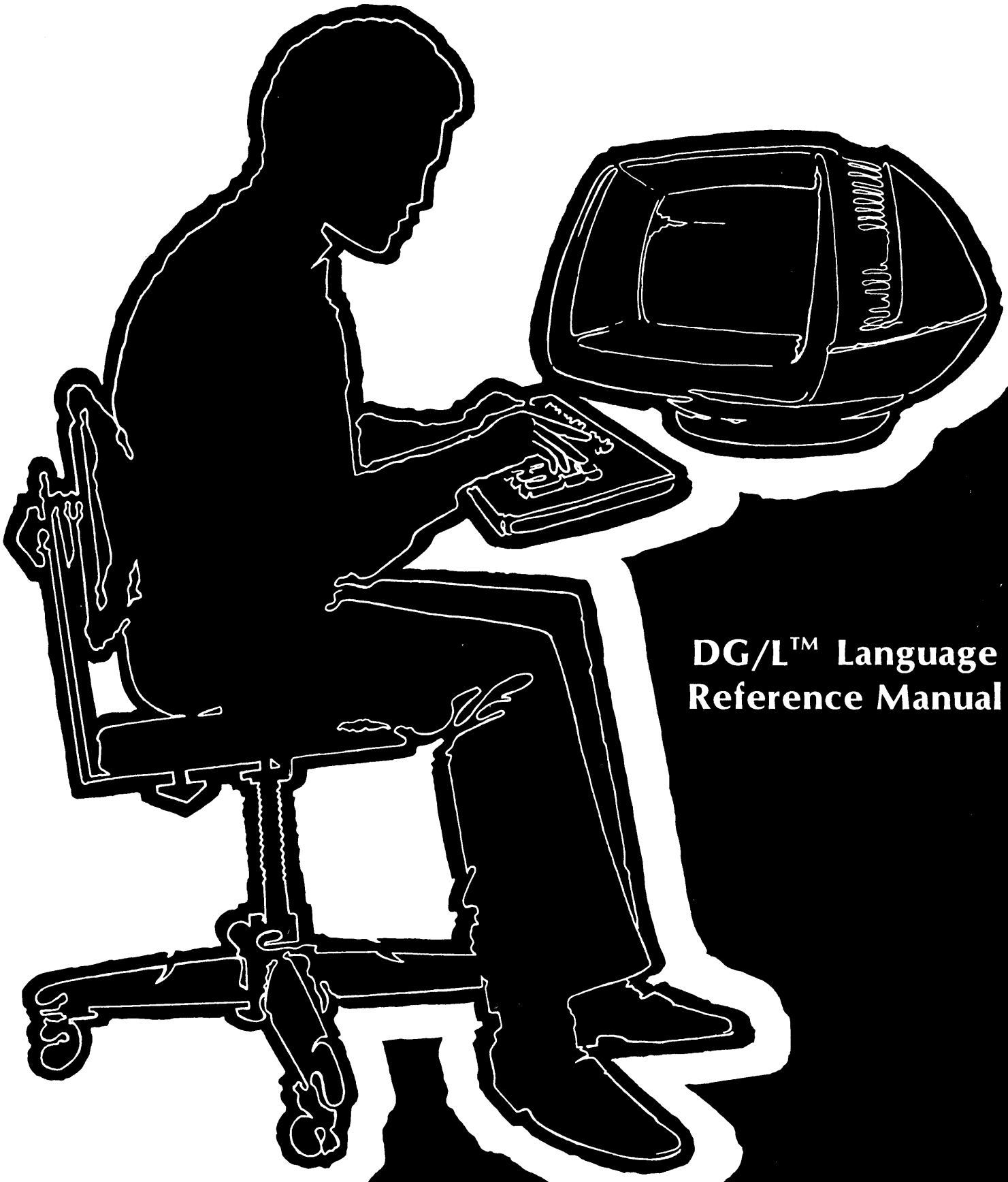
Data General



DG/L™ Language Reference Manual

DG/L™ Language Reference Manual

093-000229-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000229

Printed in the United States of America
Revision 01, February 1982

NOTICE

DG/L™ Language
Reference Manual
093-000229

Revision History:

Original Release - October 1977

First Revision - February 1982

Effective with:

DG/L Rev. 3.00

This document has been extensively revised from the previous revision; therefore change indicators have not been used.

Preface

This manual shows you how to write, compile, link, and execute programs written in the DG/L™ language. The syntax of the DG/L™ language is simple enough to allow you to write small programs easily, yet flexible enough to allow you to write large and complex programs such as compilers.

DG/L programs will run on either the ECLIPSE® or NOVA® computers under any of the following operating systems: AOS/VS, AOS, RDOS, RTOS, DOS. References to "the system" in this manual apply to any of these operating systems. This manual mentions systems by name where differences are crucial.

The DG/L and ALGOL 60 Languages

The DG/L language has the block structure of ALGOL 60 and most of its syntax. However, it gives you data types, operations, and statements that ALGOL 60 lacks. In addition, it has a large number of runtime routines. Appendix A sketches the ways the two languages differ.

The DG/L compiler optimizes as it compiles, producing efficient code, and providing extensive compile-time and runtime error checking.

The Reader of This Manual

The intended reader for this manual must already be familiar with the DG/L programming or some other high-level programming language. The manual is primarily a reference, not a tutorial guide. To ascertain whether this manual is appropriate for your needs, read Chapter 1, "An Overview of the DG/L Language."

Organization of the Manual

Chapter 1, "An Overview of the DG/L Language," sketches general properties of the language, outlines levels of structure from program modules to permitted symbols, and gives procedures for compiling, linking, and executing DG/L programs.

Chapter 2, "Symbols and Keywords," lists required, permitted, and prohibited symbols and symbol sequences.

Chapter 3, "Expressions," describes arithmetic, Boolean, string, and assignment expressions. It characterizes precision, range, and exponential notation for real and integer arithmetic. It outlines data type conversion, and specifies the order of precedence for operators.

Chapter 4, "Statements," presents the required and optional elements of the syntax of DG/L statements.

Chapter 5, "Declarations," describes three classes of declarations (type, shape, and storage class), and three other declarations.

Chapter 6, "Procedures," explains the PROCEDURE declaration. It distinguishes proper procedures from function procedures, describes external procedures, and sketches the recursive use of procedures. It outlines parameter-passing rules, side effects, and the construction of clusters.

Chapter 7, "Block Structure," describes nesting blocks of statements, and the scope of variables in them.

Chapter 8, "Built-in Functions," presents arithmetic and general functions from the runtime library.

Chapter 9, "Built-in Routines," presents general and operating system interface routines from the runtime library.

Chapter 10, "Applications Development Aids," describes cache memory management, suggests additional data reference methods, and sketches the INCLUDE facility and overlay support.

Chapter 11, "Operating Procedures," lists the options available in compiling and linking DG/L programs.

Appendix A characterizes in detail the differences between DG/L and ALGOL 60.

Appendix B annotates compiler error messages.

Appendix C gives the ASCII character set.

Appendix D contains octal and hexadecimal conversion charts.

Related Manuals

DG/L™ Runtime Library User's Manual (AOS and AOS/VS) 093-000159

DG/L™ Runtime Library User's Manual (RDOS) 093-000124

AOS Programmer's Manual 093-000120

AOS/VS Programmer's Manual 093-000241

AOS Link User's Manual 093-000254

AOS/VS Link and Library File Editor User's Manual 093-000245

AOS Macroassembler Reference Manual 093-000192

AOS/VS Macroassembler Reference Manual 093-000242

Reader, Please Note:

We have used the terms console and terminal interchangeably in this manual.

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where	Means
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: $\left\{ \begin{matrix} \text{required}_1 \\ \text{required}_2 \end{matrix} \right\}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[optional]	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	Means
⤶	Press the NEW LINE or carriage return (CR) key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, in examples we use

THIS TYPEFACE TO SHOW YOUR ENTRY⤶
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

) is the AOS and AOS/VS CLI prompt.

R is the RDOS CLI prompt.

Contacting Data General

If you:

- Have comments on this manual -- Please use the prepaid Remarks Form that appears after the Index.
- Require additional manuals -- Please contact your local Data General sales representative.
- Experience software problems -- Please notify your local Data General systems engineer.

End of Preface

Contents

Chapter 5 - Declarations

Chapter 6 - Procedures and Clusters

Chapter 7 - Block Structure

Chapter 8 - Built-in Functions

Chapter 9 - Built-in Routines

Chapter 10 - Applications Development Aids

Chapter 11 - Operating Procedures

Appendix A - DG/L and ALGOL 60

Appendix B - Compiler Error Messages

Illustrations

Tables

Chapter 1
An Overview of the DG/L™ Language

The DG/L programming language provides you with high-level programming tools. It is simple enough that you can write short programs quickly, but flexible enough that you can construct complex compiler programs.

The DG/L language has two major features: nesting syntactic categories and converting data types.

The nesting of syntactic categories means that a statement may contain statements, a block may contain blocks, procedures may contain other procedures.

The other major property is the automatic conversion of data from one type to another. This permits you to assign or combine data elements of differing types without having to convert the data types explicitly. You must specify a type for each piece of data you use in the program, stating, for example, whether the program is to interpret a variable containing a sequence of digits as a real number, an integer, or a character string. If, later in the program, you attempt to assign a character string as a value to a variable of integer type, the DG/L runtime environment automatically converts the string to an integer value and then carries out the assignment.

Levels of Structure

A DG/L program consists of one or more *program modules,* or self-contained sections of code. One module is the *main program.* The others are *external procedures* or *clusters* of external procedures. You can compile these program modules separately, and load them on call at linking time. This allows you to reuse a program module in different programs, or to revise part of a program without recompiling it totally.

A program module consists of at least one *block,* and may contain many blocks, in sequence or nested in one another.

A block has two necessary components: a *declaration* part which comes first, and a *statement* part which follows.

The *declarations* in a block allow you to specify what *variables* and *procedures* are to be valid in that block.

A procedure can be a simple statement, compound statement, or block in its own right, with its own interior declarations. You declare an *internal procedure* in the block that will call it; you declare an *external procedure* in a separate file or program module. Once you have declared a procedure, you can call it any number of times in any block for which it is valid.

The *statements* that follow the declarations of a block represent the executable code of the program. They may be *simple* or *compound.* Simple statements consist of *keywords* and *expressions.* A compound statement consists of a sequence of statements between the BEGIN and END keywords. Each statement ends with a delimiter, the semicolon (;). Normally each statement executes in turn, but the DG/L language provides labels and branching devices that allow you to change the sequence of execution.

The keywords specify how you will treat the expressions. The expressions consist of *variables, constants,* and *operators,* and contain the data of your program. The keywords and expressions consist of legal sequences of *symbols,* the ultimate constituents of a DG/L program module.

The chapters that follow describe these elements of DG/L structure in reverse order, building from symbols through expressions, statements, declarations, procedures, and clusters to the block structure of entire programs.

Using the DG/L Language

After you have constructed a DG/L program with a text editor, you must *compile* it. Assuming that you are compiling for execution under AOS or AOS/VS, the minimal compilation command line for your file TEST has the form

```
) XEQ DGL TEST↵
```

When you run the compiler, any error messages appear at the terminal. You must re-edit and recompile your file TEST until no error messages appear.

A program module must contain at least one declaration. If it contains only statements, it will not compile. A program module (a cluster, for example) may however contain only declarations and no executable statements.

Once you have a file with no compilation errors, you must *link* it. Assuming that you are working in a 32-bit AOS/VS environment, the minimal link command line for your file TEST has the form

```
) XEQ LINK/NLNS TEST [DGLIB32]↵
```

(The brackets are required.) Chapter 11, "Operating Procedures," describes optional switches, other options, and execution under other systems.

When you have successfully linked your program, you can execute it. You execute your DG/L program TEST with

```
) XEQ TEST↵
```

Figure 1-1 diagrams some of the relations of the larger DG/L programming units. The program on the right, in the dotted box, consists of three program modules: a main program, an external procedure, and a cluster of external procedures. You compile these separately, but link them together with a single command. The main program shows blocks nested within blocks. On the left is the internal structure of a typical block, with all declarations preceding all statements.

The next section in this chapter illustrates the structure and construction of a DG/L program.

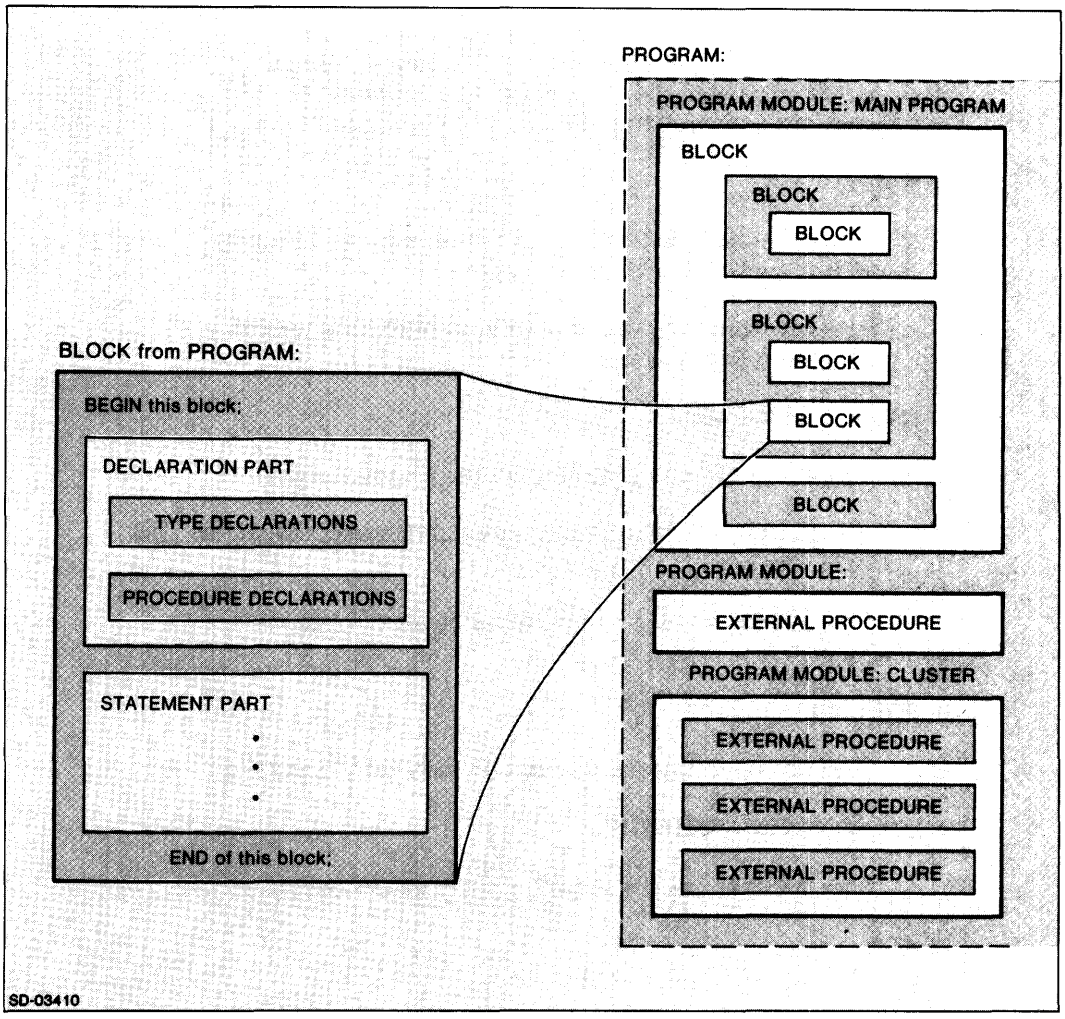


Figure 1-1. Some DG/L Programming Units

A Sample Program

The program in Figure 1-2 calculates properties of your spacecraft while it moves at velocities below the speed of light, as those properties would appear to an observer at rest. To such an observer, as your velocity increases, time on board seems to stretch out, your spacecraft seems to shorten in the direction of travel, and its mass seems to increase.

The program exemplifies block structure, the calling of a procedure, the manipulation of a variety of variables and constants, and the use of external procedures in an interactive program.

The executable statements of the main program begin on line 104 (the program listing file automatically provides line numbers). The program contains its own commentary (in color).

```
 1 /* This is a program to calculate relativistic effects for
 2 various velocities of your spacecraft.  This program requires
 3 no external procedures other than built-in DG/L runtimes for
 4 commmunication with the terminal.*/
 5
 6 BEGIN
 7
 8 /* You specify global declarations, valid over the entire program,
 9 as the first part of this outermost block.*/
10
11    /* You can substitute C for this value throughout the program
12    by declaring this literal.*/
13
14 LITERAL C (299792.458);
15
16    /* You set up the real variables you need for calculation.*/
17
18 REAL (4) VELOCITY, FRACTION_OF_C, LORENTZ_FITZGERALD_CONTRACTION, TIME,
19    LENGTH,  MASS;
20
21    /* You use this string to store replies to queries from the
22    program.*/
23
24 STRING YES;
25
26    /* You declare an internal procedure to carry out calculations.*/
27
28 REAL (4) PROCEDURE RELATIVITY;
29
30    /* The procedure contains a block with its own declaration.*/
31
32 BEGIN
33
34
35    /*  You declare a Boolean to control loop termination.*/
36
37   BOOLEAN DONE;
38
39    /* This ends the declaration part of the block and begins
40    the executable statements of the block.  The DO statement is
41    controlled by a trailing UNTIL, and contains a compound
42    statement.*/
43
44   DO
45     BEGIN
46       WRITE (1, "<NL>What is your current speed (km/sec)?  ");
47       READ (0, VELOCITY);
48       FRACTION_OF_C := VELOCITY/C;
49
50    /* Beware of malicious users.  The IF statement checks to
51    see that the value the user inputs is reasonable.  If it
52    is, control passes to the ELSE clause of the IF statement.*/
53
```

Figure 1-2. Sample Program RELATIVITY (continues)

```
54      IF (FRACTION_OF_C >= 1) OR (FRACTION_OF_C =< 0) THEN
55        WRITE (1, "Impossible speed:  check your instruments.<NL>")
56
57    /* Since the ELSE clause contains several statements, you
58    must bracket them with the BEGIN and END keywords.*/
59
60      ELSE
61        BEGIN
62
63   /* You perform calculations and write them to the terminal
64   with the following statements.*/
65
66          LORENTZ_FITZGERALD_CONTRACTION := (1-(FRACTION_OF_C)^2.)^.5;
67
68          TIME := LORENTZ_FITZGERALD_CONTRACTION*60;
69          WRITE (1, "Time dilation is ", TIME,
70                    " minutes per hour.<NL>");
71
72          LENGTH := LORENTZ_FITZGERALD_CONTRACTION*1000;
73          WRITE (1, "Length contraction is to ", LENGTH,
74                    " meters.<NL>");
75
76          MASS := 500./LORENTZ_FITZGERALD_CONTRACTION;
77          WRITE (1, "Apparent mass is ", MASS, " tons.<NL>");
78        END of the compound statement;
79
80    /* The program asks you whether to continue.*/
81
82      WRITE (1, "If you are changing speed, type y:  ");
83      READ (0, YES);
84      IF YES = "y" THEN
85        DONE := FALSE
86      ELSE
87        BEGIN
88          WRITE (1, "Then Bon Voyage!<NL>");
89          DONE := TRUE
90        END;
91     END UNTIL DONE;
92 END of procedure RELATIVITY;
93
94 /* You continue with declarations for the main program.  Below
95 are the externals you need for input and output.*/
96
97 EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
98
99 /* Declarations of the outermost block end here, and executable
100 statements of the main program begin.*/
101
102    /* You open channels for input and output.*/
103
104   OPEN (0, (GETCINPUT));
105   OPEN (1, (GETCOUTPUT));
```

Figure 1-2. Sample Program RELATIVITY (continued)

```
106
107     /* You send an introductory message to the user */
108
109   WRITE (1, C, " km/sec is the current value for c.<NL>",
110       "Your craft has a rest length of 1000 meters,<NL>",
111       "a rest mass of 500 tons, and can travel between<NL>",
112       "0 and ", C, " km/sec.<NL>");
113
114     /* You call the previously declared procedure to perform
115    and display the relativistic calculations.*/
116
117   RELATIVITY;
118
119     * Control returns here when you end the execution of
120    the procedure.  The program ends and you return to the
121    parent process.*/
122
123 END of program for relativistic effects;
```

Figure 1-2. Sample Program RELATIVITY (concluded)

A sample execution of the program, as it would appear on your screen, follows in Figure 1-3 (the user's replies to queries from the program are in color).

```
299792.458 km/sec is the current value for c.
Your craft has a rest length of 1000 meters,
a rest mass of 500 tons, and can travel between
0 and 299792.458 km/sec.


What is your current speed (km/sec)?  150000.
Time dilation is 51.949527953549 minutes per hour.
Length contraction is to 865.825465892483 meters.
Apparent mass is 577.483591897595 tons.
If you are changing speed, type y:  y

What is your current speed (km/sec)?  299792.
Time dilation is .1048789836119 54 minutes per hour.
Length contraction is to 1.74798306019923 meters.
Apparent mass is 286043.961972384 tons.
If you are changing speed, type y:  y

What is your current speed (km/sec)?  0.
Impossible speed:  check your instruments.
If you are changing speed, type y:  y

What is your current speed (km/sec)?  400000.
Impossible speed:  check your instruments.
If you are changing speed, type y:  y

What is your current speed (km/sec)?  -1000.
Impossible speed:  check your instruments.
If you are changing speed, type y:  y

What is your current speed (km/sec)?  34567.89012
Time dilation is 59.5998006445713 minutes per hour.
Length contraction is to 993.330010742855 meters.
Apparent mass is 503.357388372952 tons.
If you are changing speed, type y:
Then Bon Voyage!
```

Figure 1-3. Sample Execution of the RELATIVITY Program

End of Chapter

Chapter 2
Symbols and Keywords

This chapter outlines the basic syntactic building blocks that make up all DG/L programs. It describes reserved symbols, identifiers, and constants.

