*]]

LINPUT ["*prompt*" { , | ; }] *string variable*

LOAD *file spec* [,*execution line id*]

LOAD SUB *file spec* [,*line number* [,*increment*]]
[;*starting segment* [,*last segment*]]

LOCK# *file number* [,*wait variable*]

LOG (*numeric expression*)

LOOP

LWC$ (*string expression*)

M

MAPVOL$ (*volume id*)

MAPPNTR$ (*printer*)

MASS STORAGE IS *volume spec*

MAT *array variable* = CON [(*redim subscripts*)]

MAT *result array* = *operand array*

MAT *result vector* = CSUM *operand matrix*

MAT *result array* = *function operand array*

MAT *array variable* = (*numeric expression*)

MAT INPUT *array variable* [(*redim subscripts*)] [, ...]

MAT *result array* = *operand array operator operand array*

MAT PRINT # *file number* [,*record number*] ;*array* [,...] [,END]

MAT READ *array* [(*redim subscripts*)] [, ...]

MAT READ # *file number* [,*record number*] ;
array [(*redim subscripts*)] [, ...]

MAT *result vector* = RSUM *operand matrix*

MAT *result array* = *operand array operator operand array*

MAT *result array* = (*scalar*) *operator operand array*

MAT *array variable* = ZER [(*redim subscripts*)]

MAX (*list*)

MERGE *file spec* [,*line id* [,*line id*]]

MIN (*list*)

MMSPEC$

MSI *volume spec*

N

NEXT *loop counter*

NORMAL

NUM *string expression*

NUMERIC

NO OPERATOR

O

OFF END# *file number*

OFF ERROR

OFF HALT

OFF KEY# [*key number list*]

OFF KEYBD# [*key number list*]

OFF SIGNAL

OFF INPUT# *port number*

OFF DELAY

ON END # *file number* { GOTO *line id* | GOSUB *line id* | CALL *subprogram name* }

ON ERROR { GOTO *line id* | GOSUB *line id* | CALL *subprogram name* }

ON *numeric expression* GOSUB *line id list*

ON *numeric expression* GOTO *line id list*

ON HALT
- GOTO *line id*
- GOSUB *line id*
- CALL *subprogram name*

ON INPUT# *port number* [*branching statements*]

ON KEY #*key number* [, ...] [:*label*]
- GOTO *line id*
- GOSUB *line id*
- CALL *subprogram name*

ON KEYBD #*key number* [, ...]
- GOTO *line id*
- GOSUB *line id*
- CALL *subprogram name*

ON SIGNAL
- GOTO *line id*
- GOSUB *line id*
- CALL *subprogram name*

OPTION BASE
- 0
- 1

P

PAGE

PAUSE

PI

PID

PNTR

POS (*string expression 1* ,*string expression 2*)

PPID

POPUP BOX [*xpos,ypos,*] *Image_def* [,Return]

PRINT #*file* [,*record*] [,*word*] [; [[exp*ression*] [, ...]] [END]]

PRINT [*print list*]

PRINT USING { *line id* / *image format string* } [;*print-using list*]

PRINT ALL IS { *printer number* / *file spec* }
[,WIDTH*line width*] [,TRANSPARENT]

PRINTER IS { *printer number* / *file spec* }
[,WIDTH*line width*] [,TRANSPARENT]

PROUND (*numeric expression* , *power-of-ten-position*)

PRINTER SYNC

PURGE *file spec*

Q

QUIT

QQUIT

R

RAD

RANDOMIZE [*start value*]

READ *variable name 1* [, ...]

READ #*file* [,*record*] [,*word*] [; [*variable*] [, ...]]

READ LABEL {*string variable* [ON*volume spec*] *string array name*}

REAL *Variable* [(dim)]

REC (*file number*)

REDIM *array variable* (*subscripts*) [, ...]