Reserved Symbols

Certain DG/L symbols and symbol strings are *reserved.* They play central roles in its syntax. Use them as their descriptions direct; avoid additional, idiosyncratic uses.

Keywords

DG/L syntax reserves *keywords* for use as command names (or parts of names), operators, declarators, constants, and functions or routines (indicated by superscripts in the following list). You cannot redefine any keyword for some other use, for example as an identifier.

AND[o]	ELSE[c]	GOTO[c]	OUTPUT[f]	SUBSTR[f]
ARRAY[d]	END[c]	IF[o,c]	OVERLAY[d]	SWITCH[d]
BASED[d]	EQV[o]	IMP[o]	OWN[d]	THEN[c]
BEGIN[c]	EXTERNAL[d]	INCLUDE[c]	POINTER[d]	TRUE[b]
BIT[d]	FALSE[b]	INTEGER[d]	PROCEDURE[d]	UNTIL[c]
BOOLEAN[d]	FOR[c]	LABEL[d]	READ[f]	VALUE[d]
COMMENT[c]	FORMAT[f]	LITERAL[d]	REAL[d]	WHILE[c]
CLUSTER[d]	GLOBAL[d]	NOT[o]	STEP[c]	WRITE[f]
DO[c]	GO TO[c]	OR[o]	STRING[d]	XOR[o]

b: Boolean constant
c: command name or command name part
d: declarator
f: function or routine
o: operator

Keywords may consist of any combination of uppercase and lowercase characters. You must not intersperse spaces or other delimiters within a keyword (except GO TO).

Terminator

The DG/L statement *terminator* is the semicolon (;). It is analogous to the end punctuation of English sentences, such as a period or exclamation point. Each statement and declaration in a DG/L program must end with a semicolon. But it cannot separate parts of a simple statement. For example, it must not precede the ELSE of an IF statement. Chapter 4, "Statements," details the behavior of the ; terminator.

Delimiters

You must separate identifiers and arithmetic constants from each other with *delimiters* in your DG/L program. Delimiters include

- operator symbols
- separator symbols
- comment delimiters
- bracketing symbols
- spaces, which include the following characters:

  ASCII blank

  horizontal tab

  NEW LINE

  form feed

You may use more than one space anywhere you can use a space, and you may precede or follow other delimiters with spaces. The expressions

SUM+A

and

SUM + A

are equivalent.

Operators

Certain simple and complex nonalphabetic symbols represent arithmetic, relational, string, pointer, and assignment *operators*. Table 2-1 lists them.

Table 2-1. Operators

Arithmetic	Relational	Other
*	<	!! (string)
+	<= or =<	-> (pointer)
-	<> or ><	:= (assignment)
/	=	
	=> or >=	
	>	

Separators

You must punctuate certain parts of DG/L statements and expressions with the *separators* listed in Table 2-2.

Table 2-2. Separators

Symbol	Separates
,	elements of a list
.	whole from fractional parts of real numeric constants
:	labels from their statements, lower from upper bounds in array declarations
D or E	mantissa from exponent in numerical constants
P	numerical constant from precision specifier
R	numerical constant from radix specifier

Comment Forms

You may construct DG/L comments and intersperse them in your programs in several different ways.

/* */ You may place a comment between these paired compound symbols. The first character following the asterisk (*) must not be another asterisk if the next character is a letter or digit; the compiler will interpret this sequence as code to compile conditionally.

% The rest of a line following this symbol functions as a comment.

):(You may replace a comma in a list with a string consisting of a right parenthesis, zero or more spaces, a string, a colon, zero or more spaces, and a left parenthesis. That is, replace

```
list-item1, list-item2
```

by

```
list-item1) DOESONE: (list-item2
```

This allows you to insert descriptors for elements of a list. The characters in the string must be alphabetic.

In addition, you may construct COMMENT statements of the form

COMMENT textstring;

where the textstring contains no semicolon (;), not even one enclosed in quotation marks. You may also insert comments between the END of a compound statement and its semicolon terminator in the format

END textstring;

where the textstring contains no semicolon, nor the keywords ELSE, END, UNTIL, or WHILE. Chapter 4, "Statements," describes the COMMENT and BEGIN...END Compound statements.

Bracketing Symbols

You must enclose certain DG/L strings with paired bracketing symbols.

(...)	Parentheses enclose expressions, parameters passed to procedures and functions, and precision specifiers for numeric, bit, and string variables and constants.
[...]	Square brackets enclose specifications of array dimensions, and subscripts for arrays, labels, and switches.
<...>	Angle brackets enclose ASCII character representations within quoted strings.
`...'	Grave and acute accents enclose string constants and literals.
"..."	Double quotation marks enclose string constants and literals.
/**...*/	A slash and two asterisks precede a section of conditionally compiled code, and a single asterisk and slash follow it.

Identifiers

DG/L *identifiers* name variables, literals, labels, and procedures. They must begin with a letter or a dollar sign ($), and they may contain letters, numeric digits, dollar signs, and underscore (_) characters. They must not contain colons, spaces, or other delimiters. If you use the /Q switch at compilation time (Chapter 11), you may include question marks (?) in identifiers.

Identifiers may take any length, but only the first 32 characters are significant. Uppercase and lowercase letters are distinct, and may occur in any combination.

The identifiers

```
AREA
```

and

```
AREa
```

are two separate identifiers. The two strings

```
Read_thirty_three_characters_here
```

and

```
Read_thirty_three_characters_herE
```

are instances of the same identifier, since they do not differ in the first 32 characters. The identifiers

```
A_25
zap
GET_NAME
$_3$$
```

are all legal, but

```
A 25
```

is not legal because it contains a space,

```
_205
```

is not legal because it begins with an illegal character, and

```
A/B
```

is not legal because it contains a character that identifiers do not permit.

Variables

Some identifiers represent *variables.* A variable describes a location in memory and associates with it a data type, precision, shape, and storage class. The variable may represent a numeric value, a Boolean value, a bit string, a character string, or an array of any of these types. The value of a variable may change at different places during runtime in a DG/L program, unlike constant values, label values, literals, or based variable templates, which you can change only if you recompile or debug the program. You must *declare* each variable (Chapter 5, "Declarations"), and appropriately specify its type, precision, shape, and storage class. You must also *initialize,* or specify an initial value, for each variable. If you do not, the DG/L compiler supplies certain *default* initializations (see Appendix C).

When you include an identifier in a DG/L statement, you refer to the variable it represents. For example, if you declared A to be a real variable, the assignment statement

```
A := A + 11.5;
```

adds the real numeric constant 11.5 to whatever is the current value of variable A. If the previous value was 2.2, the new value is 13.7. If you declared Sentence, Noun, and Verb to be string variables, the assignment statement

```
Sentence := Noun!!"□"!!Verb;
```

concatenates the character string in Noun with a space and the character string in Verb, and stores the result in Sentence. If the current value of Noun is "Fish" and the current value of Verb is "fly." the new value of Sentence is "Fish fly." If you declared TABLE to be a 5-by-4 Boolean array whose current values are

	0	1	2	3	4
0	TRUE	TRUE	TRUE	TRUE	FALSE
1	FALSE	FALSE	FALSE	TRUE	TRUE
2	FALSE	TRUE	TRUE	TRUE	TRUE
3	FALSE	FALSE	TRUE	FALSE	FALSE

the assignment statement

```
TABLE[1,3] := TRUE;
```

changes the value of the cell in the second column, fourth row from FALSE to TRUE (indexes for dimensions start with 0 by default).

Literals

Other identifiers represent *literals,* which you introduce with LITERAL declarations (Chapter 5). Literals may represent variables, numeric or string constants, or expressions, but they may not replace keywords or statements.

If you declared

```
LITERAL FILENAME ("$LPT");
```

then the subsequent routine

```
OPEN (1, FILENAME);
```

opens $LPT as file number 1.

If you declared

```
LITERAL PI (3.1416);
```

then the program evaluates a subsequent expression (4.0/3.0)*PI with a value of 3.1416 for PI. If you use literal identifiers at all points in the program rather than the constant value or string, you can make global changes in the program by changing only a single line, the LITERAL declaration. Thus, if you change the declarations above to

```
LITERAL FILENAME ("$LQP");
```

and

```
LITERAL PI (3.14159);
```

then, after you recompile the program, the program will manipulate $LQP instead as file number 1, and will compute instead with 3.14159 for PI.

Labels

Identifiers also represent *labels*. A label marks the statement following it in a DG/L program. You must insert a colon between the label and its statement. In

```
TAG: A := MAT[3,3];
```

the identifier

```
TAG
```

labels the statement

```
A := MAT[3,3];
```

Other statements in the program may use the label to pass program execution to the labelled statement.

You may label a statement which already has a label. In

```
UP: UP_AND: AWAY: SUPER := CLARK!!KENT;
```

UP labels the statement beginning with UP_AND, which labels the statement beginning with AWAY, which labels the statement beginning with the variable SUPER.

Labels may have integer subscripts. This lets you specify branches to computed points in your programs. You may use any integer in the range -32768 to 32767 as a label subscript. Enclose the subscript in square brackets, as in

```
OPTION[-13]:
```

or

```
ALTER[21]:
```

A label may have as many subscript values within the range as you wish, and they need not be consecutive. You should realize, however, that nonconsecutive label subscripts result in wasted memory space at runtime, since a consecutive array of transfer addresses is generated from the lowest defined subscript to the highest.

You may also use integers as labels. For example,

```
GOTO 23;
.
.
.
23: CHECKNULL := 0;
```

passes control from the GOTO statement to the statement labelled 23. Integer labels are not integer data, so you cannot do arithmetic on integer labels.

You declare labels implicitly when you write them into your program. Only labels passed to a procedure require declaration; see Chapter 6.

Procedures

Finally, identifiers represent *procedures*. A procedure may contain its own declarations, valid only within the procedure. It consists of these optional declarations and a single statement, which can be either a simple statement, a compound statement, or a block. You *declare* the procedure in the declaration part of the block that contains it, and you *execute* the procedure when you call it by name in the statement part of the containing block. Chapter 1 sketches the role of a procedure in a simple program. Chapter 6 provides the details of procedure structure. In a block containing

```
PROCEDURE EVAL (MAX, MIN, AVG);
     REAL MAX, MIN, AVG;
     AVG := (MAX + MIN) / 2.0;

main-program-statement1;
.
.
.
main-program-statementi;
EVAL (X, Y, Z);
main-program-statementk;
.
.
.
```

execution passes from $statement_1$ through $statement_i$ to the next statement, the procedure call statement EVAL. At that point, execution passes to the declarations and statement above in PROCEDURE EVAL, with the values of X, Y, and Z (declared and initialized elsewhere) for the arguments MIN, MAX, and AVG. When the computation has been completed and the result stored in AVG, execution passes back to $statement_k$.

Constants

DG/L programs may use both *numerical* and *string* constants.

Numerical Constants

You may use either *integer* or *real* numerical data. If you write a constant with a decimal (or other radix) point, you implicitly declare its type as real. If you enter no decimal (or other radix) point, you implicitly declare its type as integer.

You may specify a radix when writing an integer or real constant, by putting an integer from 2 to 10 after the specifier R. For example, you specify the octal 126^8 by

```
126R8
```

A number containing a digit illegal for the specified radix results in an error.

You may specify either integers or reals as single or double precision, and signed or unsigned.

A single-precision integer may take any value between -32768 and 32767, inclusive (5 significant decimal digits). A double-precision integer may take any value between -2147483848 and 2147483847 inclusive (10 significant decimal digits).

A single-precision real can take any value between 10^{-78} and 10^{75} with 7 significant digits. A double-precision real can take any value in the same range with 16 significant decimal digits.

The precision of the value of a variable is the precision you declare for the variable. You may specify the precision of an integer or real constant by writing P and an integer after the number. For example, P1 specifies a single-precision integer, Pn ($n > 1$) specifies a double-precision integer; P1 or P2 specifies a single-precision real, Pn ($n > 2$) specifies a double-precision real.

If you do not specify a precision for a constant, the constant takes the smallest legal precision that will represent its numeric value as a signed quantity. For example,

```
177777R8
```

generates a double-precision integer, since there are more than 5 significant digits, but

```
177777R8P1
```

forces the generation of a single-precision integer.

If you attempt to use a precision specification too small to store the indicated constant, extra more-significant (leftmost) bits are lost.

You may write real or integer constants in exponential form by placing either D or E before the exponent (the two symbols function identically). The exponent must be an integer, and you must enter a value, whether integer or real, before the exponent. For example, both

```
3D-17
```

and

```
3E-17
```

represent the integer 3×10^{-17},

```
3.0E-17
```

represents the real 3.0×10^{-17}, and

```
E-17
```

with no immediately preceding numeric value represents no legal number.

The exponentiation markers E or D must precede any precision or radix markers, but you may use the separators P and R in either combination. For example,

```
.1E2R3P4
```

specifies a double-precision real number with the value 3.0000....When radix and exponential notations appear in the same number, the value of the exponential multiplier is the radix raised to the exponent, rather than the value 10 raised to the exponent. For example,

```
5E3R8
```

represents 5 x 8^3 rather than 5 x 10^3.

String Constants

DG/L syntax lets you create a string constant by enclosing a string of characters in quotation marks. You may print a string constant, assign its value to a variable, select characters from it with the SUBSTR function, or search for characters in it with the INDEX function. You can transform one string into another by respecifying its length with the LENGTH and SUBSTR functions, or by concatenating it with other strings using the concatenation (!!) operator. After you declare a string variable

```
STRING SELFREF;
```

you can assign a string to it with the statement

```
SELFREF := "This sentence contains it";
```

and if you respecify SELFREF with the concatenator

```
SELFREF := SELFREF!!"s own last word";
```

its new value becomes

```
"This sentence contains its own last word"
```

Spaces are significant characters within strings. The compiler generates all string constants with at least one final null byte, but does not count it in the length of the string.

You can create a string constant that contains carriage control, NEW LINE, or other nonprinting characters. Within the quotation marks, enclose the ASCII octal value of the character in angle brackets. For example,

```
"<14>"
```

represents a form feed.

You may represent certain ASCII characters within strings with symbolic names, which you must also enclose in angle brackets.

Symbolic Name	Octal Value	Description
<BEL>	<7>	bell character
<CR>	<15>	carriage return
<DEL>	<177>	delete
<ESC>	<33>	escape character
<FF>	<14>	form feed
<HT>	<11>	tab
<LF>	<12>	line feed
<NL>	<12> AOS, AOS/VS <15> RDOS	NEW LINE
<NUL>	<0>	null character
<QT>	<42>	quotation mark
<LAB>	<74>	left angle bracket
<RAB>	<76>	right angle bracket

For example, if you call the routine (Chapter 9)

```
WRITE(1, "First<NL>Second<NL>Third");
```

you write

```
First
Second
Third
```

to the file opened as unit 1.

End of Chapter

Chapter 3
Expressions

The expression is a basic building block of DG/L statements. It represents a value. Expressions are made up of operators, variables, strings, logical values, numbers, and function designators. You can nest them to any depth. There are five main categories of expressions: arithmetic expressions, Boolean expressions, pointer expressions, string expressions, and designational expressions. Each of these evaluates to a single value, whose type is described in the corresponding section. Each type of expression can also contain conditional and assignment subexpressions, which let you specify values conditionally or assign values within expressions. For example,

```
A := (C := (IF BOOL THEN 1 ELSE B))
```

is a valid DG/L expression. It first evaluates

```
(IF BOOL THEN 1 ELSE B)
```

either to 1 or to B, depending on the value of BOOL. The variable C then takes that value, and A receives the value of the outer parenthesized subexpression

```
(C := (IF BOOL THEN 1 ELSE B))
```

Since an expression must evaluate to a single and definite result in all cases, a conditional expression must include both a THEN clause and an ELSE clause. For example,

```
IF BOOL THEN 17
```

is not a legal expression, since there is no ELSE clause to evaluate if BOOL is false.

Arithmetic Expressions

A DG/L arithmetic expression represents a mathematical formula that evaluates to a single numeric value. It can consist of a combination of constants, variables, functions, and arithmetic or other types of expressions, all joined by arithmetic operators, as long as the combination is syntactically valid. It can include pointer expressions, assignment subexpressions, and conditional subexpressions. DG/L does not require that all identifiers making up an arithmetic expression be of the same numeric type, or even that they be of numeric type at all. Type conversions take place if necessary. When you use more than one operator in composing an expression, the rules of precedence determine the order of operator evaluation.

The primitive operators used to compose DG/L arithmetic expressions are

+	addition	expr + expr
/	division	expr / expr
^	exponentiation	expr ^ expr
*	multiplication	expr * expr
:=	replacement	var := expr
-	subtraction	expr - expr
-	unary minus	-expr
+	unary plus	+expr

For the precedence relations of all operators, including arithmetic ones, see the section "Precedence" in this chapter.

Some examples of arithmetic expressions:

```
4
```

represents the numeric value 4.

```
4 * B
```

represents the product of B and 4.

```
A * B + COS (C)
```

evaluates the cosine of C, and then adds the result to the product of A and B. Since addition has a lower precedence than multiplication, the multiplication is performed first.

```
A *- B
```

negates B, and multiplies the result by A. The unary operators + and - may occur next to other operators without intervening operands.

```
M[I] := M[I-1]
```

respecifies one array element with the value of another. This forms an assignment subexpression, whose value is the value of the leftmost part.

```
(PTR -> BI) := 12
```

assigns the arithmetic expression 12 to the memory location described by the pointer expression PTR -> BI.

```
IF (G > 0) THEN SIN (1.0/G) ELSE 0.0
```

is a conditional arithmetic expression. Depending on the relationship between G and 0, this expression will be evaluated either to SIN (1.0/G) or to 0.0.

```
M[I] := M[IF (Y < 0) THEN N ELSE (N + 5)]
```

uses another conditional expression to select an array subscript.

```
IF (C = 0) THEN (A ^ C) ELSE (A ^ (C := C+1))
```

is an arithmetic expression incorporating both an assignment subexpression and a conditional subexpression. The assignment subexpression, C := C+1, is executed only if the value of the Boolean expression C = 0 is FALSE, selecting the expression's ELSE clause. The entire expression's value is A ^ C, with C conditionally incremented before the exponentiation only if C is not equal to 0 before this expression is executed.

Boolean Expressions

A DG/L Boolean expression represents a formula that is evaluated to one of the Boolean values TRUE or FALSE. It is made up of a combination of constants, variables, functions, and Boolean or other types of expressions, joined by Boolean and relational operators. It can include assignment subexpressions and conditional subexpressions. Identifiers appearing in Boolean expressions need not be type Boolean; type conversions occur if necessary. DG/L treats numeric values as Boolean FALSE if they are equal to 0 and as TRUE otherwise. A Boolean value is represented internally as a single-precision integer, with a value of 0 or 1. When more than one operator is used in composing Boolean expressions, the rules of precedence govern the order in which operator-operand combinations are evaluated.

The simplest Boolean expression is one of the Boolean constants TRUE or FALSE. You can assign the Boolean constant values TRUE and FALSE to Boolean variables, as in:

```
BOOL := TRUE;
```

There are six Boolean operators:

AND	logical and	expr1 AND expr2
EQV	equivalence	expr1 EQV expr2
IMP	implication	expr1 IMP expr2
NOT	unary not	NOT expr
OR	logical or	expr1 OR expr2
XOR	exclusive or	expr1 XOR expr2

Boolean Operators on Boolean Operands

Table 3-1 describes the operations of the Boolean operators on Boolean operands.

Table 3-1. Boolean Operations or Boolean Operands

Operand Values		Operators					
A	B	A AND B	A EQV B	A IMP B	A OR B	A XOR B	NOT A
F	F	F	T	T	F	F	T
F	T	F	F	T	T	T	T
T	F	F	F	F	T	T	F
T	T	T	T	T	T	F	F

For example,

```
BOOL1 AND (BOOL2 XOR BOOL3)
```

evaluates to the implied Boolean result, taking the exclusive OR of BOOL2 and BOOL3, and then taking the AND of the result and BOOL1. If BOOL2 and BOOL3 are both TRUE, the XOR of BOOL2 and BOOL3 is FALSE. If BOOL1 is TRUE, the expression evaluates as FALSE, since the AND of TRUE and FALSE is FALSE.

You can use the Boolean operators on non-Boolean operands to perform bit comparisons. DG/L carries out all comparisons bit-by-bit on the internal binary representations of the values that it tests.

Boolean Operators on Integer and Pointer Operands

When applied to integer, double-precision integer, or pointer operands, a Boolean operator such as

```
var := expr1 AND expr2
```

sets a bit position in var to 1 only if both expr1 and expr2 have ones in their corresponding bit positions. This operator is often used for bit masking. For example, if SINT is a single-precision integer,

```
SINT AND 377R8
```

selects SINT's lower byte, since 377R8 (that is, 377_8) represents the 16-bit word

```
0000000011111111
```

When you take the AND of 377_8 and any 16-bit quantity, the result is an all-zero high byte.

var := expr1 EQV expr2

sets a bit position in var to 1 only if expr1 and expr2 have the same value in their corresponding bit positions.

var := expr1 IMP expr2

sets a bit position in var to 1 if expr1 has a 0 in that position or if expr2 has a 1 in that position.

var := NOT expr

sets each bit position in var to the complement of the bit in the corresponding position in expr. The NOT operator complements each bit of an integer operand. This is not the same as the unary minus operator applied to an integer. For example, the single-precision representation of the integer 4 is

0000000000000100

Its NOT is

1111111111111011

but its unary minus (a two's complement) is

1111111111111100

var := expr1 OR expr2

sets a bit position in var to 1 if either expr1 or expr2 has a 1 in its corresponding position.

var := expr1 XOR expr2

sets a bit position in var to 1 if the values in the corresponding positions in expr1 and expr2 are opposite.

Relational Operators

Relational operators let you evaluate relationships between expressions. They yield Boolean results. DG/L's relational operators are

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

The relational operators yield the value TRUE if their conditions are true, and the value FALSE otherwise. Some examples of Boolean expressions with relational operators are

A > B

which is TRUE if A is greater than B, and FALSE otherwise,

(W < X) AND (W < Y)

which evaluates the two relations, "W is less than X" and "W is less than Y", and then takes the AND of their Boolean results,

BOO := NOT BOO

which logically inverts the Boolean value BOO, and yields the inverted value as its result.

For numeric values, the relational comparisons follow the usual rules of mathematics. When two equal-length strings are compared, the string with a lower-valued ASCII code representation is considered the lesser. For example,

"ABC" < "ABD"

is true. If two strings with unequal lengths are compared, and if they differ within the length of the shorter string, the above rule applies. If the strings are the same within the length of the shorter string, the longer string is considered greater. For example,

"DEF" < "DEFAAA"

is TRUE,

"AB" < "ZZZZ"

is TRUE, since the two strings differ within the length of the shorter string and the left-hand string occurs earlier in the alphabet.

"ABC" = "DEF"

is FALSE, but

"ABC" < "ABD"

is TRUE, since the ASCII representation of the character D is greater than the ASCII representation of the character C.

Boolean expressions can contain conditional subexpressions that evaluate to Boolean values.

IF (K < 1) THEN (S > W) ELSE (H < C)

evaluates to the Boolean value of the expression S > W if K is less than 1, and the Boolean value of the expression H < C otherwise.

The expression

IF IF A THEN B ELSE C THEN D ELSE E

evaluates to the same Boolean value as

IF (IF A THEN B ELSE C) THEN D ELSE E

Pointer Expressions

A pointer expression represents the equivalent of a DG/L variable, and includes an arithmetic expression, the pointer operator ->, and a based variable. The general format of a pointer expression is

arith-expr -> based-var

Throughout the samples in this section, assume the declarations

```
POINTER P;
BASED INTEGER BI;
```

The simple pointer expression

P -> BI

refers to the word whose address is contained in P as a (single-precision) integer.

The arithmetic expression, which can itself include pointer expressions, evaluates to the memory address of a DG/L datum. As a simple case, the arithmetic expression may be a numeric constant.

```
100 -> BI
```

refers to memory location 100 as a single-precision integer. The based variable serves as a descriptor of the datum's type and precision. If you declare a BASED variable, the declaration does not reserve memory space; the based identifier serves as a descriptor or template. If the based variable is an array, you can subscript it in the same fashion as a conventional array.

If you were to assign

```
(P -> BI) := 1.5
```

the word that P points to would receive the integer value 1. (Since the BASED integer template BI describes the memory address as containing a single-precision integer, a real-to-integer type conversion is performed, truncating the fractional part of 1.5).

The pointer expression as a whole simulates a DG/L variable having the same attributes as the based variable of the expression. By using DG/L pointer expressions, you can specify any address in memory, and manipulate the data at that address as any one of DG/L's data types. Keep in mind that since pointers can refer to and modify any location in memory, you may change words in the object program or the runtime environment, causing hard-to-detect errors.

Pointer expressions can occur on either or both sides of an assignment symbol.

```
I := (P -> BI);
```

sets variable I to the value described by the pointer expression P -> BI. DG/L performs any type conversion necessary to match I's data type.

If you declare an integer array

```
INTEGER ARRAY I[1:10]
```

then the statement

```
I[1] := 666;
```

performs the same operation as

```
P := ADDRESS (I[1]);
(P -> BI) := 666;
```

The ADDRESS function is described in Chapter 8; it returns the memory address of an identifier.

Pointer expressions commonly include offsets, as well as pointer variables.

```
(P + 1) -> BI
```

adds 1 to the contents of P, and treats the sum as the address of a data element. In this case, BI calls for treating the data element as an integer. Whenever a pointer expression contains an offset, you must enclose the arithmetic expression in parentheses, since the pointer operator has a higher precedence than any arithmetic operator except exponentiation. Parentheses are optional if no offset is present. For example, the statements

```
(P -> BI) := ((P + 1) -> BI);
(P + 1) -> BI := ((P + 2) -> BI);
(P + 2) -> BI := (P -> BI);
```

interchange the contents of memory locations P + 1 and P + 2, using location P for temporary storage. You can use the same based variable descriptor to refer to any number of data elements.

The arith-expr describes the starting address of the datum, no matter how long the datum is.

Pointer expressions may themselves contain pointer expressions. This facilitates definition of complex addressing modes and data structures, as is described in Chapter 10. As a simple case, if you had declared BASED POINTER BP with other declarations as in the previous examples,

```
(P -> BP) -> BI
```

would refer to the integer whose address is contained in the word pointed to by P.

Pointer expressions, like other expressions, may incorporate conditional and assignment subexpressions.

```
(P + IF (X < 0) THEN 1 ELSE 2) -> BI
```

selects a pointer expression's offset based on a conditional clause.

```
(P := P + 1) -> BI
```

incorporates an assignment subexpression, respecifying P to the value P+1. This statement increments an address before fetching a datum.

Figure 3-1, which contains a program simulating formal data structures, illustrates the use of pointers.

```
 1 BEGIN
 2
 3 % A sample program to show how to approximate formal data structures using
 4 % DG/L's Literal, Pointer and Based variable facilities.
 5
 6 % Declare based variables for memory templates.
 7
 8   BASED INTEGER INTEGER_1;
 9   BASED INTEGER (2) INTEGER_2;
10   BASED STRING (SYMBOL_SIZE) STRING_SYM;
11
12 % Paramaterize size of memory allocated for SYMBOL "structure."
13
14   LITERAL $SYM_SIZE ( 128 );
15
16 % Declare elements of SYMBOL.
17
18   LITERAL SYMBOL_SIZE (MEMORY_POINTER -> INTEGER_1);
19   LITERAL SYMBOL_TYPE ((MEMORY_POINTER + 1) -> INTEGER_1);
20   LITERAL SYMBOL_VALUE ((MEMORY_POINTER + 2) -> INTEGER_2);
21   LITERAL SYMBOL_NAME ((MEMORY_POINTER + 4) -> STRING_SYM);
22
23 PROCEDURE WRITE_SYMBOL (SYMBOL_POINTER);
24   POINTER SYMBOL_POINTER;
25 BEGIN
26   FORMAT (1,
27     "Symbol size:  #<NL>Symbol type:  #<NL>Symbol value:  #<NL>Symbol"!!
28     " name:  #<NL>", SYMBOL_SIZE, SYMBOL_TYPE, SYMBOL_VALUE, SYMBOL_NAME);
29 END of procedure WRITE_SYMBOL;
30
31 PROCEDURE ENTER_SYMBOL (SYMBOL_POINTER, NAME, TYPE, VAL);
32   POINTER SYMBOL_POINTER; STRING NAME; INTEGER TYPE; INTEGER (2) VAL;
33 BEGIN
34   SYMBOL_SIZE := LENGTH (NAME);
35   SYMBOL_TYPE := TYPE;
36   SYMBOL_VALUE := VAL;
37   SYMBOL_NAME := NAME;
38 END of procedure ENTER_SYMBOL;
39
40 % Declare variables for main program.
41
42   POINTER MEMORY_POINTER;
43   STRING INPUT_NAME;
44   INTEGER INPUT_VAL;
45
46   EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
47
```

Figure 3-1. Pointer Expressions (continues)

```
48 % Main program starts here.
49
50   OPEN (0, (GETCINPUT));
51   OPEN (1, (GETCOUTPUT));
52   ALLOCATE (MEMORY_POINTER, $SYM_SIZE);
53   DO
54     BEGIN
55       WRITE (1, "<NL>Please enter a name:  ");
56       READ  (0, INPUT_NAME);
57       WRITE (1, "<NL>Please enter a value:  ");
58       READ  (0, INPUT_VAL);
59       ENTER_SYMBOL (MEMORY_POINTER, INPUT_NAME, 1, INPUT_VAL);
60       WRITE_SYMBOL (MEMORY_POINTER);
61     END
62       UNTIL INPUT_NAME = "DONE";
63 END;
```

Figure 3-1. Pointer Expressions (concluded)

The main program begins on line 50. Lines 50 and 51 open channels for input and output to the terminal, using the built-in external procedures declared on line 46. Line 52 reserves in memory 128 words, the resolution of the literal $SYM_SIZE, declared on line 14, beginning at the address given by MEMORY_POINTER, which line 42 declares as a pointer for the main program. Lines 53 through 61 comprise a DO loop controlled by the UNTIL clause on line 62. The DO loop consists of a Compound statement that takes data from the terminal and calls the two procedures ENTER_SYMBOL and WRITE_SYMBOL. Line 56 enters the first input in the string INPUT_NAME, declared on line 43. Line 58 enters the second input in the integer variable INPUT_VAL, declared on line 44. Line 59 calls the procedure with the arguments MEMORY-POINTER, INPUT_NAME, 1, and INPUT_VAL for the formal parameters SYMBOL_POINTER, NAME, TYPE, and VAL. The procedure, whose declaration begins on line 31, declares the formal parameters on line 32. The Compound statement that follows the declarations on lines 33 through 38 assigns to four literals the values LENGTH of NAME, TYPE, VAL, and NAME. At compilation time, the literals expand to the four pointer expressions on lines 18 through 21, with offsets to the pointer MEMORY_POINTER of no, one, two, and four words. Each pointer expression contains a BASED data type, as declared on lines 8 through 10: two INTEGER (1), one INTEGER (2), and one STRING variable. So, for example, the value entered at the terminal for INPUT_NAME is assigned to SYMBOL_NAME, and its length to SYMBOL_SIZE. The entry is stored at offset 4 to MEMORY_POINTER as data of STRING type with a maximum length of SYMBOL_SIZE, its actual length.

Control passes at the end of the procedure to line 60, still in the DO loop, which calls the WRITE_SYMBOL procedure on line 23. MEMORY_POINTER substitutes for the formal parameter SYMBOL_POINTER. This procedure prints out at the terminal a representation of the data structure.

Control then returns to the beginning of the DO loop. The DO loop continues to execute until the user enters DONE for the symbol name, as line 62 specifies. Line 63 terminates the main program.

String Expressions

A string expression evaluates to a single string result. String expressions are commonly made up of string variables and constants, substring clauses, conditional string expressions, and arithmetic and relational operations with string operands, although nonstring elements will be type-converted to strings if used in string expressions.

You can declare a string quantity as a string, or as a literal with a quoted value, or as a substring selected with the SUBSTR function.

A call to the SUBSTR function takes the form

```
SUBSTR (string, start-offset, [end-offset])
```

SUBSTR selects substrings. You can use it on either or both sides of an assignment symbol.

The string concatenation operator !! concatenates two strings.

```
"OS" !! SUBSTR ("BAYC",2,3)
```

evaluates to the string "OSAY".

String expressions may use relational operators that perform comparisons and assign Boolean truth values based on the result. For example,

```
"A" > "B"
```

evaluates to FALSE, and

```
"1" <= "2"
```

evaluates to TRUE.

The following example, program PIG, illustrates string expressions using a process that you may recall from your childhood; conversion of English words into "Pig Latin". The rules for Pig Latin are fairly simple. If a word begins with a vowel, its Pig Latin equivalent appends a "yay" to the word's end. If the word begins with consonants, all consonants preceding the first vowel are moved to the end of the word, and "ay" is appended to the result. For example, "they" becomes "eythay". Program PIG converts a single word typed in from the terminal to its Pig Latin representation and prints it.

```
1 BEGIN
2 EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
3 STRING INSTR;
4 LITERAL VOWELS ("AEIOU");
5
6 STRING PROCEDURE PIGLAT (IN);
7 STRING IN;
8
9 IF 0 <> INDEX (VOWELS, SUBSTR (IN,1))
10 THEN PIGLAT := IN !! "YAY"
11 ELSE BEGIN
12 MOVE:  IN := SUBSTR (IN, 2, LENGTH (IN)) !! SUBSTR (IN, 1);
13 IF 0 = INDEX (VOWELS, SUBSTR (IN,1)) THEN GOTO MOVE;
14 PIGLAT := IN !! "AY";
15 END OF PROCEDURE PIGLAT;
16
17 OPEN (0, (GETCINPUT));
18 OPEN (1, (GETCOUTPUT));
19
20 WRITE (1, "ENTER WORD FOR TRANSLATION:  ");
21 READ (0, INSTR);
22 WRITE (1, "ATTHAY ORDWAY ISYAY:  ", PIGLAT (INSTR), "<NL>");
23
24 END OF PROGRAM PIG;
```

Figure 3-2. String Expressions

Lines 1-4 head the outermost block of PIG. Line 2 accesses runtime procedures GETCINPUT and GETCOUTPUT to determine terminal input and output filenames. Line 3 declares INSTR, which will be used to store the input string, as a string variable. Line 4 declares and defines the literal VOWELS, which will be used to determine whether a character is a vowel or a consonant.

Line 6 begins the declaration of PIGLAT, a string procedure. Given any valid English word, the procedure will return a string result, the string's Pig Latin equivalent. (If PIGLAT receives a string containing no vowels, it will loop indefinitely.) Line 9 uses the INDEX function (described in Chapter 9) to determine whether the first character in the variable IN is a vowel. If it is, INDEX's result is nonzero, and line 10 (the THEN clause) executes, concatenating "yay" to the end of IN, assigning the result as the result of PIGLAT, and then exiting the procedure.

If INDEX returns a zero result, indicating that it did not find the first character of IN in VOWELS (so if it is a letter, it is a consonant), you must move consonants from IN's beginning to its end. Line 12 moves a single character, and line 13 checks to see if the new first character is a vowel. If it is not, line 13 branches back to line 12. If the new first character is a vowel, the consonant shift terminates, and execution continues at line 14, concatenating an "ay" to the end of IN, assigning that result as PIGLAT's result, and then exiting from the procedure. Line 15 returns to the calling program.

In the main, calling program, Lines 17 and 18 open the terminal for input and output, and lines 20 and 21 prompt for and receive the input string. Line 22 calls PIGLAT to convert this string, and prints the result. Line 24 ends program PIG.

String Arithmetic

DG/L provides a comprehensive string arithmetic capability. If you use the operators addition, subtraction, multiplication, division, or negation between two string expressions, the string operands are evaluated as decimal numbers, with no type conversion to numeric data types. The arithmetic operation treats the digits in the string character by character, and yields as many digits of precision as the digits in the string permit. No truncation or rounding occurs to match types if the data types represented by the strings are mixed.

"9" + "9.5"

yields the string "18.5". String arithmetic is employed only when both operands are strings.

9 + "7.1"

converts the string to type INTEGER, and uses integer arithmetic, returning an integer result, and truncating the fractional part of 7.1. All numbers used in string arithmetic must be decimal, and no radix specification field is permitted. The program returns the result as a string.