REFRESH [ON / OFF]

RELEASE *printer/port number*

RELEASE# *taskid*

REM *any combination of characters*

REN [*beginning line number* [,*increment value*]]

RENAME *old file spec* TO *new file name*

REPEAT

REQUEST *device address* [,*return variable*]

REQUEST# *taskid* [,result]

{RE-SAVE | RESAVE} *file spec* [,*beginning line id* [,*ending line id*]]

RE-STORE *file spec*

RESTORE [*line id*]

RETURN [*numeric expression* | *string expression*]

REVISION

REVISION$

RND

ROTATE (*numeric expression*, *numeric expression*)

ROW (*operand array*)

RPT$ (*string expression*,*number of repetitions*)

RUN [*line id* | *file spec* [,*line id*]]

S

SCAN (*string expression 1* ,*string expression 2*)

$$\text{SCRATCH}\begin{bmatrix}\text{'A}\\ \text{'C}\\ \text{'P}\\ \text{'V}\end{bmatrix}$$

SD

SELECT *conditional expression*

SEND SIGNAL# *taskid*

$$\text{SFA}\begin{Bmatrix}\text{ON}\\ \mathit{OFF}\end{Bmatrix}$$

SGN (*numeric expression*)

SHIFT (*numeric expression*, *numeric expression*)

SHOWTASK

SI

SIN (*numeric expression*)

SIZE (*file number*)

SLEEP [*number of milliseconds*]

SLEN (*file number*)

$$\text{SOFTKEYSET}\begin{Bmatrix}\text{ON}\\ \text{OFF}\\ \textit{numeric expression}\end{Bmatrix}$$

SPA (*number of spaces*)

SPACE DEPENDENT

SPACE INDEPENDENT

SQR (*numeric expression*)

STANDARD

STOP

STORE *"file name"*

SUB *subprogram name* [(*formal parameter list*)]

SUBEND

SUBEXIT

SUM (*operand array*)

SYSID$

SYSTEM PRINTER IS {*printer number* | *file spec*} [,WIDTH*line width*] [,TRANSPARENT]

T

TAB (*character position*)

TAN (*numeric expression*)

TASKID

TIME$ [*format*]

TRACE [*beginning line id* [,*ending line id*]]

TRACE ALL

TRACE PAUSE *line id* [,*numeric expression*]

TRACE VARIABLES *variable list*

TRACE WAIT *number of milliseconds*

TRIM$ (*string expression*)

TSTAT

TYP (*file number*)

TYPEAHEAD

TYPEAHEAD *mode*

TYPEAHEAD CLEAR

U

UNLOCK# *file number*

UNTIL *conditional expression*

UPC$ (*string expression*)

USRID

V

VAL (*string expression*)

VAL$ (*numeric expression*)

W

WAIT [*number of milliseconds*]

WHILE *conditional expression*

WRD (*file number*)

X

XASSIGN# *file number*TO*file name;file type* [*,assign_options ...*]

XOWNID *taskid*

XLENTER *String_var$*

XPOS

XPURGE *filename* [*;type*]

XTRACE [*trace level*]

Y

YPOS

C
Error Messages

Operator errors often result from incorrect procedures, inappropriate commands, or commands with incorrect parameters. Software errors are caused by an error in a program being executed. These may be logical errors or inappropriate commands or commands with incorrect parameters. Hardware errors result from hardware failures, absence, or malfunction. The Eloquence error message numbers and an appropriate description of each are listed below.

1 Unsupported statement.

2 Memory or program segment overflow.

3 Line not found in current program segment.

4 Improper RETURN.

5 Abnormal program termination.

6 Improperly matched FOR/NEXT.

7 Undefined function or subprogram.

8 Improper parameter matching.

9 Improper number of parameters.

10 String value required.

11 Numeric value required.

12 Attempt to re-declare a variable or label.

13 Array dimensions not specified.

14 Incorrect OPTION BASE statement.

15 Invalid bounds on array dimensions or bad string length.

16 Dimensions are improper or inconsistent.

17 Subscript out of range.

18 Subscript out of range or substring too long.

19 Improper value.

20 Integer-precision overflow.

21 Short-precision overflow.

22 Real-precision overflow.

23 Intermediate-precision overflow.

24 (TAN(N*PI/2), when N is odd.

25 Argument of ASN or ACS is >1 in absolute value.

26 0 to a negative power.

27 Negative number to non-integral power.

28 Argument of LOG or LGT is negative.

29 Argument of LOG or LGT is 0.

30 Argument of SQR is negative.

31 Division by 0, or modulo 0.

32 String does not represent a valid number, or bad input.

33 Argument of NUM, CHR$, or RPT$ is improper.

34 Reference line is not an IMAGE statement.

35 Improper image.

36 Out of data.

37 Edit string too long.

38 Syntax error in ENTER or attempt to input from protected line.

39 Function program not allowed.

40 Improper replace or delete of line.

41 First line number > second line number.

42 Attempt to replace or delete a busy line.

45 Nested keyboard-entry statements.

46 Nothing to (RE-)STORE or (RE-)SAVE.

47 Subprogram COM declaration inconsistent with main program.

48 Recursion in single-line function.

49 Line specified in ON statement not found.

50 File number out of range from 1 to 10.

51 File not currently assigned.

52 Improper volume label or mass storage specifier.

53 Improper file name.

54 Duplicate file name.

56 File name or directory undefined or inaccessible.

57 Attempt to use unknown device specifier.

58 Improper file type.

59 End of file found.

60 Physical or logical end of record found in direct access mode.

61 Defined record size too small for data item.

62 File is protected or wrong protect code specified.

63 Number of records or bytes per record exceeds 999999 or program too big.

64 Medium overflow.

65 Incorrect data type.

66 You are not authorized to store this program.

67 Parameter <= 0.

68 Invalid line number encountered during MERGE, GET or LINK.

69 Not allowed on HP-UX files.

77 Specified volume label not found.

79 Requested subprogram segment not present in LOAD SUB.

83 No access.

90 Mass storage system error.

91 Attempt to access a busy file.

92 Cannot get exclusive access to a specified file.

93	File opened in conflicting mode.
94	Specified file cannot currently be locked.
95	String not intact on file.
100	IMAGE specification expects a numeric item.
101	IMAGE specification expects a string item.
102	Numeric field specification is larger than internal buffer.
103	Item in PRINT USING list has no corresponding IMAGE specification.
120	Output field overflow
121	Improper value in CURSOR statement.
130	Parameter for REQUEST or RELEASE out of range.
131	Specified device not available.
132	Referenced device missing or wrong type.
133	Printer is down or broken pipe.
134	Printer is offline.
140	Improper spool file type.
143	Expansion of spool file would cause medium overflow.
150	Type of expression in CASE does not match type of SELECT.
151	Parameter out of range on INDENT.
152	Improper matching of structured construct.
153	No structured construct active.
155	Invalid statement specified in COMMAND.
156	More than one level of recursion not allowed in COMMAND.
170	HP-UX command fails.

Pack Errors

200	Referenced line not a PACKFMT.
202	Insufficient dim length in PACK or insufficient current length in UNPACK.
204	Conversion error.
205	UNPACK requires a source string of greater current length.

IMAGE Errors

210	Bad status array.
211	No DBASE IS statement active.
212	Specified data set not found.
213	Too many variables in list.
214	IN DATA SET already active for data set.
215	Number of elements does not match.
216	Variable type does not match with associated field in set.
217	String length in list insufficient.
218	Variable in common.
219	Line referenced is not an IN DATA SET LIST statement.
220	Improper or illegal use of maintenance word.
221	Data set not created.
225	Incompatible data base.
226	Corrupt data base - must recreate it.
227	Corrupt data base - must erase in its entirety.

PREDICATE Errors

320	Set or item specifier is out of range or is an invalid set or item name.
321	Relational operator is invalid.
322	The predicate specifier is not in a valid form.

SORT Errors

230	Improper nesting of SORT statements, including DBASE IS and IN DATA SET.
231	Cannot reactivate workfile, or file is not a workfile.
233	** No read access to specified data set.
234	Missing or improper set linkage.
235	NO WORKFILE IS # statement active.
236	Improper data item or data item not found (also QFIND).
238	Improper synthetic linkage.
239	Insufficient space in workfile.
241	Improper operation attempted on workfile.
242	Improper READ# or PRINT# on workfile.
243	Workfile contains invalid information.
244	Data base corrupt.
246	Workfile not empty (QFIND).
247	Unexpected error accessing data base (QFIND).
248	Improper QFIND relation (QFIND).
249	Improper value type or improper number of value parameter (QFIND).