Figure 3-3 illustrates the use of string arithmetic.

```
1       BEGIN STRING (256) INPUT;
2
3               EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
4
5               STRING PROCEDURE EXECUTE;
6
7                   BEGIN STRING DIR, NUM1, NUM2;
8                   INTEGER POSN;
9                   LITERAL DIGITS ("0123456789");
10
11                      STRING (1) PROCEDURE GCHAR;
12
13                       BEGIN
14                       POSN := POSN + 1;
15                       IF POSN <= LENGTH (INPUT) THEN
16                       GCHAR := SUBSTR (INPUT, POSN)
17                       ELSE
18                             GCHAR := " ";
19                       END OF GCHAR PROCEDURE;
20
21                      STRING PROCEDURE GETDIR;
22
23                       GETDIR := IF 0 <> INDEX (INPUT, "ADD") THEN "ADD"
24                       ELSE IF 0 <> INDEX (INPUT, "MUL") THEN "MUL"
25                       ELSE "";
26
27                      PROCEDURE GETNUMS (NUM1, NUM2);
28                       STRING NUM1, NUM2;
29
```

Figure 3-3. String Arithmetic Program (continues)

```
30                      BEGIN STRING TEMP1, TEMP2;
31
32                      TEMP1 := TEMP2 := "";
33                      UNTIL 0 < > INDEX (DIGITS, TEMP1 := GCHAR) DO;
34                      UNTIL 0 = INDEX (DIGITS, TEMP2 := GCHAR) DO
35                      TEMP1 := TEMP1 !! TEMP2;
36                      NUM1 := TEMP1;
37
38                      UNTIL 0 < > INDEX (DIGITS, TEMP1 := GCHAR) DO;
39                      UNTIL 0 = INDEX (DIGITS, TEMP2 := GCHAR) DO
40                      TEMP1 := TEMP1 !! TEMP2;
41                      NUM2 := TEMP1;
42                      END OF GETNUMS PROCEDURE;
43
44              POSN := 0;
45              DIR := GETDIR;
46              IF 0 = LENGTH (DIR) THEN GO TO EXIT;
47              GETNUMS (NUM1, NUM2);
48              EXECUTE := IF (DIR = "ADD") THEN NUM1 + NUM2
49               ELSE NUM1 * NUM2;
50
51      EXIT:   END OF EXECUTE PROCEDURE;
52
53      OPEN (0, (GETCOUTPUT));
54      OPEN (1, (GETCINPUT));
55      DO BEGIN
56              WRITE(0, "YOUR WISH, SIRE ?");
57              READ (1, INPUT,NO_MORE);
58              WRITE (0, (EXECUTE), "<NL>");
59              END;
60
61      NO_MORE:
62
63      END OF ENCLOSING PROGRAM;
```

Figure 3-3. String Arithmetic Program (concluded)

The program accepts an input line of the form

ADD 396 1772

or

MUL 26279 59888881

and uses the string arithmetic package to get a result to an arbitrarily large precision.

The program begins at line 1 by declaring a 256-character string, INPUT, which will hold the input line. Line 3 declares GETCINPUT and GETCOUTPUT as external procedures; the runtime library provides these procedures.

Most of the program's operations are described within STRING PROCEDURE EXECUTE, whose declaration begins at line 5. EXECUTE returns the string arithmetic result of the operation requested. Line 7 declares three string variables: it uses DIR to store the command directive, and NUM1 and NUM2 to hold the two operands. Line 8 declares POSN, an integer, which will store an index in the input line as the line is processed. Line 9 declares LITERAL DIGITS to be used in determining whether a given input character is a numeric digit.

Line 11 begins the declaration of procedure GCHAR, which returns successive characters from the input line. Line 13 increments POSN, and then returns either the "POSN-th" character in INPUT or an empty string (if the input string is not as long as the current value of POSN).

Line 21 begins the declaration of STRING PROCEDURE GETDIR, whose function is to return the command directive. If the string "ADD" is in INPUT, GETDIR takes the string "ADD"; if "MUL" is in INPUT, GETDIR takes the string "MUL"; if INPUT contains neither, GETDIR takes the empty string.

Line 27 begins the declaration of procedure GETNUMS, which returns the two operands through parameters NUM1 and NUM2. Line 28 declares the parameter types, and line 30 declares two local string identifiers. Line 32 initializes both TEMP1 and TEMP2 to empty strings. Line 33, a DO statement, calls GCHAR for successive characters until the program finds a numeric digit. Once it finds a digit, lines 34 and 35 concatenate successive digits onto TEMP1, and when a nondigit is found, control passes to line 36, which assigns the result to NUM1. An identical process, extending from line 38 to line 41, collects the digits of the second operand and assigns the result to NUM2. Line 42 ends the GETNUMS procedure.

Line 44 begins the statement part of the EXECUTE procedure, by initializing POSN to 0. Line 45 assigns the result of the GETDIR procedure to DIR. If the length of DIR is 0, meaning that no directive was recognized, line 46 branches to exit from the procedure. Line 47 calls GETNUMS to determine the two operands. Lines 48 and 49 use string arithmetic to either add or multiply the two operands, depending on the directive. Line 51 ends the EXECUTE procedure.

The main program is simple. Lines 53 and 54 use OPEN, GETCINPUT, and GETCOUTPUT to open the terminal input and output files. The DO block extending from line 55 to line 59 requests input, processes it, and returns results until it receives an empty input line, at which point the program branches to NO_MORE and terminates.

Bit Expressions

This data type represents a string of bits. You may think of them as signed integers of arbitrary precision.

Bit Operations

Bit strings take various operations. These include

- concatenation and the operations SUBSTR, INDEX, LENGTH, and BYTE
- bit operations TBIT, SBIT, and CBIT (see the *DG/L™ Runtime Library (AOS and AOS/VS) User's Manual)*
- Boolean operations AND, OR, NOT, XOR, EQV, and IMP
- arithmetic operations +, -, *, and /

If you perform the Boolean operations on strings of different lengths, the operation right-justifies the shorter bit string in a field to the same size as the longer, sign extended, so that the precision will be that of the longer string. In sign extend, the leftmost bit of the shorter bit is duplicated in all leading bits up to the length of the longer. This does not actually modify the operands.

Because bit operations treat the bit strings as signed, two's complement binary numbers with the leftmost bit (SUBSTR (B, 1)) as the sign bit, you can use them to perform integer arithmetic at precisions greater than INTEGER or INTEGER (2) allow. The operand and resulting bit strings must have sufficient precision to contain the values.

The assignment operator treats the resulting bit string as a signed two's complement number and compresses it to its minimum representation. It deletes all duplicate leading bits to leave only one leading sign bit before it performs the assignment operation. To prevent this deletion, use SUBSTR on the left side of the assignment. SUBSTR sign-extends the input to the full precision of the result field if the input is shorter than the field, and prevents compression if the input is the same length or longer.

```
SUBSTR (B, 1, SIZE (B)) := B1 + B2;
```

Bit Conversions

The only legal conversions are between BIT strings and STRING, BOOLEAN, INTEGER, INTEGER (2), and POINTER.

Strings convert character by character from left to right, creating a 1 bit for an ASCII "1" and a 0 bit for an ASCII "0". The conversion from BIT to STRING follows the reverse procedure. If you assign a STRING to BIT as in

```
BITVAR := "000011";
```

the conversion results in a 6-bit string, but the assignment compresses this to 011. If instead you use

```
SUBSTR (BITVAR, 1, 6) := "000011";
```

you retain BITVAR as a 6-bit string.

In a conversion from BOOLEAN to BIT, the BOOLEAN value is tested for TRUE or FALSE and the BIT string is set to "1" or "0" respectively. The conversion from BIT to BOOLEAN tests the first bit (SUBSTR (B, 1)) and sets the BOOLEAN variable to a TRUE for "1" or FALSE for "0".

In the conversion from BIT to INTEGER, INTEGER (2), or POINTER, the operator treats the BIT string as a signed binary number. The operator converts BIT to one of these types by sign-extending the leftmost bit to fill 16 (or 32 for INTEGER (2)) bits and by putting them into the number. For example, it converts "10" to INTEGER by first sign-extending it to "1111111111111110" and then puts it into INTEGER, where it represents the value -2. If the BIT string is longer, the resultant data type will truncate and the RIGHTMOST 16 (or 32) bits will be assigned to the variable. For the opposite conversions from these types to BIT, the operator treats the value as a 16-bit (32-bit for INTEGER (2)) string matching the bit pattern in the variable in memory. The assignment compresses duplicate sign bits to give the shortest bit string that represents the desired signed number. For example, 2 converts to "010" and -4 converts to "100".

If you move a bit string value into a BIT string for which it is too long, truncation takes place on the LEFT end, and not on the RIGHT as would happen for character strings. Truncation might result from assignment to a BIT variable of small maximum length, or assignment into a SUBSTR shorter than the BIT string assigned.

Precedence

DG/L follows strict rules of operator precedence in evaluating the results of expressions. The list shows this order, from first-evaluated (^) to last (:=).

1. ^
2. ->
3. !!
4. unary +, unary -, NOT
5. *, /
6. +, -
7. >, >=, =>, =, <>, <=, =<, <
8. AND
9. OR, XOR
10. EQV, IMP
11. :=

If more than one operator with the same precedence occurs in an expression, DG/L evaluates the results from left to right, except for assignments, which it performs from right to left. You can control the order of result evaluation by using parentheses; parenthesized subexpressions are evaluated before their surrounding context. When DG/L evaluates an expression, the precedence of its operators and its parenthesization determine the order of result evaluation. Effectively, a series of subexpressions are evaluated, which are eventually evaluated to a single value. For example,

```
4 * 3 + 5
```

evaluates to 17, since the multiplication is performed before the addition. The order is reversed in

```
4 * (3 + 5)
```

which performs the parenthesized addition first and evaluates to 32.

Certain hardware considerations occasionally dictate that you use parentheses to direct the order of expression evaluation. For example,

```
(A*B) / (C*D)
```

and

```
(A/C) * (B/D)
```

are arithmetically identical. If, however, the variables have large values, on the order of 10E40, the first expression can cause an overflow error, but the second cannot.

If parentheses are nested, DG/L evaluates the innermost parenthesized expressions first, and follows with their successively enclosing expressions. For example,

```
(TRUE AND (TRUE OR (TRUE IMP FALSE)))
```

evaluates to TRUE. DG/L first evaluates the expression TRUE IMP FALSE as FALSE, with the result

```
(TRUE AND (TRUE OR FALSE))
```

It then evaluates TRUE OR FALSE as TRUE, leaving

```
(TRUE AND TRUE)
```

which is certainly TRUE. However,

```
TRUE AND TRUE OR TRUE IMP FALSE
```

evaluates to FALSE. First, since there are no parentheses, DG/L evaluates the highest-precedence operator. TRUE AND TRUE is TRUE, leaving

```
TRUE OR TRUE IMP FALSE
```

Since OR has a higher precedence than IMP, DG/L next evaluates TRUE OR TRUE as TRUE. This leaves

```
TRUE IMP FALSE
```

which is clearly FALSE.

Data Type Conversion

The DG/L language lets you manipulate expressions whose variables and constants are of different data types. In many cases, the DG/L runtime environment carries out automatic type conversions on the data in order to evaluate the expression T1 Op T2 = T3. In order to know what the resultant type (T3) of an expression will be, you must know the data type of the first operand (T1) and the second operand (T2). In addition, you must know whether or not the operator (Op) is an exponentiator. Tables 3-2 and 3-3 specify the resultant data type, given the two operand data types. Table 3-3 covers exponentiation, and Table 3-2 covers all other operators. To determine the resultant data type, find the intersection of the row for the first operand and the column for the second operand. The entry "Error" means that type conversion does not take place and you get an error message instead. The tables are symmetric about the diagonal. The terms Integer (1), Integer (2), Real (2), Real (4) have the same meanings as in Chapter 2: single- and double-precision integer, single- and double-precision real, respectively.

Table 3-2. General Resultant Data Types

Operand 1 Data Type	Operand 2 Data Type							
	Int (1)	**Int (2)**	**Real (2)**	**Real (4)**	**Boolean**	**String**	**Pointer**	**Bit**
Int (1)	Int (1)	Int (2)	Real (2)	Real (4)	Boolean	Int (1)	Int (1)	Bit
Int (2)	Int (2)	Int (2)	Real (2)	Real (4)	Boolean	Int (2)	Int (2)	Bit
Real (2)	Real (2)	Real (2)	Real (2)	Real (4)	Error	Real (2)	Real (2)	Real (2)
Real (4)	Real (4)	Real (4)	Real (4)	Real (4)	Error	Real (4)	Real (4)	Real (4)
Boolean	Boolean	Boolean	Error	Error	Boolean	Boolean	Pointer	Bit
String	Int (1)	Int (2)	Real (2)	Real (4)	Boolean	String	Pointer	Bit
Pointer	Int (1)	Int (2)	Real (2)	Real (4)	Pointer	Pointer	Pointer	Pointer
Bit	Bit	Bit	Real (2)	Real (4)	Bit	Bit	Pointer	Bit

Table 3-3. Exponentiation Resultant Data Types

Base Data Type	Exponent Data Type							
	Int (1)	**Int (2)**	**Real (2)**	**Real (4)**	**Boolean**	**String**	**Pointer**	**Bit**
Int (1)	Int (1)	Real (4)	Real (2)	Real (4)	Error	Error	Int (1)	Error
Int (2)	Real (4)	Real (4)	Real (4)	Real (4)	Error	Error	Real (4)	Error
Real (2)	Real (2)	Real (4)	Real (4)	Real (4)	Error	Error	Real (2)	Error
Real (4)	Real (4)	Real (4)	Real (4)	Real (4)	Error	Error	Real (4)	Error
Boolean	Error	Error	Error	Error	Error	Error	Error	Error
String	Error	Error	Error	Error	Error	Error	Error	Error
Pointer	Int (1)	Real (4)	Real (2)	Real (4)	Error	Error	Int (1)	Error
Bit	Error	Error	Error	Error	Error	Error	Error	Error

Designational Expressions

Designational expressions are formulas evaluating to DG/L statement labels. They may be simple integer labels, identifier labels, switch references, subscripted label references, or conditional designational expressions.

A switch reference takes the form

```
sw-id [ arith-expr ]
```

where **sw-id** is a declared switch, and **arith-expr** evaluates to a positive integer. In this case, the brackets around **arith-expr** are not meant to indicate that it is optional, but that it should be enclosed in brackets. For example

```
SWIVAR[A+2]
```

where SWIVAR has been declared as a switch (Chapter 5).

A subscripted label reference takes the form

```
sublab-id [ arith-expr ]
```

where **sublab-id** is the name of a subscripted label, and **arith-expr** evaluates to any integer. As with switch references, the brackets around **arith-expr** are not meant to indicate that it is optional, but that **arith-expr** should be enclosed in brackets. For example

```
SLAB[1 + B * 4]
```

where SLAB is a subscripted label.

Assignment subexpressions may occur within designational expressions, but assignment of a value to a label identifier is illegal.

For both switches and subscripted labels, an arith-expr outside the range of defined subscripts has no effect, except for side effects occurring while evaluating the arith-expr, as would occur in the expression

```
SLAB[1 + (A := A + 2)]
```

This expression respecifies A. You cannot bypass the respecification for an undefined subscript, since the subscript value is not known until the assignment computes it.

A conditional designational expression takes the form

```
IF bool-expr THEN desig-expr1 ELSE desig-expr2
```

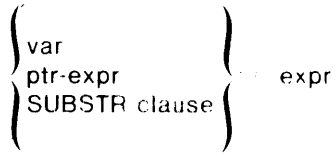
As with all conditional expressions both desig-exprs must be included. The value of the bool-expr selects the desig-expr that will be evaluated as the value of the designational expression; if bool-expr is TRUE, desig-expr1 will be selected, otherwise desig-expr2 will be selected. An example of a conditional designational expression is

```
IF DONE THEN EXITLAB ELSE SW[N + 1]
```

If this expression occurs in a GO TO, and DONE is TRUE, the program branches to EXITLAB. If DONE is FALSE, and if N + 1 is within the range of the switch or subscripted label SW, the program branches to switch or subscripted label SW[N + 1].

The Assignment Expression

You can embed assignment expressions, which perform the same actions as assignment statements, in any DG/L expression. The general format of assignment expressions is



Arbitrarily many assignments may occur within a single expression, but each assignment must be to one of the specified left part types, and not to any other type of expression. As with the assignment statement, and unlike other operators, DG/L performs multiple assignments from right to left. It carries out type conversions as needed. The expression

```
T := X := 3 + 4 * B
```

is legitimate, and assigns subexpression 3 + 4 * B's value to variables X and T. But

```
T := 3 + X := 3 + 4 * B
```

is illegal, since it attempts to respecify an expression, 3 + X, that is not a legal type for respecification. The assignment operator has lower precedence than any other operator. Therefore, DG/L performs the operation 3 + X before the assignment operation. The result of the subexpression 3 + X is an expression, not a variable, and so cannot be respecified. If you wish to include assignment subexpressions within a DG/L expression, you should generally parenthesize the subexpressions, as in

```
T := 3 + (X := 3) + 4 * B
```

In this expression, DG/L performs the assignment of 3 to X before it executes any other operations.

DG/L processes an expression occurring within a statement and not bypassed by a conditional clause before it executes the statement's action, so you could replace the two statements

```
C := C + 1;
GO TO SW[C];
```

with

```
GO TO SW[C := C + 1];
```

This performs the action of adding 1 to C, and then branches to SW with a subscript as specified by the new value of C.

Conditional Expressions

The DG/L conditional expression facility lets you select the value of an expression based on the value of a Boolean expression. It has the format

IF bool-expr THEN expr1 ELSE expr2

If necessary, DG/L converts data types as Table 3-2 specifies. The value of the expression is expr1 if the bool-expr is TRUE, and expr2 if the bool-expr is FALSE. A conditional expression must include both an expr1 and an expr2, so that DG/L can evaluate the expression to a definite value for both possible values of the Boolean expression. For example,

```
IF A > B THEN C ELSE D
```

evaluates to the value of C if A is greater than B, and to the value of D if A is not greater than B.

The expr1 and expr2 of a conditional expression may incorporate assignment expressions. In this case, only those assignments in the clause that the Boolean expression selects will execute. In

```
IF DOINC THEN VAL := VAL + 1
```

the value of VAL increments only if the Boolean value of DOINC is TRUE. Similarly, if A initially has the value 1.0 and B has the value 2.0, the expression

```
IF A > B THEN A := A/2 ELSE B := B/2
```

changes the value of B to 1.0, leaving A unchanged. DG/L first tests the relationship between A and B. Then, based on the result, one of the clauses A := A/2 or B := B/2 executes. In this example, the ELSE clause executes, respecifying B to 1.0, and this value becomes the result of the conditional expression.

End of Chapter

Chapter 4
Statements

DG/L programs have two main parts: the declaration part and the statement part. You build both parts from the syntactic units described in Chapters 2 and 3. Chapter 5 discusses declarations; this chapter discusses statements.

DG/L statements describe the actions that take place in a DG/L program.

- The Assignment statement sets the value of a variable.
- The IF statement carries out an action only if certain conditions are true. Its optional ELSE clause specifies an alternate action if the conditions are not true.
- The GOTO statement passes control of execution to a statement other than the one following it.
- The DO statement carries out an action repeatedly. You control the number of repetitions with appropriate WHILE, UNTIL, FOR, and STEP clauses.
- A BEGIN...END Compound statement lets you treat the sequence of statements that it contains as a single statement. If the Compound statement also contains declarations, it constitutes a block (see Chapter 7).
- A Procedure Call statement passes control of execution to a procedure declared previously in the same program (an internal procedure), or declared in a separate program module (an EXTERNAL procedure). See Chapter 6 for more on the structure and use of procedures.
- A Dummy statement may occur wherever any other type of statement occurs.
- A COMMENT statement contains remarks you wish to place between other statements.