Report Writer Errors

250 BEGIN REPORT does not reference a REPORT HEADER statement.

251 Report Writer is already active.

252 An END REPORT DESCRIPTION statement is missing.

253 Duplicate Report Writer section.

254 Invalid blank lines in PAGE LENGTH statement.

255 Expression in a Report Writer statement evaluates to an unacceptable value.

256 A (GRAND) TOTALS ON is improperly positioned.

257 A Report Writer operation was requested while outside the program scope.

258 Effective page length is less than three lines.

259 Illegal execution of a Report Description section statement.

260 Insufficient space for printed output within the current page.

261 Left margin specified is less than 1 or greater than current printer width.

262 Control variable in BREAK WHEN statement has greater length.

263 A DETAIL LINE statement may not occur in the Report Description section.

264 Level parameter is out of range from 0 thru 9.

265 (GRAND) TOTALS ON statement not active for the level requested.

266 Sequence parameter is out of range for (GRAND) TOTALS ON statement.

267 Illegal WITH number LINES parameter in header, trailer or detail line.

268 OLDCV($) function references a level which does not have a

	break defined.
269	OLDCV($) function does not match the data type for the control variable.
270	PRINTER IS statement may not be executed while Report Writer is active.
271	A Report Writer statement may not be used recursively.

FORMS Errors

290 Not allowed when form is active.

291 Not allowed within form image.

292 Attempt to input after last field of form.

293 Attempt to output after last field of form.

294 Not allowed unless form is active.

TIMER Errors

302 Date or time cannot be changed here.

303 ON DELAY value is incorrect.

TIO Errors

310	Port ordinal out of range from 11 thru 20.
311	Priority value out of range from 1 thru 15.
312	Invalid address in ONinterrupt statement.
313	Can’t access port, or maximim number of ports already REQUESTed.
314	Must do REQUEST before ON INPUT.

TASK Errors

400 Task functionality not available or unable to start task.

401 Specified TASKID not a task.

402 Specified TASKID not a secondary task or not available.

403 Executing task is not a primary task or currently in background.

413 Protection violation in SEND SIGNAL# statement.

The error codes have different meanings for the REQUEST #, ATTACH #, and DETACH statements. The error numbers in the table are execution errors caused by unsuccessful commands with no optional result parameter. The result column in the table is the returned status indicating the outcome of the command.

Table 31 **REQUEST # Statement**

Error number	Result	Description
none	0	Ownership granted.
400	2	Task functionality not available, or unable to start process.
401	1	Specified TASKID not a valid task.
402	2	Specified TASKID not a secondary task or
		already owned by another user.
403	3	Executing task is not a primary task.

Table 32 **ATTACH # Statement**

Error number	Result	Description
none	0	Attach initiated.
400	1	Task functionality not available.
401	1	Specified TASKID not a valid task.
402	2	Specified TASKID not owned by executing task.
403	3	Executing task not a primary task, or currently in background.

Table 33 **DETACH Statement**

Error number	Result	Description
403	3	The terminal is busy transferring files.

User Defined Types Errors

The following runtime errors are used with types:

900	Undefined base type
901	Nested types are not supported
902	Statement not allowed in type definition
903	Illegal or incomplete type definition

HP-UX Errors

Listed here are the HP-UX system errors that might occur within the Eloquence environment:

860	Old password does not match.
861	Improper number of array elements.
870	Improper POPUP BOX format.
871	Improper number of text lines in POPUP BOX.
872	Improper number of buttons in POPUP BOX.
873	Improper button preset value in POPUP BOX.
874	POPUP BOX does not fit on screen.
875	POPUP BOX positioned outside screen.
999	Program or forms file not compatible.
1000	System Files table full
1002	Request would result in deadlock
1004	Keyword not recognized.
1005	HP-UX resource table overflow.

Glossary

ASCII ASCII is the acronym for American Standard Code for Information Interchange. It is a standard way of representing characters and printing commands within a computer.

array A collection of data items of the same type having from one to six dimensions.

array elements Individual data items in an array.

array identifier A numeric array variable name followed by ($\ast$), indicating the use of the entire array variable (for example, A($\ast$)).

array variable name An array variable name consists of 1 to 15 characters, followed by a subscript. The first character must be an uppercase (capital) letter while the remaining characters must be lowercase letters, digits, or the underscore character (_). Array variable names are divided into two types—string and numeric. A string array variable name must end in a dollar sign ($).