You can nest statements within other statements indefinitely. Except for passing execution control with GOTO or Procedure Call statements, DG/L statements execute in the order in which you list them.

Space Delimiters and Statement Termination

Each DG/L statement must end with the semicolon (;) *terminator*. You must follow the final END statement of a DG/L program with a semicolon.

A semicolon is optional in two cases:

- after the BEGIN and before the END keyword of a BEGIN...END Compound statement
- before another semicolon

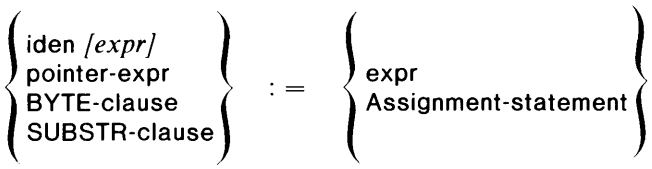
A semicolon must not appear within a statement in three cases:

- before the ELSE clause of an IF statement
- before a WHILE clause, following the statement it controls
- before an UNTIL clause, following the statement it controls

You must follow each keyword with a ; terminator (if it falls at the end of its statement), or with a space delimiter. You may count as a space the ASCII blank, the horizontal tab, a form feed, or a NEW LINE. You may insert more than one space for readability wherever you may insert a single space. With the exception of the alternate forms GOTO and GO TO, you must not insert a space within a keyword. None of the spaces acts as a terminator in DG/L programs.

The Assignment Statement

The *Assignment statement* assigns a value to a declared variable. The statement has the form



The assignment operator is the := symbol. You must distinguish it from the relational operator = symbol. You must not separate the two parts of the assignment operator with spaces.

The left part of the Assignment statement receives the value specified by the right side of the Assignment statement.

The left part of the Assignment statement specifies a memory location or locations, and may be a variable, a subscripted variable, a pointer expression, a substring specified with the SUBSTR function, or a byte specified with the BYTE or ASCII function (see Chapter 8).

The right part of the Assignment statement specifies the values to be stored, and may be any of the constructs permitted for the left part, a numeric, character string, or bit string constant value, or any other DG/L expression.

The data types of the left and right parts need not match. If they do not, the Assignment statement attempts to convert the right part to the data type of the left part.

Since an Assignment statement may occur as the right part of an Assignment statement, you may specify assignments to any depth. When one Assignment statement occurs within another, the assignments are carried out sequentially from right to left. The order in which you specify the assignments may lead to different results. For example, if you declared REA as real and INT as integer, the statement

```
INT := REA := 4.5;
```

assigns REA the value 4.5 and INT the value 4. If you reverse the order of assignments, however, as in

```
REA := INT := 4.5;
```

INT first gets the value 4 (since its type, integer, gets no fractional part), and REA then gets the real value 4.0, the result of converting the integer to a real.

Data Type Conversions in Assignment Statements

If the variable and the value you are assigning to it are of different types, the Assignment statement automatically converts the value on the right of the Assignment statement to the type of the variable on the left. To see which rule applies or what type results from an Assignment statement, follow the column in Table 4-1 which matches the type of your input value type down to its intersection with the row of the variable type.

Table 4-1. DG/L Assignment Data Conversions

Left Part Variable Assignment Type	Right Part Input Value Type							
	Int (1)	Int (2)	Real (2)	Real (4)	String	Pointer	Boolean	Bit
Int (1)	Int (1)	10	11	11	4	Int (1)	7	Int (1)
Int (2)	1	Int (2)	Int (2)	Int (2)	4	1	7	Int (2)
Real (2)	Real (2)	Real (2)	Real (2)	Real (2)	4	Real (2)	9	9
Real (4)	Real (4)	Real (4)	2	Real (4)	4	Real (4)	9	9
String	3	3	3	3	String	3	8	String
Pointer	Pointer	10	11	11	3	Pointer	7	Pointer
Boolean	6	6	9	9	5	Boolean	Boolean	Boolean
Bit	Bit	Bit	9	9	String	Pointer	Boolean	Bit

Numerical entries refer to the rules below.

RULES for data type conversion in Assignment statements:

1. The assignment operator (:=) converts single-precision integer or pointer values to double-precision form by appending a most-significant word to the left side of the existing value. The inclusion of the most-significant word changes neither the sign nor the value of the original data item.

2. The operator converts single-precision real values to double-precision real values by appending two all-zero least-significant words to the mantissa.

3. In converting numeric values to strings, the assignment operator follows the same pattern as the FORMAT routine (Chapter 9). The operator converts them as if you were going to print them with a "#" field character. For integer and real values, leading zeroes are removed.

When converting a	you get a string with
single-precision real	up to seven significant places (exclusive of exponent)
double-precision real	up to 16 significant places
single-precision integer	up to 5 significant places
double-precision integer	up to 10 significant places
pointer	an unsigned octal value, with leading zeroes

4. The operator converts string to integers or reals as follows: starting with the leftmost (first) character of the string, runtime code checks for numeric digits. If the first character is not a numeric digit or a sign character, the numeric value of the string is zero. If the first character is numeric, the code reads the string from left to right until it reaches a character that is not valid for numeric notation. For example, the operator converts the string "1.23E8XTRA CHARS" to the numerical value 1.23*10^8 and ignores the remaining string "XTRA CHARS". It then puts the converted result through numeric type conversion as needed. The operator converts to all types but pointer in base 10; it converts strings to pointers as base 8 quantities.

5. The operator converts a string to Boolean TRUE if the string begins with the uppercase character "T," and to Boolean FALSE otherwise.

6. The operator converts an integer value of 0 to Boolean FALSE, and all other integer values to Boolean TRUE.

7. The operator converts a Boolean TRUE to numeric 1, and Boolean FALSE to numeric 0.

8. Boolean TRUE converts to the string "TRUE" and Boolean FALSE converts to the string "FALSE". If the specified string is not long enough to store the full string "TRUE" or "FALSE", the specified string receives the single letters "T" or "F" instead. A string beginning with "T" converts to Boolean TRUE, and to Boolean FALSE otherwise.

9. Attempted Boolean or bit to real or real to Boolean or bit conversions result in an ILLEGAL TYPE CONVERSION error at compilation time.

10. The operator truncates the double-precision integer to single by removing the most-significant word, resulting in a value between -32768 and 32767.

11. The operator truncates double-precision reals to single-precision reals by changing from 16 digits of precision to 7 decimal digits. For further details, consult the diagram of internal representation in Chapter 5, "Declarations."

The sample program in Figure 4-1 shows several different Assignment Statements.

```
 1    BEGIN INTEGER J;
 2          REAL R;
 3          REAL ARRAY MAT [2];
 4          STRING CHARS;
 5
 6          R :=  7.3;
 7          CHARS := "7.3";
 8          R := 46500.3 + SIN (1.25);
 9          MAT [0] := 10.;
10          MAT [1] := 11.3;
11          MAT [2] := J := MAT [0] + SQRT (R);
12    END;
```

Figure 4-1. Assignment Statements

Lines 1 through 4 declare the variables that the program uses. Line 6 assigns the real value 7.3 to the real variable R. Since the left-part and right-part data types match, data type conversion does not occur. Line 7 assigns the character string "7.3" to the string variable CHARS. If you had declared the data type of CHARS as real, instead of string, the assignment statement would have converted the character string "7.3" to the real value 7.3. Line 8 computes the sine of 1.25 radians, adds it to 46500.3, and stores the result in the real variable R. Line 9 gives the value 10 to the first element (element 0) of the array MAT. Line 10 gives a value of 11.3 to element 1, the second element of the array. Line 11 first computes the square root of R; SQRT returns a real result. It then adds the result to the value of element 0 of MAT, converts the result of the addition to integer type, and assigns it to variable J in the smaller Assignment statement (with the rightmost assignment operator). The conversion to integer truncates any fractional part. The larger Assignment statement (with the leftmost assignment operator)

converts the value of J to real type and stores it in element 2 of MAT. As the example shows, it is possible to make several assignments on the same line, but the conversion effects can be significant: the real value of MAT[0]+SQRT (R) may differ from the real value of MAT[2].

The IF Statement

The *IF statement* specifies conditional execution of an action. The statement has the form

IF bool-expr THEN *[label]* st1 *[ELSE [label] st2]*;

The IF statement executes the statement in the THEN clause only when the expression in the IF clause has the Boolean value TRUE. If the condition in the IF clause has the Boolean value FALSE, the IF clause executes the statement in the ELSE clause, if there is one. Otherwise, control passes to the statement following the IF statement. In

```
IF X = 0 THEN X := 1 ELSE X := Y;
```

if X = 0 is true, the THEN clause assigns the value 1 to X, and control passes over the ELSE clause to the next statement. If X = 0 is false, control passes over the THEN clause to the ELSE clause. In the simpler statement

```
IF X = 0 THEN X := 1;
```

if X = 0 is true, the THEN clause assigns the value 1 to X, and control passes to the next statement. If X = 0 is false, control passes over the THEN clause to the next statement.

The statements in the THEN and ELSE clauses may be any valid DG/L statements, including other IF statements and BEGIN...END Compound statements (discussed later in this chapter), or a block (Chapter 7). The IF statement, in conjunction with Compound statements, lets you specify the conditional execution of long sequences of code. In

```
IF EVAL THEN BEGIN
      st1;
      .
      .
      .
      sti;
      END
ELSE BEGIN
      stj;
      .
      .
      .
      stn;
      END;
```

only statements 1 through i execute if EVAL is true, and only statements j through n execute if EVAL is false.

The optional ELSE clause is a part of the IF statement; you must not precede the ELSE keyword with a semicolon.

You may nest one IF statement within another. When you do, any ELSE belongs to the closest preceding IF not already matched with an ELSE, unless BEGIN...END pairs intervene. In

```
IF expr THEN
     IF expr THEN st1
       ELSE st2;
```

the ELSE belongs to the second IF. But in

```
IF expr THEN BEGIN
     .
     .
     .
     IF expr THEN st2
     END
ELSE st2;
```

the ELSE belongs to the first IF, which is not enclosed in the BEGIN...END pair. (Note that neither END nor ELSE follows a semicolon.)

You may transfer control, or branch, using a GOTO statement, from anywhere in the program to a labelled statement in a THEN or ELSE clause, even to one within a BEGIN...END Compound statement, as long as you follow the standard legal scoping rules (Chapter 7): you may branch from the current block or a block it contains, but not from a parallel block or a block enclosing the current block.

When you branch, evaluation of the Boolean expression in the IF clause does not take place. If you branch to a statement in the THEN clause, the labelled statement and statements up to the ELSE execute, and control then passes to the statement following the ELSE clause. If you branch to a statement in the ELSE clause, statements from the labelled point to the end of the IF statement execute, and control then passes to the next statement. The expression in the IF clause may be as simple or as complex as you like, so long as the statement can evaluate it to a Boolean value. The statement converts data type if necessary for the evaluation.

The sample program in Figure 4-2 illustrates the use of an IF statement.

```
 1    BEGIN STRING (132) INPUT_CHARS;
 2      EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
 3
 4      OPEN (0, (GETCINPUT));
 5      OPEN (1, (GETCOUTPUT));
 6      WRITE (0, "Please type in a character string:<NL>");
 7      READ (1, INPUT_CHARS);
 8      IF LENGTH (INPUT_CHARS) > 0
 9          THEN WRITE (0, "<NL>You wrote:  ", INPUT_CHARS)
10          ELSE WRITE (0, "<NL>Then I have nothing to say either.");
11    END;
```

Figure 4-2. IF Statement

Line 1 declares a string variable INPUT_CHARS with a maximum length of 132 characters. Lines 2 through 5 declare and open channels to and from the terminal. Line 6 asks for input and Line 7 stores the input in INPUT_CHARS. Lines 8 through 10 constitute the IF statement. LENGTH is a built-in function (Chapter 8) that returns the length of its string argument. If the user of the program typed in a character string, the length of INPUT_CHARS is greater than 0, and the Boolean expression in the IF clause is true. So Line 9 executes, displaying the user's input, and control passes over the ELSE clause to Line 11, which ends the program. If instead the executor typed in only a NEW LINE, the length of INPUT_CHARS is 0, and the Boolean expression is false. So control passes over the THEN clause to the ELSE clause on Line 10. The program sends its own message and the program again ends with Line 11.

The GOTO Statement

The *GOTO statement* transfers control to another statement. You can combine it with other statements and features for conditional or computed branching. The statement has the form

{GO TO / GOTO} label-expr;

You may put any number of blanks (or other spacing delimiters) between the GO and the TO, but you need not put any. TO is a keyword only when used with GO.

You may make label-expr a simple statement label, an integer, a subscripted label, or a switch as declared in a SWITCH declaration (Chapter 5).

A simple GOTO statement such as

```
GOTO LAB1;
```

makes an unconditional transfer to the statement labelled with LAB1. You may also embed a GOTO statement in an IF statement to set up conditional transfers. For example, the statement

```
IF CNT < 0 THEN GOTO LAB1
     ELSE GOTO EXIT;
```

branches to LAB1 if CNT < 0 is true, and branches to EXIT otherwise. You may embed an IF statement in a GOTO statement for the same effect. The statement

```
GOTO IF CNT < 0 THEN LAB1 ELSE EXIT;
```

is equivalent to the preceding one.

If you use a GOTO statement with an IF statement,

- the IF statement must include an ELSE clause to provide a transfer location for both possible values of the Boolean expression
- the expressions following the THEN and ELSE must evaluate to labels

You can define a loop in which the GOTO statement determines the number of times a BEGIN...END Compound statement executes. In the construction below, the statements divide X by 2 just MAX times.

```
BEGIN INTEGER COUNT, MAX, X;
      .
      .
      .
      COUNT := 0;
LAB1:   COUNT := COUNT+1;
      IF COUNT <= MAX THEN BEGIN
         X := X/2;
         GOTO LAB1;
         END;
END;
```

You may branch out of an IF or DO statement with a GOTO. If you do this, the execution of the IF or DO statement terminates and cannot resume. If you exit a DO loop with a GOTO statement before the DO loop terminates, the controlled variable for the DO loop maintains the last value it receives. You cannot use a GOTO statement to transfer to a point inside a DO statement, but you may use it to transfer to the beginning of the entire DO statement. (DO loops are described later in this chapter.)

GOTO statements are sensitive to block structure (described in Chapter 7). A GOTO statement can transfer to a label in any enclosing or dominant block or the current block. It cannot transfer to a label in a parallel or subordinate block, since such labels have no definition from the point of view of the GOTO statement. Figure 4-3 shows allowed and prohibited references.

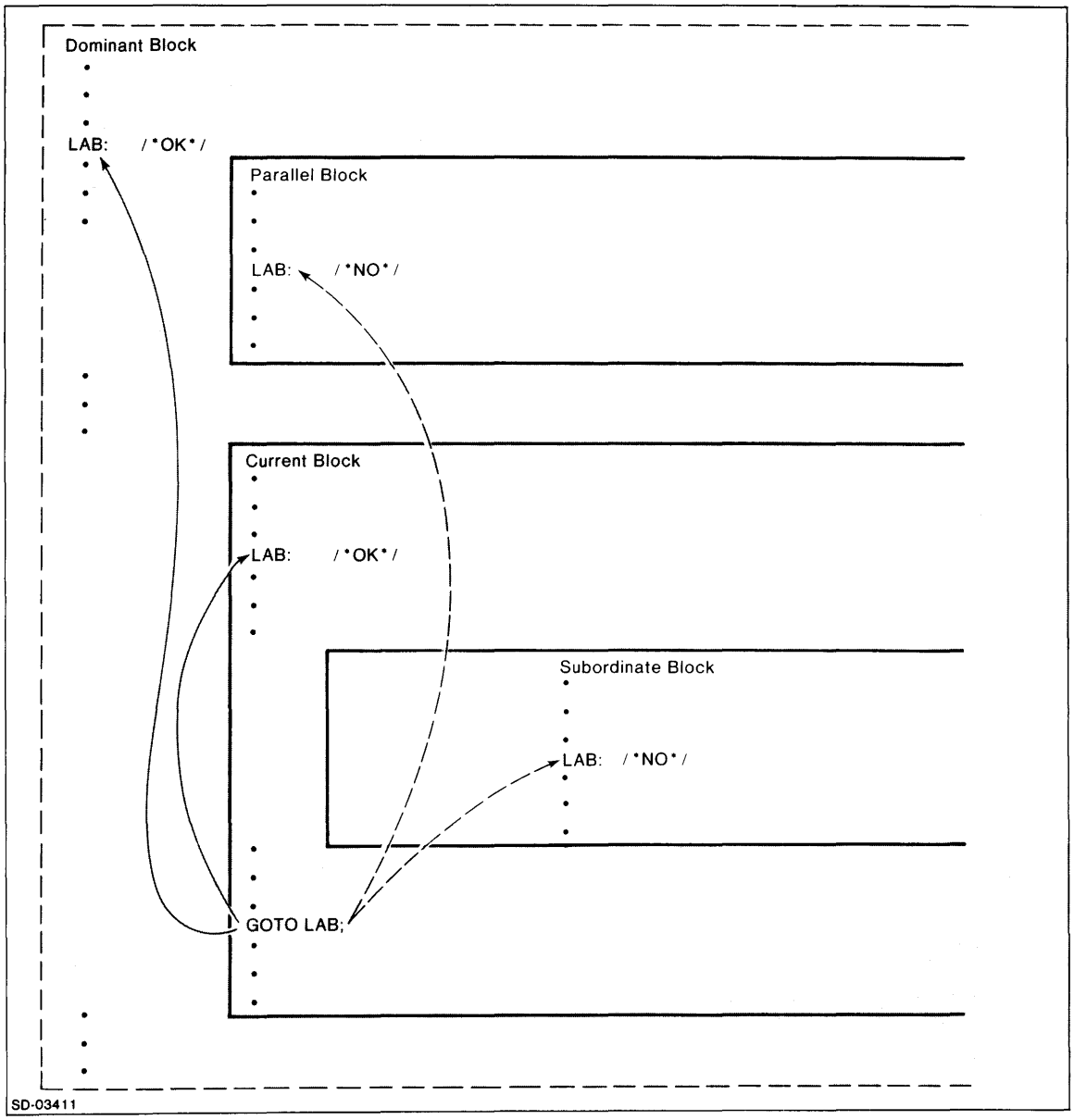


Figure 4-3. Use of GOTO Statements

You may use a GOTO statement to branch from a procedure, but to do so you must pass the label to the procedure as a formal parameter of type LABEL. Chapter 6 gives details.

You may also use a GOTO statement to branch to a computed point. For example, the statement

```
GOTO SW[N];
```

branches through the Nth expression in the list defining SWITCH SW (Chapter 5) or the subscripted label SW[N] (Chapter 2). If the GOTO statement mentions an undefined switch or subscript, control passes to the next statement. You may place a subscripted label anywhere in a statement part of a DG/L program.

Subscripted labels let you pass to a point selected by an integer value. The subscript can be positive or negative. Defined subscripts need not be consecutive. You could define SLAB[1] and SLAB[3], with no definition for SLAB[2]. In the program fragment

```
  BEGIN INTEGER INT;
        .
        .
        .
      GOTO SLAB[INT];
      GOTO EXIT;
        .
        .
        .
SLAB[1]:  statement;
        .
        .
        .
SLAB[2]:  statement;
        .
        .
SLAB[-1]:  statement;
        .
        .
        .
EXIT:  END;
```

if you reach the GOTO SLAB[INT] statement with INT less than -1, equal to 0, or greater than 2, control passes to the next statement and the block ends. Otherwise, control passes to the appropriate statement.

The DO Statement

The *DO statement* executes the statement it contains repeatedly. A DO statement runs indefinitely unless

- the statement within it branches outside the DO and leaves the loop
- a WHILE or UNTIL clause controls its termination
- a FOR clause controls its termination

The simple DO statement has the form

```
DO statement;
```

You may nest one DO statement within another.

You may precede the DO keyword with a label, enabling a GOTO statement to pass control to the loop, but you must not place a label between DO and its following statement. If the statement following DO is itself a BEGIN...END Compound statement, you may label only the statements between the BEGIN and END keywords. You must not label the BEGIN...END Compound statement itself. You can pass control out of the DO loop with a GOTO statement, and of course from one point to another within the loop. But it is illegal to attempt to pass control from outside the loop to any point inside it. In the program fragment

```
DO BEGIN
     A := A!!"B";
     IF (LENGTH (A)) > 10 THEN GOTO BYE;
   END;
BYE:  statement;
```

the concatenation of Bs onto string A continues until the length of A exceeds 10. Then the GOTO statement branches outside the DO loop.

The WHILE and UNTIL Clauses

You can control a DO loop with preceding and following *WHILE clauses* and *UNTIL clauses*. Any of the nine options below is legal:

$$\left[\begin{matrix}\textit{UNTIL bool-exp}\\ \textit{WHILE bool-exp}\end{matrix}\right] \text{ DO statement } \left[\begin{matrix}\textit{UNTIL bool-exp}\\ \textit{WHILE bool-exp}\end{matrix}\right] \ ;$$

If the expression in the WHILE or UNTIL clause is not Boolean, the compiler converts its type to Boolean. If the WHILE or UNTIL clause precedes the DO statement, runtime checks the condition *before* executing the DO statement. If the WHILE or UNTIL clause follows the DO statement, it checks the condition *after* executing the DO statement.

If the expression in a preceding WHILE clause is false, the inner DO statement never executes. If it is initially true, execution of the DO statement begins, and continues until the expression becomes false. If nothing in the DO statement resets the value of the expression, the DO statement attempts to execute endlessly. In the following example, the DO statement never increments C, since the initial condition is false:

```
C := 100;
WHILE C < 0 DO C := C+1;
```

In the next example, the initial condition is true, and the DO statement begins execution. But since the DO statement changes the value of C only to make it larger, the initial condition remains true, and the DO statement attempts to loop indefinitely.

```
C := 100;
WHILE C > 0 DO C := C+1;
```

In the next example, the initial condition is true, and the DO statement begins execution. Each time it executes, it decrements the value of C. When the value of C reaches 1, the statement executes one last time, the condition becomes false, and control passes to the next statement.

```
C := 100;
WHILE C > 0 DO C := C-1;
```

If the WHILE clause *follows* the DO statement (and no clause precedes it), the DO statement executes at least once, since checking of the condition follows an execution of the loop. For example,

```
C := 100;
DO C := C+1 WHILE C < 0;
```

the value of C increments once to 101 before the WHILE clause passes control to the next statement.

WHILE clauses can both precede and follow a DO statement. In

```
C := 10;
Q := 100;
X := 20;
WHILE C >< Q DO Q := Q-C WHILE Q > X;
```

the statement checks first to see that C is still not equal to Q, then decrements Q, and then checks to see whether Q is still greater than X. If it is, it checks the initial condition again, and if not, control passes to the next statement.

The UNTIL clause works in a similar fashion, with the truth conditions reversed. If the expression in a preceding UNTIL clause is true, the inner DO statement never executes. If it is initially false, execution of the DO statement begins, and continues until the expression becomes true. If nothing in the DO statement resets the value of the expression, the DO statement attempts to execute endlessly. For example,

```
C := 100;
UNTIL C > 0 DO C := C+1;
```

the initial condition is already true, so the DO statement does not execute and control passes to the next statement. In

```
C := 100;
UNTIL C < 0 DO C := C+1;
```

the initial condition is false, so execution of the loop begins. But nothing in the loop can make the condition true, so it attempts to execute endlessly. In

```
C := 100;
UNTIL C < 0 DO C := C-1;
```

the initial condition is false, and execution of the DO loop begins. Since the DO loop decrements C, it eventually assigns the value -1 to C. The next time it checks, the condition will be true and control will pass to the next statement.

If the UNTIL clause follows the DO statement (and no clause precedes it), the DO statement executes at least once, since checking of the condition follows an execution of the loop. In

```
C := 100;
DO C := C+1 UNTIL C > 0;
```

the value of C increments once to 101 before the UNTIL clause passes control to the next statement.

UNTIL clauses can both precede and follow a DO statement. In the next example, the statement checks first to see that C is still not equal to Q, then decrements Q, and then checks to see whether Q is still greater than X. If it is, it checks the initial condition again, and if not, control passes to the next statement.

```
C := 10;
Q := 100;
X := 20;
UNTIL C = Q DO Q := Q-C UNTIL Q < X;
```

You may also use a mixture of WHILE and UNTIL clauses, for example

```
C := 10;
Q := 100;
X := 20;
WHILE C >< Q DO Q := Q-C UNTIL Q = X;
```

Here, the final value of Q is 20, since the expression in the following UNTIL clause becomes true before the check for falsity of the expression in the WHILE clause.

Figure 4-4 shows the use of a WHILE clause in a DO loop.

```
1     BEGIN INTEGER POW;
2       POW := 1;
3       WHILE POW < 16000 DO POW := 2*POW;
4     END;
```

Figure 4-4. WHILE Clause in a DO Loop

Line 1 declares an integer POW and Line 2 initializes it to 1. In Line 3, the DO loop assigns to POW the result of multiplying the previous value by 2. The WHILE clause terminates the loop when POW receives the value 16384, making the condition false. The program calculates (but does not display) the first 14 powers of 2.

Figure 4-5 shows the use of an UNTIL clause in a DO loop.

```
1     BEGIN REAL DIV_2;
2       DIV_2 := 5;
3       UNTIL DIV_2 < .00001 DO DIV_2 := DIV_2/2.;
4     END;
```

Figure 4-5. UNTIL Clause in a DO Loop

Line 1 declares a real DIV_2 and Line 2 initializes it to 0.5. In Line 3, the DO loop assigns to DIV_2 the result of dividing the previous value of DIV_2 by the real 2.0. The UNTIL clause passes control to Line 4 when the result is less than .0001.

The FOR Clause

You use a *FOR clause* to repeat a DO statement a computed number of times. Statements using a FOR clause have the following general form:

FOR control-var := for-elmt *[, for-elmt]* ... DO statement;

where:

control-var is { iden *[expr]* | SUBSTR-clause | BYTE-clause }

for-elmt is { expr | expr1 WHILE expr2 | expr1 *[STEP expr2]* UNTIL expr3 }

statement is statement *[UNTIL expr | WHILE expr]*

The statement following DO can be as complex as you like; it can be a BEGIN...END Compound statement, or another DO statement with its own FOR clause.

The controlled variable control-var may be any form of DG/L variable, subscripted variable, substring specified by the SUBSTR function, or byte of any of the preceding specified by the BYTE function.

The portion of the statement between the assignment operator and the DO keyword is the FOR list. If the FOR list has more than one element, the elements must be separated by commas. A statement with a comma list such as

```
FOR A := 1, 2, 3 DO B := B+A;
```

processes as

```
FOR A := 1 DO B := B+A;
FOR A := 2 DO B := B+A;
FOR A := 3 DO B := B+A;
```

Elements in the FOR list need not occur in any particular order or with any fixed increment between elements. Each element in the list can be an arbitrarily complex expression.

The FOR list can consist of any DG/L expressions, but it can also contain the more complex elements using the WHILE, UNTIL, and STEP keywords. The WHILE clause in a FOR list operates in the same way as a simple WHILE clause controlling a DO loop. In

```
INTEGER VAL;
INTEGER ARRAY A[10];
.
.
.
VAL := 0;
FOR VAL := VAL+1 WHILE VAL < 10 DO A[VAL] := VAL*VAL;
```

element 0 of A receives no assignment since the FOR controlled variable is set initially to 1. Element 10, the eleventh element of A, receives no assignment since the condition in the WHILE clause becomes false after the assignment to A[9] and the increment of the FOR controlled variable to 10. So i^2 for element A[i] is assigned only for A[1] through A[9].

The UNTIL clause in a FOR list operates in the same way as a simple UNTIL clause controlling a DO loop. The program fragment

```
INTEGER VAL;
INTEGER ARRAY A[10];
.
.
.
VAL := 0;
FOR VAL := VAL+1 UNTIL VAL = 10 DO A[VAL] := VAL*VAL;
```

assigns the same values as those obtained with the WHILE clause. A[0] does not receive a value since the FOR controlled variable VAL is reset in the FOR statement from 0 to 1, and A[10] gets no assignment since the expression in the UNTIL clause becomes true after the assignment to A[9].

The UNTIL clause in a FOR list can, however, operate in a more complex way than it does in a simple DO loop, using a *STEP clause*. This statement must use all three keywords in the form

```
FOR control-var := exp1 STEP exp2 UNTIL exp3 statement;
```

The assignment to the controlled variable specifies the initial value for execution of the statement. The statement first compares the value of the controlled variable with the expression in the UNTIL clause. If the value is less than the expression, the DO loop executes. The STEP clause specifies the increment (or decrement, if its expression is negative) of that value after each execution of the statement. After the execution of the statement, the controlled variable increments (or decrements) by the value in the STEP clause. The new value is then compared with the expression in the UNTIL clause. If the incremented value is still less (or the decremented value is still greater) than or equal to the expression following UNTIL, the statement executes again. Otherwise, control passes to the next statement. The program fragment

```
BEGIN INTEGER A;
EXTERNAL STRING PROCEDURE GETCOUTPUT;
OPEN (1, (GETCOUTPUT));
FOR A := 0 STEP 3 UNTIL 12 DO
    WRITE (1, "<NL>Value of A is ", A);
END;
```

generates

```
Value of A is 0
Value of A is 3
Value of A is 6
Value of A is 9
Value of A is 12
```

Ordinarily, you will not want to specify a Boolean operator in the UNTIL clause, since in the environment of a STEP clause the evaluation control-var > exp3 (or control-var < exp3 for decrementing values) is automatic. In fact, if your program fragment contained such an operator, as in

```
BEGIN INTEGER A;
EXTERNAL STRING PROCEDURE GETCOUTPUT;
OPEN (1, (GETCOUTPUT));
FOR A := 0 STEP 3 UNTIL A > 12 DO
    WRITE (1, "<NL>Value of A is ", A);
END;
```

you would see only

```
Value of A is 0
```

Since the Boolean expression A > 12 is false and upon type conversion becomes 0, the condition is met at the first increment (3 > 0) and control passes to the next statement.

A FOR list may be as complex as you like. Each list element acts as an individual FOR statement, but each has the same controlled variable. The statement

```
FOR I := 10, X STEP 1 UNTIL X+10, X+11 WHILE X < 25 DO statement;
```

does the same thing as the following three statements:

```
FOR I := 10 DO statement;
FOR I := X STEP 1 UNTIL X+10 DO statement;
FOR X+11 WHILE X < 25 DO statement;
```

You may nest one FOR statement within another. The following example

```
BEGIN INTEGER ARRAY MAT[5,5];
INTEGER I, J;
FOR I := 0, 1, 2, 3, 4, 5 DO
    FOR J := 0 STEP 1 UNTIL 5 DO
    MAT[I,J] := 0;
END;
```

initializes all the elements of a two-dimensional array to 0 by nesting one FOR loop within another. A nested FOR loop can differ in its internal structure from its controlling FOR loop even though, as in the fragment, they evaluate to the same values.

Figure 4-6 illustrates the use of a FOR loop.

```
1     BEGIN REAL (4) ARRAY RAD[1:3,1:100];
2       INTEGER COL, ROW;
3       FOR COL := 1, 2, 3 DO
4           FOR ROW := 1 STEP 1 UNTIL 100 DO
5                   RAD[COL,ROW] := ROW;
6       COL := 1;
7       ROW := 0;
8       FOR ROW := ROW+1 WHILE ROW < 101 DO
9           RAD[COL,ROW] := RAD[COL,ROW]^.5;
10      COL := 3;
11      ROW := 0;
12     FOR ROW := ROW+1 UNTIL ROW > 100 DO
13         RAD[COL,ROW] := RAD[COL,ROW]^2;
14    END;
```

Figure 4-6. FOR Loop

The program in Figure 4-6 creates a 3 by 100 array, and places in the three columns the square root, the number itself, and the square of the first 100 positive integers. Lines 1 and 2 declare the real array RAD (the array must be real since not all square roots are integers), and the integers COL and ROW, with which the program can refer to specific cells in the array. Line 3 abbreviates three separate statements, for which the value of COL is successively 1, 2, 3. DO introduces the loop that begins on Line 4. The loop in turn contains a FOR clause. Its controlled variable, ROW, is set initially to 1. Since 1 is less than 100, the interior DO loop executes, setting RAD[1,1] = 1. ROW then increments to 2 (by STEP 1) and, since 2 < 100, executes the DO loop again, setting RAD[1,2] = 2. The interior DO loop continues to execute, and ROW to increment, until RAD[1,100] = 100, at which point the UNTIL clause becomes false. The value of COL then becomes 2 for the outer loop, and control passes to the inner DO loop, with ROW reset to 1. The loop assigns RAD[2,i] = i, for each value i of ROW. When the UNTIL clause again becomes false, COL takes the final value 3. The interior loop again assigns RAD[3,i] = i for all values i of ROW. The result so far is the array

```
1.      1.      1.
2.      2.      2.
3.      3.      3.
4.      4.      4.
.       .       .
.       .       .
.       .       .
98.     98.     98.
99.     99.     99.
100.    100.    100.
```

Lines 6 and 7 reset COL from 3 to 1, and ROW from 100 to 0. Line 8 show a FOR clause with a WHILE clause. The initial value of the controlled variable ROW is 0+1 = 1, and the WHILE clause remains true until ROW increments to 101. The DO loop it controls assigns to each cell of the first column (since COL = 1), the square root (that is, the power .5) of its current element. The array now has the entries

```
1.                  1.      1.
1.4142135623731     2.      2.
1.73205080756888    3.      3.
2.                  4.      4.
.                   .       .
.                   .       .
.                   .       .
9.89949493661163    98.     98.
9.94984437106616    99.     99.
9.99999999999997    100.    100.
```

Lines 10 and 11 again reset COL and ROW, this time to 3 (for the third column) and 0. Line 12 shows a FOR clause with an UNTIL clause but without a STEP clause. The controlled variable ROW increments, and the DO loop executes, until ROW becomes 101. This DO loop assigns to each cell in the third column the square of its current entry. The array then has the entries

1.	1.	1.
1.4142135623731	2.	4.
1.73205080756888	3.	9.
2.	4.	16.
.	.	.
.	.	.
.	.	.
9.89949493661163	98.	9604.
9.94984437106616	99.	9801.
9.99999999999997	100.	10000.

which was the goal of the exercise. When ROW takes the value 101, control passes to the END of Line 14 and the program terminates.

The BEGIN...END Compound Statement

The *BEGIN...END Compound statement* treats the statements it encloses as a single statement. It has the form

[label] BEGIN *[stat-part]* END *[comment]*;

Each of the statements in the statement part except the last must end with the semicolon terminator; you may precede the END with a semicolon if you wish. While writing programs, you may leave the statement part empty as a placeholder for statements not yet developed. BEGIN...END Compound statements can nest within BEGIN...END Compound statements to any depth. In general, you can use a BEGIN...END Compound statement anywhere you can use a simple statement.

Placing a Compound statement within a DO loop lets you perform a sequence of operations repeatedly. In the fragment

```
CTRL := 0;
SEQ := "a";
CHAR := "x";
WHILE CTRL < 79 DO BEGIN
     SEQ := SEQ!!CHAR;
     WRITE (1, "<NL>"SEQ);
     CTRL := CTRL+1;
     END;
```

(assuming appropriate declarations), the BEGIN...END Compound statement first concatenates "a" and "x", then writes the result of that to the screen. It then increments the expression controlling the DO loop. Since CTRL is still less than 79, the DO loop executes again; the execution of the Compound statement continues until you have a string consisting of "a" and 79 "xs".

You may also use BEGIN...END Compound statements within IF statements. The example

```
BACK:   IF CTRL = 79 THEN
  GOTO STOP
  ELSE BEGIN
    SEQ := SEQ!!CHAR;
    WRITE (1, "<NL>", SEQ);
    CTRL := CTRL+1;
    GOTO BACK;
    END;
STOP:   ;
```

does the same work as the WHILE statement, using labels to pass control back to the IF expression and to the final END. You may place Compound statements in the THEN clause, the ELSE clause, or both.

When nesting Compound statements, the BEGIN and END keywords must properly enclose the component sentences. The structure

```
BEGIN
    st1;
    st2;
    BEGIN
        st3;
        st4;
        END;
    st5;
    END;
```

is acceptable, but the structure

```
BEGIN
    st1;
    BEGIN
    st2;
    st3;
    END;
```

is not acceptable, since it lacks one END, and

```
BEGIN
        st1;
        st2;
JUMP:   END;
        st3;
        st4;
SKIP:   END;
```

is not acceptable, since it lacks one BEGIN.

When you add declarations before the component statements of a BEGIN...END Compound statement, you create a DG/L program *block*. You may branch to a label within a Compound statement, but not to one within a subordinate block. Blocks are the basic building units of DG/L programs. Chapter 7 sketches their properties.

Figure 4-7 exemplifies a BEGIN...END Compound statement.

```
 1      BEGIN REAL (4) ARRAY L[1:100];
 2      INTEGER X;
 3      X := 0;
 4      REDO: IF X < 100
 5            THEN BEGIN
 6                 X := X+1;
 7                 L[X] := LOG10 (X);
 8                 GOTO REDO
 9                 END;
10      END;
```

Figure 4-7. BEGIN...END Compound Statement

Lines 1 through 3 declare an array and an integer, and set the integer to 0. Line 4 begins a labelled IF statement. The condition X < 100 is initially true, so the THEN clause executes. The THEN clause on Line 5 CONTAINS a BEGIN...END Compound statement with three component statements: Line 6 increments the variable which simulates the controlled variable of a FOR clause, Line 7 specifies the computation of a logarithm base 10, and Line 8 returns control to Line 4 for checking of the condition in the IF clause. If it remains true, the Compound statement executes again, and continues executing until X = 100.

This program utilizes a BEGIN...END Compound statement within an IF statement to mimic a DO statement with FOR, STEP, and UNTIL clauses.

```
BEGIN REAL (4) ARRAY L[1:100];
INTEGER X;
X := 0;
FOR X := X+1 STEP 1 UNTIL 100 DO
        L[X] := LOG10 (X);
END;
```

The Procedure Call Statement

The *Procedure Call statement* lets you execute built-in DG/L functions and routines, procedures you have declared and compiled externally to your current program, and procedures you have declared in the current program. Chapter 8 catalogs DG/L built-in functions. Chapter 9 lists useful DG/L built-in routines. Chapter 6 gives you the details of declaring procedures.

The general form of the Procedure Call statement is

procedure-name *[id-list]*;

where procedure-name is the name of the routine, function, or procedure you are calling, and *id-list* is the parameter list you are passing to it. For example, if you call the built-in WRITE routine with

```
WRITE (1, "<NL>", COUNT);
```

you write to whatever file is open on channel 1 (perhaps the terminal) a NEW LINE character and the current contents of variable COUNT. If you have declared (in the declaration part of the proper block) the procedure

```
PROCEDURE PHYPO (A, B, C);
REAL(4) A, B, C;
C := SQRT ((A^2)+(B^2));
```

then the Procedure Call statement (in the statement part of an accessible block)

```
PHYPO (S1, S2, H);
```

executes procedure HYPO, passing current values of S1, S2, and H to A, B, and C. After the procedure executes, control passes to the statement following the Procedure Call PHYPO.