bit bucket An imaginary device where data is dumped and cannot be retrieved. Output from a program that you do not want sent to a terminal, printer, or file can be assigned to the bit bucket.

catalog spec The optional catalog spec is a string expression of 1 through 6 characters in length. When specified, only those files whose names begin with that combination of characters are listed. A catalog spec is used with the CAT[ALOG] statement.

character A letter, number, symbol, or any eight bits defined by the CHR$ function.

character position Each position on a screen or printed page has a number associated with it. A screen has positions 1 through 80, while a printer has 1 through 132.

constant A fixed number within the system range, such as 2.12.

current line The next program line to be executed. Normally the first line in memory, unless the program was suspended by HALT or PAUSE.

data list A data list is a collection of items, separated by commas. The items can be variables, array identifiers, and numeric or string expressions. Including the optional END causes an EOF mark to be printed at the end of the data. Otherwise, an EOR mark is placed after the data list is written.

default device The device to which all file storage operations are directed if no other device is specified. The default device is defined in the global or user configuration file.

device address Each device connected to the HP 9000 responds to a unique address. The address is an integer between −2 and 99. For a list of reserved addresses, see page 249 in this manual.

display list A display list is used when displaying output to the screen. The list may contain variable names, array names, numeric expressions, string expressions, or TAB, SPA, LIN, and PAGE functions. All items on the list must be separated by commas or semicolons.

file name A 14 to 255 character string with the exception of a comma or a colon. It is recommended that a file name

not include wildcard characters. Included in the file name is a five character extension—.DATA, .PROG, .FORM, .ROOT, or .DSET. Refer to page 195 for more information.

file number The number assigned to a data file by an ASSIGN statement. Its range is 1 through 10.

file spec File spec is a string expression of the form: *file name*[*volume spec*]

The optional *volume spec* is needed when addressing a mass storage device other than the default device.

formal parameters Formal parameters are used to define a subprogram.

formal parameter list The formal parameter list is used to define subprogram variables and relate them to calling program variables. It includes non-subscripted numeric and string variable names, array identifiers, and file numbers. Parameters must be separated by commas, and the parameter list must be enclosed in parentheses.

format string Format strings consist of a list of specifiers, each separated by a delimiter. Each specifier creates part of an output format are numeric fields, string fields, blanks, or carriage control. Each numeric or string field specifier corresponds to an item in a print-using list, indicating how that item is to be output.

function A function is a routine that manipulates numeric or string data, and produces a numeric or string value as a result. A set of commonly used functions is supplied as part of the Eloquence language. These functions are known as built-in functions.

function name ($) Whatever variable name you give to a function of your own creation is that function's name. Built-in functions are already named, and are listed in page 125 of this manual.

integer precision variables Integer precision variables hold whole numbers only (no fractional part). Integer-precision numbers range from –32768 to 32767. They are held in binary 2's complement form (not exponent and mantissa).

key number There are 24 programmable softkeys, numbered 1 through 24.

line id A program line can be identified either by its line number (GOTO 150) or, if present, its label (GOTO Routine).

line id list This list consists of two or more line ids separated by commas. Computed GOTO statements use line id lists to branch according to the value of a variable.

line label A unique name assigned to a program line. It can contain up to 15 alphanumeric characters including the underscore. The first character must be a capital letter. The line label is separated from the line number by one or more spaces and must be followed by a colon.

line number An integer from 1 through 32767.

line width Line width is a numeric expression from 20 through 264, which specifies the number of characters output per line for PRINT and PRINT USING statements. If you omit line width, either the previously set line width or the default width for that device will be used (display default = 80 char, all others = 132 char). Specifying -1 sets an infinite line length.

matrix A matrix is a two-dimensional array.

multiple-line function subprogram The multiple-line function subprogram is designed to return a value to the call-

ing program, and is used like a built-in function such as SGN or CHR$. It is defined using the DEF FN statement.

number of linefeeds The number of linefeeds can be any numeric expression; rounded to an integer, its range is from -32768 through 32767. Linefeeds cause lines to be skipped. If a negative number is given for number of linefeeds, its absolute value is used.

number of records When creating a data file, you must specify the number records that you want your file to have. The number of records specified can be a numeric expression from 1 through 999999.

numeric expression The combination of one or two operands (values) with an operator constitutes a numeric expression. The operator may be arithmetic, string, relational, or operational.