A Procedure Call statement may occur anywhere any other statement may occur. For example, in

```
IF X > 0 THEN PHYPO (S1, S2, H);
```

the Procedure Call succeeds only if the condition X > 0 is true.

You may call the same procedure from different points in the program, passing the same or different values. You may call one procedure within another, and you may call a procedure from within itself (that is, recursively).

The procedure PHYPO is a *proper procedure* in that you declared no type for it. It does not return a value through its name. If, however, you declare a similar procedure with a type, as in

```
REAL (4) PROCEDURE FHYPO (A, B);
REAL (4) A, B;
FHYPO := SQRT ((A^2)+(B^2));
```

it is a *function procedure,* and it will return a result through its name. In this case the two Procedure Calls

```
FHYPO (H, S1, S2);
H := FHYPO (S1, S2);
```

are equivalent. The first argument H receives the result in either case.

The Dummy Statement

You can place inoperative *Dummy statements* in your DG/L programs. A semicolon with no statement preceding it counts as a dummy statement. You may place a semicolon dummy statement any place that you can use any other type of statement. The *optional* use of a semicolon in a DG/L statement, such as the one before the END keyword, is actually an example of the use of a dummy statement.

You may also use the keywords THEN, ELSE, WHILE, UNTIL and the BEGIN...END pair as dummy statements. Since you can place tabs, spaces, and NEW LINEs within and between DG/L statements, you can make listings of programs under development more readable. You can leave space for an option not yet defined, as in

```
IF BOOL
    THEN
    ELSE statement;
```

which passes to the next statement without taking any actions if BOOL evaluates to a Boolean true. You can remind yourself to complete a Compound statement, for example:

```
FOR Z := Z+2 STEP 2 UNTIL 256 DO
BEGIN

END;
```

The COMMENT Statement

The *COMMENT statement* allows you to intersperse descriptions, explanations, and reminders among the elements of an executable program. It has the form

COMMENT string-containing-no-semicolon;

and can contain spaces, tabs, and NEW LINES. The COMMENT statement can follow the BEGIN keyword. It must always end with the semicolon terminator. It cannot precede the initial BEGIN of a program. Otherwise, a COMMENT statement can occur anywhere any other statement can occur. Figure 4-8 summarizes the various DG/L commenting facilities.

```
1     BEGIN COMMENT Global declarations follow;
2     REAL (4) S1, S2, H;
3
4     /*External procedures below provides for terminal input
5     and output.*/
6
7     EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
8
9     PROCEDURE HYPO (A, B) HYPOLENGTH: (C);
10
11    /*String and interior parentheses replace comma in list above,
12    permitting you to identify parameter.*/
13
14          REAL (4) A, B, C;
15
16          C := SQRT ((A^2)+(B^2)); %This is the computation I want.
17
18    COMMENT The main program begins here.;
19
20    OPEN (0, (GETCINPUT)); %Double parentheses necessary here.
21    OPEN (1, (GETCOUTPUT)); %And here.
22
23    COMMENT Channels are now open for input and output;
24
25    WRITE (1, "Enter two sides:<NL>"); %Angle brackets for NEW LINE.
26    READ (0, S1, S2);
27
28    /*Procedure Call below passes control to procedure HYPO in the
29    declaration part of this program.  When the computation is done,
30    control passes to the statement below the Procedure Call.*/
31
32    HYPO (S1, S2, H);
33
34    WRITE (1, "<NL>Hypotenuse, given ",S1," and ",S2," is ",H,"<NL>");
35    END of main program;
36
```

Figure 4-8. Commenting Facilities

The program in Figure 4-8 computes the length of the hypotenuse of a right triangle from the length of the sides. The listing contains the various DG/L commenting facilities. In Line 1, a COMMENT statement follows the initial BEGIN of the program. The COMMENT statement must not and does not contain a semicolon. Lines 4 and 5 contain a delimited comment. A second asterisk must not follow the initial one, since that is the delimiter for conditionally compiled code. The delimited comment on Lines 11 and 12 explains that) HYPOLENGTH: (replaces a comma in the parameter list in Line 9. On Lines 16, 20, 21, and 25, the strings following the percent (%) sign are interpreted as comments. Lines 18 and 23 contain COMMENT statements preceding and within the main program. Lines 28 through 30 contain a delimited comment within the main program. Line 35 shows a comment between the END and the final semicolon terminator. That comment must not and does not contain any END, ELSE, WHILE, or UNTIL keyword.

End of Chapter

Chapter 5
Declarations

All DG/L programs and blocks must include *declarations* for the treatment of the data you supply. A declaration associates specific attributes with the identifiers you use to refer to the data. All declarations for a block must precede every executable statement in the block.

The *type* declaration for an identifier determines the kind of data element you wish to associate with it. A data element may be an INTEGER, a REAL, a character STRING, a BIT string, a BOOLEAN value, or a POINTER. You may specify a precision with which you wish to treat numerical and string values.

The *shape* declaration for an identifier determines the complexity of the data elements you wish to associate with it. You may specify that data elements for a given specifier have the properties of an ARRAY, a PROCEDURE, or a CLUSTER. If you omit a shape declaration for an identifier, DG/L treats the elements you associate with it as simple, scalar values.

The *storage class* declaration for an identifer determines the range of code within which it is valid, how its values are stored, and how its values can change when you access them again. If you omit a storage class declaration, DG/L treats the identifier as *local,* subject to the scope rules of block structure, which Chapter 7 describes. The storage class declarations are OWN, EXTERNAL, GLOBAL, and BASED.

The general format for an identifier declaration is

[storage-class-dec] type-dec *[shape-dec]* id-list;

A typical declaration explicitly specifying all attributes is

```
OWN INTEGER ARRAY ANAME[3,5];
```

This declares an array of integers, with storage class OWN.

Except for ARRAY and proper PROCEDURE declarations, you must include the type-dec. This declarator specifies the identifier's type. The shape and storage class declarators are optional. The id-list is made up of one or more identifiers, with dimension specifications for arrays. If you are declaring more than one identifier with the same set of attributes, separate the identifiers in the id-list with commas.

The SWITCH and LITERAL declarations do not readily fit into any of the above three categories, and they receive separate descriptions. Some procedures require VALUE and LABEL declarations, which Chapter 6 describes. Declarations describe the attributes of identifiers, but generally do not assign values to the identifiers. The OWN storage class and the LITERAL declaration are exceptions to this rule.

All DG/L variables have a property called scope, which determines from where in the DG/L program you can access them. Scope is discussed as part of program structure, in Chapter 7.

Some sample declarations are

```
REAL FLO;
```

which declares FLO as a real variable,

```
INTEGER (2) ARRAY MATT[1:5,1:5,1:5];
```

which declares MATT as a three-dimensional array of double-precision integers, each of whose dimensions ranges between 1 and 5, and

```
OWN BOOLEAN ARRAY LOGMAT[6];
```

which declares LOGMAT as a Boolean array of 7 elements (if you omit a lower bound on an array subscript, it is assumed to be zero), in the OWN storage class.

Type Declarations

You must declare the type of every identifier in a DG/L program that receives a value. The only exception to this rule is described under arrays. You use the LABEL declarator in conjunction with procedures. You generally use POINTER declarations in conjunction with BASED declarations. This section describes the details of all other DG/L type declarators.

The general format of a type declarator is

type-spec *[(precision)]*

A type-spec is one of the keywords BOOLEAN, INTEGER, REAL, POINTER, STRING, BIT, or LABEL.

The *precision* field determines how much precision (how many words of memory) you use for an identifier. You use the precision field only with type-specs INTEGER, REAL, BIT, and STRING; for each of these, DG/L uses default precision values if you omit the precision field. Numeric precision specifiers are processed at compile time, and must therefore be constants. String precisions are processed dynamically at runtime, and therefore can be defined with any DG/L expression.

By default, DG/L stores integers as two's complement quantities in single 16-bit words. You can declare a double-precision integer by specifying a precision of 2. If you declare a double-precision integer, the first word is the more significant, and the second word is the less significant. Non-OWN integers are not initialized. OWN integers are initialized to 0.

Integer	[word 0]
Integer (2)	[word 0] [word 1]

By default, DG/L stores reals as two 16-bit words. In the default representation, the most significant bit of the first word of the real is a sign bit (1 for a negative number, 0 for a positive), followed by a 7-bit exponent, represented in Data General's standard hardware floating-point format. The remaining 8 bits of the first word and the entire second word make up a 24-bit mantissa. In the case of a double-precision real, the additional two words extend the mantissa, making a 56-bit mantissa. DG/L does not initialize non-OWN REALs; it initializes OWN REALs to 0.

Real	[word 0] [word 1]
Real (4)	[word 0] [word 1] [word 2] [word 3]

For both integers and reals, other precision specifications are legal, but any specified value will result in the selection of one of the precisions described above.

The precision for a bit string or character string specifies its maximum length. The default precision for a string is 32 characters. DG/L stores the string in sixteen 16-bit words (32 characters), packed as two characters per word. You can specify the precision, setting a different maximum length, as any value from 0 to 32,767 characters.

BOOLEANs and POINTERs are always stored as single 16-bit words. A POINTER can assume any value, but a BOOLEAN assumes only values of 0 or 1.

```
Pointer [word 0]
Boolean [word 0]
```

DG/L strings have the internal representation

```
String [word 0] [word 2] [word 2]
```

```
Word 0: Byte pointer to the string
Word 1: Maximum length of the string
Word 2: Current length of the string
```

BASED strings do not have string specifiers. The default maximum length for strings is 32 characters. The null string is the default initialization for strings, whether BASED or not.

Some examples of type declarations are

```
BOOLEAN BOO, CRY;
```

which declares BOO and CRY to be Boolean values,

```
REAL (4) FNUM;
```

which declares FNUM as a double-precision real,

```
POINTER FINGER;
```

which declares FINGER as a pointer,

```
STRING (60) CHARS;
```

which declares CHARS as a string with a maximum length of 60 characters, and

```
BIT (8) PREC;
```

which declares PREC as a bit string with a precision of 8.

Shape Declarations

Identifiers may have the *shapes* scalar, ARRAY, PROCEDURE, or CLUSTER. If you make no shape declaration for an identifier, it is treated as a simple variable (numeric scalar, or string). The remainder of this section treats the ARRAY declaration; Chapter 6 describes the PROCEDURE and CLUSTER declarations.

Array declarations take the general form

[storage-class-dec] [type-dec] ARRAY array-id-list;

where *storage-class-dec* and *type-dec* are storage class declarators (OWN, EXTERNAL, or GLOBAL) and type declarators (INTEGER, REAL, BOOLEAN, STRING, BIT, or POINTER). The *type-dec* field is optional in an array declaration; this is true only for arrays. You must explicitly declare the types of all simple variables, but arrays have a default type of single-precision real. So

```
ARRAY IRVING[0:3,0:3];
```

declares the same array as

```
REAL ARRAY IRVING[0:3,0:3];
```

The form of an array-id-list is

id-list dim-spec *[,id-list dim-spec ...]*

The id-list is a list of one or more identifiers, and a dim-spec is a specification of the upper and lower bounds on an array's dimensions. The entire dim-spec is enclosed in square brackets ("[ ]"). By default, the lower bound on an array's dimension is 0. Thus,

```
REAL ARRAY RARR[5];
```

declares a 6-element, one-dimensional, single-precision real array, whose elements you could refer to with a subscript ranging from 0 to 5.

If you wish to declare an array with more than one dimension, separate the dimension list elements with commas. For example,

```
REAL ARRAY RAR2[5,5];
```

declares RAR2 as a two-dimensional, 6-by-6 element (single-precision) real array, with a total of 36 elements.

If you wish, you can specify an array dimension with a nonzero origin. Do this by specifying a pair of bounds for the dimension, where the two bounds are separated by a colon (:). The bound pair's first element is a lower bound on that dimension, and the second element is an upper bound. Either or both bounds can be negative, but the lower bound must always be less than the upper bound. Note that there is a default lower bound (0) for array dimensions, but there is no default upper bound. For example,

```
REAL ARRAY BAD[5:];
```

is an illegal declaration, and results in an error.

```
REAL (4) ARRAY RAR2[-5:0,5];
```

declares a two-dimensional, double-precision real array, where the first dimension varies between -5 and 0, and the second between 0 and 5. This array has 36 elements.

Arrays may have types REAL, INTEGER, POINTER, BOOLEAN, or STRING. You can declare real, integer, or string arrays with any precision that is permissible for simple variables of the same type; the precision field specifies the storage that will be used for each element in the array. For example:

```
REAL (4) ARRAY MATR[1:5];
```

declares MATR as an array to be stored in 20 memory words: 5 elements each consisting of 4 words.

You can specify array dimensions with any DG/L expression, as long as the expression can be evaluated at runtime. Type conversion is performed, if needed, so as to make the dimension specification an integer value.

You can include several array-id-lists for a single attribute list, if they are separated by commas.

If you declare more than one array with the same attributes and dimensions, you need not repeat the dim-spec for each one. Simply separate the identifiers in the id-list with commas, as in

```
INTEGER ARRAY INTMAT, NUMS[4,4];
```

which declares both INTMAT and NUMS as two-dimensional integer arrays with both dimensions varying between 0 and 4.

```
POINTER ARRAY PAR[1:10];
```

declares a one-dimensional array of pointers, subscripted from 1 to 10.

```
INTEGER ARRAY IAR[1,10];
```

declares a two-dimensional 2-by-11 array of integers.

```
REAL (4) ARRAY LATER[I,J,K];
```

declares a three-dimensional, double-precision real array. The upper dimension bounds I, J, and K must be defined in a dominant block or must be formal parameters (see Chapter 6) for this declaration to succeed.

```
STRING (10) ARRAY CHARAR, OTHER[5,5];
```

declares two 36-element string arrays, where each element can store up to 10 characters.

```
REAL ARRAY A,B[1:5,0:9], P[8];
```

declares A and B as 50-element, two-dimensional real arrays, and P as a 9-element, one-dimensional real array.

```
BOOLEAN ARRAY BOOBA[1:J,K:L*2];
```

declares a two-dimensional Boolean array, whose first dimension ranges between 1 and J (which must, of course, currently be defined), and whose second dimension ranges between K and (2*L).

You generate incorrect references to an array with variable bounds if all of the following conditions hold:

1. An array in an outer block or procedure has a lower bound defined by a variable (for a one-dimensional array), or has either bound of any but the rightmost bound pair in a multidimensional array defined by a variable.
2. You declare an internal procedure in this block with a formal parameter that has the same name as the variable used in the bound pairs of the array declaration. The two occurrences of the name need not refer to the same location in memory.
3. The subprocedure accesses that array with the variable bound as the index.

Incorrect allocation of the array and references results. Avoid this problem by changing the name of the procedure's formal parameter.

If you initialize any but the high bound of the last dimension of an array with an expression, the program evaluates the expression each time you access the array, in order to determine this array bound. This may lead to unexpected results. For example,

```
INTEGER ARRAY THIS[LOW := 1 : 20];
LOW := LOW + 1;
THIS [4] := 7;
```

will leave LOW as 1 because the array calculation for THIS[4] causes the assignment LOW := 1 to execute.

Internal Representation of Arrays

The storage for an array contains both the elements of the array and an array specifier. The array specifier contains the number of dimensions in the array and the extent of each dimension.

The name of the array refers to the beginning of the elements' storage. The array specifier is stored in negative offsets from the beginning of the storage for the elements. The word of the specifier that immediately precedes the first word of data contains the number of dimensions in the array; the next word contains the upper bound of the first dimension, the next the lower bound of the first dimension, and so on through all the dimensions, as the schema shows:

```
[Word   -2n+1]  Lower bound of dimension n
[Word   -2n]    Upper bound of dimension n
.               .
.               .
.               .
[Word   -3]     Lower bound of dimension 1
[Word   -2]     Upper bound of dimension 1
[Word   ]       Number of dimensions
[Word   0]      First word of data
[Word   1]      Second word of data
.               .
.               .
.               .
[Word   m-1]    Next-to-last word of data
[Word   m]      Last word of data
```

You can realize great storage advantages in programs which need a variable array size by declaring array dimensions dynamically, rather than reserving a maximum size that will not always be needed. If you wish to do this, define dimensions with variables or expressions rather than with numeric constants. For example:

```
.
.
B:   BEGIN INTEGER I,J;
.
.
.
I := 50; J := 100;
.
.
C:   BEGIN REAL ARRAY A[I,J];
.
.
I := J := 10;
GO TO C;
.
.
END C;
END B;
```

In this example, the values assigned to I and J in enclosing block B determine the size of the array allocated on entry to block C. If you define array dimensions using variables or expressions, their values must be accessible from the declaration. Since all of a block's declarations must occur before any of its statements, any identifiers used to define a dimension must already have been assigned values in an enclosing block. If you branch to the beginning of a block from within the block, a new block entry occurs, causing the identifiers declared in the block header to be reallocated, as with the statements within block C in the example.

Storage Class Declarations

If you do not specify otherwise, an identifier's *storage class* is local. Runtime allocates and frees storage space for a local identifier on entry to or exit from the block where you declared it. Local identifiers take no storage class declarator; the storage class declarators (OWN, EXTERNAL, GLOBAL, and BASED) let you override the ordinary rules of block structure.

The OWN Declarator

The OWN declarator lets you declare that an identifier's value be preserved each time that the program leaves the block declaring the OWN identifier, and that the previous value be restored the next time you re-enter that block. The format of a simple variable OWN declaration is

OWN type-decl id-list

OWN arrays are declared using the format

OWN *[type-decl]* ARRAY array-id-list

When a DG/L program starts running, it initializes all OWN variables except arrays to zero. It initializes OWN strings, like all other strings, to null strings.

You can declare integers, reals, pointers, Booleans, bit strings and character strings OWN. When declaring dynamically dimensioned OWN arrays, you run the risk of accessing data outside the defined array. This may happen if you re-enter a block with the dimensions of an OWN array larger than they were on last exit from the block, and you are not using the compiler's option for full subscript checking. DG/L allocates an OWN array once, at the beginning of runtime. It never reallocates an array.

The program in Figure 5-1 exemplifies the use of an OWN identifier.

```
 1 BEGIN STRING (120) LINE; STRING WORD;
 2
 3    STRING PROCEDURE GWORD (INSTR);
 4            STRING INSTR;
 5
 6            BEGIN OWN INTEGER LASTEND; /* OWN integers initially 0 */
 7                    STRING GWORDT;
 8                    BOOLEAN GOT;
 9
10                    GOT := FALSE;
11
12 SCAN:                      IF LASTEND => LENGTH (INSTR) THEN GOTO THRU;
13                    IF SUBSTR (INSTR, LASTEND+1) = " " THEN BEGIN
14
15                            IF GOT THEN GOTO LEAVE;
16                            LASTEND := LASTEND+1;
17                            END
18
19                    ELSE BEGIN
20
21                            GWORDT := GWORDT !! SUBSTR (INSTR, LASTEND+1);
22                            GOT := TRUE;
23                            LASTEND := LASTEND+1;
24                            IF LASTEND > LENGTH (INSTR) THEN GOTO LEAVE;
25                            END;
26
27                    GOTO SCAN;
28
29 THRU:                      IF GOT THEN GOTO LEAVE;
30
31                    LASTEND := 0;
32                    GWORDT := "";
33
34 LEAVE:                     GWORD := GWORDT;
35                    END OF PROCEDURE GWORD;
36
37    COMMENT MAIN PROGRAM BEGINS HERE;
38
39    OPEN (1, "AFILE");
40    LINE := " DG/L IS THE  COMPILER FOR ALL GOOD USERS";
41 CALL:      WORD := GWORD (LINE);
42    FORMAT (1, "#<NL>", WORD);
43    IF WORD < > "" THEN GOTO CALL;
44
45    END;
```

Figure 5-1. Using an OWN Identifier

This program produces the following output, on file AFILE:

```
DG/L
IS
THE
COMPILER
FOR
ALL
GOOD
USERS
```

The greater portion of this program is a function procedure, GWORD. As will be explained in detail in Chapter 6, function procedures return explicit results through their names. GWORD accepts a string of up to 120 characters as input. Each time the program calls GWORD, it returns a successive word from the input string (for purposes of this example, a word is defined as a group of nonspace characters; a space delimits another word). GWORD uses an OWN variable to store its position in the input string so as to return successive, and not repeated, words. When GWORD reaches the string's end, it returns an empty string. This tells the calling program that there are no more words to be found in the input string.