numeric variables Numeric variables hold numbers, both positive and negative, integer or fractional. Numeric variables are themselves split into three types—integer, short and real.

numeric variable name A numeric variable name is from 1 to 15 characters, the first of which must be a capital letter. Remaining characters must be lowercase letters, digits, or the underscore character (_).

parameter Parameters are values in certain variables that are passed between a program and a subprogram. For example, the variable “Food$” may contain the value “Apple” in a main program. When the parameters are passed, “Apple” may be sent to the variable “Fruit$” in the subprogram. The variable name was not passed, but the value contained in that variable was.

parameter lists Values are passed between a subprogram and the calling program segment using parameter lists. There are two kinds of parameters—formal and pass.

pass parameters Pass parameters are used to pass values from the main or calling program to the subprogram. Each pass parameter must correspond to a formal parameter.

pass parameter list The pass parameter list used in calling the subprogram can include numeric and string variable names, array elements and identifiers, numeric and string expressions, and data file numbers. The pass parameter list must be enclosed in parentheses, and the parameters must be separated by commas.

program debugging Finding and fixing the mistakes in a program. Three methods of program debugging are available—single-stepping, program tracing, and program cross-referencing.

program segment The term program segment refers to either a main program or a subprogram.

prompt Prompts are any words or symbols that appear on the CRT screen when input is needed. INPUT and LINPUT statements use prompts to ask for specific data input. A prompt is not limited in any way, except possibly in length. An Eloquence line cannot exceed 160 characters; this means that the sum of a statement and its prompt may not exceed 160 characters.

real precision variables Real precision variables are allotted twelve significant digits of precision. They are the most accurate form of holding numeric data but take up the most space.

record length Record length is a numeric expression specifying the length of logical records in bytes, rounded up to an even integer. The length can be from 1 through 65534 bytes. If not specified, the default is 256 bytes.

record number Records are numbered sequentially from 1. You know how many records you have if you created the file. (See the definition for “number of records”)

redim subscripts Numeric expressions separated by commas and enclosed in parentheses to redefine array working bounds. (The number of dimensions cannot change, and the total number of elements cannot increase over the number originally dimensioned).

return variable A return variable can be a simple numeric variable or an array element. It is included in some statements to check for errors. Its value is returned after execution to indicate various results. The results are numbers that stand for different errors.

short precision variables Short precision variables hold whole or fractional numbers. They are represented internally with a mantissa of six significant digits and an exponent in the range from −63 through 63.

simple variable A simple variable holds either one number (simple numeric variable) or a string of characters (simple string variable).

standard printer The output device selected by the SYSTEM PRINTER IS statement.

statement Statements are any valid, executable program command. Statements may be one of two types: 1) Declaratory (for example, DIM, DATA, REM, SUB, COM) 2) Executable (for example, GOTO, STOP, INPUT, PRINT).

string A string is a series of ASCII characters which can be stored in a string variable. (In Eloquence a string may also hold display enhancement and line drawing characters.)

string expression String expressions may contain any combination of string characters within quotes, string variable names, substrings, and string functions.

string variable A string variable can hold any sequence of ASCII characters.

string variable name A string variable name has from 1 to 15 characters with $ added at the end. The first character must be a capital letter; the remaining characters must be lowercase letters, digits, or the underscore character (_).

subprogram A subprogram is a group of one or more statements that performs a certain task under the control of the calling program segment.

subprogram name A subprogram is separate from the program that calls it. Therefore, the subprogram has its own name consisting of one to six characters, including letters, digits, or the underscore. The subprogram name must be lowercase.

subroutine subprogram A subroutine subprogram is designed to perform a specific task. It is defined using the SUB statement.

subscript A subscript is used whenever a variable represents an array. A subscript consists of two or more integers separated by commas and enclosed in parentheses. These integers specify the array's size. The subscript comes immediately after the array variable name. For example, M(2,3) represents a two dimensional array named M with upper bounds of two and three. (2,3) is the subscript.

superuser The system administrator for the HP-UX operating system.

text A string of characters, quoted or unquoted.

unit spec A string expression of the form: :*A....Z*[*select code*[,*device address*[*unit code*]]]

volume letter: See the definition for "volume letters."

select code: An integer from 1 through 15.

device address: An integer from 0 to 9.

unit code: An integer from 0 through 9.

variable A variable describes a location in memory in which values can be stored. Computer languages use variable names to represent these locations. Then each time that variable name is quoted, the computer looks up the corresponding memory location and finds the value. The value contained within a variable may be altered, hence the name "variable".

variable list A series of numeric or string variables, separated by commas.

variable name A variable name is from 1 to 15 characters, the first of which must be a capital letter. Remaining characters must be lowercase letters, digits, or the underscore character (_). Each string variable name must end in a dollar sign ($). Variable names must be unique.

vector A vector is a one-dimensional array. The size of a vector is limited by memory size.

volume label A one- to eight-character string assigned to an HP-UX directory in either the global, group, or user configuration file. Blanks, nulls, commas, and colons are ignored.

volume spec A string specifying either a volume label (preceded by a comma) or a unit spec.

Index

A
statements
    CAT 205
arrays
    DIM 101
    elements 92
    numeric 90
    OPTION BASE 100
    REDIM 106
    size 90
    storage and retrieval 236
    string 78

C
C argument types
    EqChar 360
    EqInt 360
    EqReal 360
    EqString 360
    EqVoid 360
C function type 360
C functions
    dll_cleanup 366
    dll_setup 366
    EqArgInfo 365
    EqFuncInfo 364
    EqMaxStrlen 365
    EqMkstr 366
    EqStr2str 366
    EqStrcat 365
    EqStrcmp 366
    EqStrcpy 365
    EqStrlen 365
    EqSubstr 366
    str2EqStr 366
C variable

dll_debug 367
commands
CONTINUE 40
FETCH 41
INDENT 55
LIST 42
REN 40
RUN 39

D

development version
starting 36
display features
alternate character sets 284
field enhancements 272
formatted output (IMAGE) 280
input and output fields 275
protecting and unprotecting lines 274
restrictions on ASCII control characters 251

E

error messages
internal errors 35, 53
run-time errors 35, 53
syntax errors 35, 53

F

files
cataloging 205
closing 237
copying 247
creating 217
data access 201
data storage requirements 245
EOF 200
EOR 200
error trapping 243
locking 246
multi-tasking considerations 326

naming 198
opening 218
purging 238
records in 200
renaming 248
storage of 239
types of 200
functions
ABS 138
AOVFL 339
AREAD$ 338
built-in numeric 138
built-in string 143
CHR$ 143, 146
CLOCK 311
COL 307
CURKEY 138, 165, 342
DATE$ 310
defining 148
DROUND 138
ERRMSG$ 143
EXP 138
FRACT 138
GETENV$ 143
INT 138
LDSPEC$ 143
LEN 143
LGT 138
LIN 266
LOG 138
LWC$ 143
MAPPNTR$ 143
MAPVOL$ 143
MAX 138
MIN 138
MMSPEC$ 143
multiple-line 184
NUM 143, 146
NUMREC 138

OWNID 331
PAGE 267
PI 138
POS 143
PROUND 138
RANDOMIZE 138
REC 241
RES 138
REVISION 138
REVISION$ 143
RND 138, 142
ROW 307
RPT$ 143
SCAN 143
SGN 138
SHOWTASK 332
SIZE 241
SPA 266
SQR 138
SUM 307
SYSID$ 147
TAB 265
TASKID 138
TIME$ 310
trigonometric 140
TRIM$ 143
TSTAT 331
TYP 239
UPC$ 143
USERID 138
VAL 143, 145
VAL$ 143, 145
WRD 241
XOWNID 331
XPOS 272
YPOS 272

M

matrices

arithmetic operations on 305, 306
assigning values to 303, 304
functions 307
operations 308
reading and printing 300, 301
storing and retrieving 302
memory
consumption 124
multi-tasking
configuration requirements 317
database considerations for 327
error codes 321
example program 321
file access 326
functions 331
performance considerations 329
primary tasks 316
programming considerations for 325
secondary tasks 316
statements 318, 320
using HP-UX commands 322

O
operators
arithmetic 127
binary 134
definition of 126
logical 132
order of execution 136
relational 130

P
ports
device addresses 252
port numbers 252
printer numbers 252
printer 253
printers
multi-tasking 325

printer control functions 295
program debugging
cross-referencing 65
HOP 59
single-stepping 59
tracing 59
programs
comments within 30
entering 28
listing 55
multi-tasking considerations 325
running 50
stopping 51

R

run-time version
starting 22

S

SCRATCH 67
SFA 371
sfagen 373
sfarpt 374
SHOWTASK 332
SIGNAL 332
spool files 292
appending to 294
creating 292
dumping (COPY) 293
error handling 294
recording into 292
statements
ACCEPT 116
ASSIGN 218
ATTACH 319
ATTACH # 318
AUTO 40
BEEP 257
CALL 187

CALL DLL 357
CATCLOSE 207
CATGETMSG 207
CATOPEN 207
COM 103
COMMAND 30
COPY 247
COPY (spool files) 293
CREATE 217
CURSOR 268
DATA 107
DEF FN 148, 184
DEFAULT 137
DEL 41
DEL DLL 357
DEL FN 189
DEL SUB 189
DETACH 319
DIM 101
DINTEGER 103
DISABLE 165, 342
DISP 258
EDIT 112
ENABLE 165, 342
END 51
ENTER 113
FIXED 86
FLOAT 87
FN 148
FORNEXT 158
GET 212
GOSUB 162
GOTO 154
IFTHEN 156
IFTHENELSE 171
IMAGE 280
INPUT 110
INTEGER 102
KBCODE 116

LDISP 260
LENTER 114
LINK 213
LINPUT 111
LOAD 211
LOAD DLL 356
LOAD SUB 211
LOCK# 246
LOOPEND LOOP 172
MASS STORAGE IS 204
MAT INPUT 303
MAT PRINT 301
MAT READ 300
MAT-arithmetic 306
MATCON 303
MAT-copy 305
MAT-copy (scalar) 305
MATCSUM 308
MAT-initialize 304
MATRSUM 308
MATZER 304
MERGE 214
MSI 204
NORMAL 63
OFF DELAY 314
OFF END# 244
OFF ERROR 167
OFF HALT 169
OFF INPUT # 339
OFF KEY # 165
ON DELAY 314
ON END# 243
ON ERROR 167
ON HALT 169
ON INPUT # 337
ON KEY # 164
ONGOSUB 163
ONGOTO 155
OPTION BASE 100

PAUSE 51
PRINT 278
PRINT ALL IS 254
PRINT USING 280
PRINT# (direct) 228
PRINT# (direct-word) 232
PRINT# (serial) 222
PRINTER IS 253
PURGE 238
READ 107
READ LABEL 208
READ# (direct) 230
READ# (direct-word) 233
READ# (serial) 224
REAL 103
REDIM 106
REFRESH 264
RELEASE 341
RELEASE # 319
RENAME 248
REPEATUNTIL 173
REQUEST 255, 341
REQUEST # 318
RESAVE (RE-SAVE) 214
RE-STORE 211
RESTORE 109
SAVE 212
SCRATCH 67
SELECTEND SELECT 174
SHORT 103
SLEEP 51, 313
SOFTKEY On/OFF 166
SPACE DEPENDENT 33
SPACE INDEPENDENT 33
STANDARD 86
STOP 51
STORE 54, 209
SUB 187
SUBEND 187

SYSTEM PRINTER IS 253
TRACE 61, 62
TRACE ALL 62
TRACE ALL VARIABLES 62
TRACE PAUSE 61
TRACE VARIABLES 62
TRACE WAIT 61
UNLOCK# 246
WAIT 51, 312
WHILEEND WHILE 172
XASSIGN 218
XLENTER 115
XPURGE 238
storage
data 216
strings
definition of 78
null 81
string array defaults 79
string arrays 78
string concatenation 81
string expressions 79
substrings 79
subprograms
adding and deleting 189
busy lines 193
COM statements, use of 190
default state of 189
definition of 177
multiple-line function 184
parameters 179
subroutine 187

T
trigonometric functions 140
TYPEOF$ 144

V
variabels

numeric precision 84
variables
assigning values to 107
declaring 100
dimensioning 100
display formats 85
format FIXED 86
format FLOATing-point 87
format STANDARD 86
integer/dinteger numeric ranges 84
names 77
numeric 84
real numeric ranges 84
rounding of 88
string 78
types of 75

Y
YPOS 272