Line 1 declares two string variables for the main program. LINE stores the line of characters that GWORD scans, and WORD stores each successive result that GWORD returns.

Line 3 begins GWORD's declaration. It is declared as a string procedure, meaning that the result it returns will be a string. Line 4 declares the type of its argument string INSTR. It is not necessary to declare INSTR's precision, since it will be adjusted to match the precision of the actual parameter passed to the procedure.

Lines 6-8 declare variables to be used within the procedure. LASTEND is the OWN integer that will store an index into the input string. After each call to GWORD, it will store the position of the first space character after a word. Since LASTEND is an OWN variable, its last value will be maintained and available when the procedure is next called. Also, LASTEND is automatically initialized to zero. Line 7 declares a local string, to be used for temporary storage within the procedure. Line 8 declares a Boolean variable, GOT, which the procedure uses as a flag.

Line 10 initializes Boolean variable GOT to false each time the procedure is entered. It is not necessary to initialize string GWORDT, since all strings are automatically initialized to an empty string.

Line 11 checks the string offset, LASTEND, to determine whether it has been incremented beyond the end of the input string. If it has, the procedure exits, through label THRU, which reinitializes LASTEND to 0. Line 12 checks for a space in the LASTEND + 1 position of the input string. If it is found, and if GOT is true, meaning that a word has been located on this call to GWORD, the procedure exits, returning the word found. Line 15 increments LASTEND, and the procedure continues scanning by way of line 26.

If the current character is not a space, the program executes lines 20-24 (the IF statement's ELSE clause). At line 20, the present value of GWORDT is concatenated with the current character.

You must use a temporary value to build up the procedure's result instead of, for example, using GWORD := GWORD !! ..., which would lead to unintended recursive calls of GWORD.

Line 21 sets GOT to true, assuring that the program will treat the next space found as the end of a word. Line 22 then advances LASTEND, and line 23 prepares to exit the procedure and return the result word if that offset is now greater than the input string's length. If this is not the case, the program branches back to SCAN to process the next character.

If the procedure branches to Line 29 (label THRU), the string offset is greater than the length of the input string. If the program finds a word (if GOT is true), the procedure branches to LEAVE to return the result. If no word has been found (if only spaces have been located in this call to GWORD), lines 31 and 32 reset LASTEND to 0 and the temporary variable holding the result to an empty string.

No matter how the exit was reached, lines 34 and 35 set the procedure's result to the temporary value GWORDT and return to the calling program.

Line 37 begins the main program. Line 39 opens the output file (routines OPEN and FORMAT are discussed fully in Chapter 9). Line 40 gives LINE a value. Line 41 calls the procedure, and assigns the string it returns to string variable WORD. Line 42 sends the word to the output file, following it with a NEW LINE character. If the word returned is nonempty (meaning that the input string is not yet exhausted) line 43 branches back to CALL to get another word. Otherwise, the program ends, at line 45.

To compare the use of OWN variables to local variables, you may wish to recompile, link, and execute the program with the declaration

```
OWN STRING GWORDT;
```

in place of line 7.

The EXTERNAL Declarator

You can access variables and procedures defined outside your DG/L program by declaring them EXTERNAL. EXTERNAL declarations follow the same scope rules as any other storage class, and they can occur in the declaration part of any block. All EXTERNAL identifier names must be unique in their first five characters.

DG/L's external procedure facility lets you compile procedures separately from your main program. This saves compilation and debugging time by allowing you to separate a large program package into distinct modules. Reasons for using EXTERNAL data include communication with external procedures and runtime routines, as well as flexible data area initialization. You may define EXTERNAL data in assembly language. (Details on this process, and more information on externals, are included in the *DG/L Runtime Library (AOS and AOS/VS) User's Manual.*)

To access typed simple variables defined outside a DG/L program from within that program, use a declaration of the form

EXTERNAL type-dec */shape-dec/* id-list;

The **type-dec** can include a precision specification for reals, integers, and strings. The **id-list** can contain more than one identifier.

EXTERNAL arrays are declared using the format

EXTERNAL */type-dec/* ARRAY array-id-list

Similarly, you can access externally compiled DG/L procedures by declaring

EXTERNAL */type-dec/* PROCEDURE id-list;

The *type-dec* can include a precision specification for reals, integers, bit strings, and character strings. You need these only for function procedures (those that return a result through their names, described in Chapter 6). The **id-list** contains the procedure name or names. After making this declaration, you may refer to the EXTERNAL procedure as you would any other DG/L procedure.

The GLOBAL Declarator

The GLOBAL storage class declarator allows you to define data within one DG/L procedure that other procedures can access without passing the data as an argument. You can use this instead of an assembly language definition. The GLOBAL declaration resembles the OWN declaration except that any other procedure in which you declare the same item EXTERNAL can access the item.

The GLOBAL declaration has the form

GLOBAL type-dec *[ARRAY]* name-list;

For example,

```
GLOBAL STRING (256) S1, S2;
```

declares two strings, each with a maximum length of 256 characters, to be global.

To use the GLOBAL storage class effectively, follow these rules:

1. Declare a variable GLOBAL in one procedure (preferably, in your main program) and EXTERNAL in all other procedures that will access it. Assembly routines can refer to it using .EXTN.

2. Since strings and arrays are allocated at DG/L runtime, the procedure in which you declare a variable GLOBAL must execute at least once before any procedure in which you declare it EXTERNAL executes. The procedure need not stay active; you can return from it and the GLOBAL data remains there.

3. GLOBAL variables initialize the same way OWN variables do: strings and string arrays are null strings, all other scalars are zero, and arrays have no initial value. If these initial values are not acceptable, define them as assembly language entry points as described in the *DG/L Runtime Library (AOS and AOS/VS) User's Manual* and then declare them EXTERNAL in all DG/L programs.

4. If you define a GLOBAL ARRAY variable in a procedure that is not the main program, it is important that you not access it from any procedure, including the main program, that runs before the procedure in which it is declared without explicit array bounds. The difficulty is that the descriptors do not initialize until the declaration code has run. The code that figures out the dimensions of arrays moves to the beginning of a procedure, where all accesses of the array can share it. You will have no trouble if you declare the bounds of the object as part of the EXTERNAL declaration, but you cannot then use the variable bounds feature.

You may also declare procedures themselves to be GLOBAL, but only within a CLUSTER (see Chapter 6).

The SWITCH Declaration

Using the SWITCH declaration, you can declare identifiers representing label lists to be used in computed GOTOs. The format of a SWITCH declaration is

```
SWITCH id := label-list;
```

The id is the switch identifier. The label-list is made up of one or more labels, separated by commas. Labels can be simple labels (which can be integer labels), subscripted labels (in which the expression defining the subscript can be arbitrarily complex), or references to subscripted switch identifiers. In the last case, the expression defining the subscript can be as complex as you wish. Labels can also be clauses of the form IF bool-expr THEN label ELSE label.

The labels in the switch list are considered as being numbered from left to right, starting with 1. If you GOTO a subscripted switch, the appropriate label provides directions to the branch location. For example, if you have declared:

```
SWITCH SW := 7, SW[1], SW[2];
```

and you execute the following statement:

```
GO TO SW[3];
```

the program will eventually branch to integer label 7, after resolving SW[3] as SW[2], SW[2] as SW[1], and SW[1] as 7.

If you use a GOTO to branch to a location outside the switch list's range (for instance, if the last example had been a GO TO SW[4];), the GOTO is ignored and the program continues with the next sequential statement. This does not generate an error. Note that you cannot define arbitrary noncontiguous subscripts for switches as you can for subscripted labels.

The sample program in Figure 5-2 provides a simple explanation of the use of switches:

```
 1 BEGIN
 2    EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
 3    STRING INPUT;
 4    SWITCH SWI := LAB1, LAB2, LAB3;
 5
 6    OPEN (0, (GETCINPUT));
 7    OPEN (1, (GETCOUTPUT));
 8
 9 ASK:       WRITE (1, "ENTER STRING FOR TEST:<NL>");
10    READ (0, INPUT);
11    IF 0 = LENGTH (INPUT) THEN GOTO EXIT;
12
13 BR:        GOTO SWI [(INDEX ("ABC", SUBSTR (INPUT, 1)))];
14    WRITE (1, "NOT AN A, B, OR C<NL>");
15    GOTO ASK;
16
17 LAB1:      WRITE (1, "FIRST CHARACTER IS AN A<NL>");
18    GOTO ASK;
19
20 LAB2:      WRITE (1, "FIRST CHARACTER IS AN B<NL>");
21    GOTO ASK;
22
23 LAB3:      WRITE (1, "FIRST CHARACTER IS AN C<NL>");
24    GOTO ASK;
25
26 EXIT:      END OF EXAMPLE;
```

Figure 5-2. Using Switches

The program's outermost and only block begins at line 1. Line 2 accesses the external procedures GETCINPUT and GETCOUTPUT in the runtime library to resolve the names of the terminal input and output files under all operating systems. Line 3 declares INPUT, a string which will store characters entered from the terminal. Line 4 declares a SWITCH, SWI, which is associated with a three-element label list.

Lines 6 and 7 open the terminal input and output files.

Lines 9 and 10 send a prompting message and accept an input string from the terminal. If the input string is empty, line 11 causes a branch to exit from the program. Otherwise, the program would loop indefinitely.

Line 13 computes a subscript into the switch list based on the first character of the input string. The INDEX function will return one of four possible results. If it returns a 1, meaning that an A is the first character in INPUT, the GOTO branches through the first expression in the switch list, LAB1. If INDEX returns a 2, the GOTO branches through LAB2. Similarly, if INDEX returns a 3, the GOTO branches through LAB3. If INDEX returns a 0, meaning that the first character is not an A, B, or C, the GOTO is ignored and the program continues at line 14.

All four of the possible WRITEs are followed by GOTOs that proceed to the request for a new input string. Line 26 ends the program.

The program in Figure 5-3 uses switches to simulate operations of a simple calculator.

```
 1 BEGIN
 2   REAL (4) OP1, OP2, RESULT;
 3   STRING OP;
 4   INTEGER OP_CODE;
 5   LITERAL $PLUS (53R8),
 6           $SUBTRACT (55R8),
 7           $DIVIDE (57R8),
 8           $MULTIPLY (52R8),
 9           $Q (121R8);
10
11   EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
12
13   OPEN (0, (GETCINPUT));
14   OPEN (1, (GETCOUTPUT));
15
16   UNTIL OP = "QUIT" DO
17     BEGIN
18       WRITE ( 1, "Please enter operand 1, NEW LINE,<NL>",
19         " the operator, NEW LINE,<NL>",
20         " and operand 2:<NL>");
21       READ ( 0, OP1, OP , OP2);
22       OP_CODE := BYTE ( OP, 1);
23       GOTO CODE[ OP_CODE ];
24       FORMAT ( 1, "Sorry but I don't know the operator <QT>#<QT><NL>",OP);
25       GOTO SKIP_OUTPUT;
26
27 CODE[ $PLUS ]:
28       RESULT := OP1 + OP2;
29       GOTO END_LOOP;
30
31 CODE[ $SUBTRACT ]:
32       RESULT := OP1 - OP2;
33       GOTO END_LOOP;
34
35 CODE[ $MULTIPLY ]:
36       RESULT := OP1 * OP2;
37       GOTO END_LOOP;
38
39 CODE[ $DIVIDE ]:
40       RESULT := OP1 / OP2;
41       GOTO END_LOOP;
42
43 CODE[ $Q ]:
44         GOTO SKIP_OUTPUT;
45
46 END_LOOP:
47       FORMAT ( 1, "# # # = #<NL>", OP1, OP, OP2, RESULT);
48 SKIP_OUTPUT:
49     END;
50 END;
```

Figure 5-3. Imitating a Calculator

The literals declared on lines 5 through 9 resolve to the octal ASCII values of arithmetic operator symbols. Each literal is an identifier of a switch on the label CODE.

The program reads input, which it assumes is an arithmetic expression, into its REAL variables. It stores the first byte of OP, the variable for the arithmetic operator, as an integer in OP_CODE. Line 23 branches to the appropriate index of CODE, determined by the value in OP_CODE. If, for example, you request an addition, OP gets the value "+", line 22 assigns the first byte of the character string in OP (which is 53R8 for a "+") to OP_CODE, and line 23 indexes into the label array CODE with OP_CODE. Since CODE [53R8] is a defined label, the program branches to line 27, which specifies and stores the addition. If the user indexed into CODE with a value for which you defined no label, for instance 114R8, control would pass to the statement immediately after line 23, and line 24 would send an "unknown operator" message.

The program continues to loop until the user enters "QUIT" as an operator.

The LITERAL Declaration

The LITERAL declaration lets you assign a name to a DG/L expression. Its format is

```
LITERAL id (expr);
```

Following the LITERAL declaration, expr replaces any occurrences of the id. For example,

```
BEGIN LITERAL FILNAME("$LPT");
   .
   .
   OPEN (1,FILNAME);
```

will open file $LPT as file number 1.

If you declare constant values as literals, and use the literal identifiers at all points in your program that use the constants, you can make global changes to the values of constants. You need change only a single line (the LITERAL declaration), rather than changing numerous occurrences of numeric or string values.

The id is replaced by the expr at compile time. DG/L literals are a general method of expression for identifier substitution. The LITERAL declaration does not allow replacement of keywords or statements. Literals need not be strings or numeric constants. Thus,

```
LITERAL EXP(3*X+4);
Z := Z/EXP;
```

performs the same operation as:

```
Z := Z/(3*X+4);
```

Literals may be nested, as in:

```
.
.
LITERAL LEFT (Z);
LITERAL RIGHT (3*VAR+4);
LITERAL VAR (X);
LEFT := LEFT/RIGHT;
.
.
```

To make these statements work, you must declare variables Z and X and must assign them values before you reach the assignment statement. If this is the case, these statements do the same operations as the last example.

When you use literals, they behave as if they are enclosed in parentheses, as far as order of evaluation is concerned. For example, in

```
LITERAL A (4+3);
INTEGER B;
B := 14/A;
```

B receives a value of 2, not 6, since the operation of adding 4 and 3 occurs before the division.

You may nest literals to an arbitrarily complex level, but you may not define a literal recursively.

As with other DG/L identifiers, the rules of scope apply to literals. An identifier declared as LITERAL is global to the block declaring it and to blocks subordinate to it, but is not accessible from dominant or parallel blocks. You may redeclare a LITERAL in a subordinate block; the ordinary rules of block structure apply.

The POINTER Declaration and the BASED Storage Class

DG/L's POINTER data type and BASED storage class let you address variables on an assembly-language level. They also let you define flexible data structures and addressing modes; Chapter 10 describes this use of pointers. POINTERs contain memory addresses of data, and BASED variables are templates used to interpret the type of data (e.g., STRING, INTEGER (2), etc.) at the address contained in the pointer.

The format of a POINTER declaration is

[storage-class-decl] POINTER *[shape-decl]* id-list;

and the format of a simple BASED variable declaration is

BASED type-decl id-list;

You can declare a BASED array using the format

BASED *[type-decl]* ARRAY array-id-list;

In all three cases, the identifier list can have one or more elements. You can refer to several BASED structures with the same BASED variable template. Permissible types for BASED variables are REAL, INTEGER, POINTER, BOOLEAN, BIT, and STRING.

If you declare a BASED string, the declaration merely serves to describe a contiguous area in memory as containing characters. It does not create a string descriptor, so no information about the BASED string's current length is maintained. Therefore, using SETCURRENT (see Chapter 8) on a BASED string has no lasting effect. You can use SUBSTR or BYTE on a BASED string, but not LENGTH (Chapter 8 describes these functions).

If you declare a BASED array, the declaration serves to describe a memory area as containing an array. No array specifier is generated, so address computations must be valid and are made at compile time. For information on appropriate specifiers and descriptors, see the *DG/L Runtime Library (AOS and AOS/VS) User's Manual.*

You may declare a BASED STRING ARRAY, but the compiler will assume that the BASED structure has been properly initialized with string specifiers. For information on creating these specifiers, refer to the *DG/L Runtime Library (AOS and AOS/VS) User's Manual.*

For reals, integers, bit strings, and character strings, you may include a precision specification.

Declaring a BASED variable does not allocate memory space for data storage; the BASED variable declaration serves only to direct the type interpretation of the memory locations involved.

To allocate memory space for storage of based data items, use the ALLOCATE routine. Chapter 9 describes this routine; an example is

```
ALLOCATE (PTR, 400);
```

which allocates 400 contiguous memory words, initializing pointer PTR to point to their beginning.

Alternately, the ADDRESS function, which is described in Chapter 8, returns a POINTER value pointing to the address of a DG/L variable. An example is

```
PTR := ADDRESS (AVAR);
```

which assigns PTR the address of variable AVAR. If you had declared

```
BASED REAL (4) BR4;
BASED INTEGER BI;
```

you could refer to words that PTR (or any other defined POINTER) points to as either double-precision reals or as single-precision integers.

To refer to data with POINTERs and BASED variables, you must include both the pointer and a based variable descriptor in this format

arith-expr -> based-var

This constitutes a simple pointer expression.

Unlike other integer quantities, a POINTER's default display representation (as used by the OUTPUT and WRITE routines) is in unsigned octal, with leading zeros.

The following example of the use of POINTERs and BASED variables assigns 101 consecutive 5-word groups, each of which contains a single-precision integer, from 0 to 100, followed by its 4-word, double-precision real representation:

```
BEGIN POINTER PTR;
  INTEGER I;
  BASED INTEGER BI;
  BASED REAL (4) BR4;
  .
  ALLOCATE (PTR, 505);
  FOR I := 0 STEP 1 UNTIL 100 DO
  (PTR+(5*I))->BI := ((PTR+(5*I))+1)->BR4 := I;
  .
  .
END;
```

The expression (PTR + (5*I)) describes a data address, and BI describes its contents as a single-precision integer. The expression ((PTR + (5*I)) + 1) describes the address one word above the previous one. It points to BR4, which you declared as a four-word quantity (a double-precision real number).

End of Chapter

Chapter 6
Procedures and Clusters

DG/L's procedure facility allows you to define modular functions and routines that will perform the operations your programs require. You declare a *procedure* as a single statement. The statement may be a simple statement, a compound statement, or a block. Procedure declarations, like all other declarations, occur in the declaration parts of DG/L blocks; you execute the body of a procedure when you use the procedure's name as a command name or argument in the statements of the DG/L block. There are two basic types of procedures: proper procedures and function procedures.

When you call a proper procedure in a statement, the name of the proper procedure acts only as a command name. In statements that call a function procedure, you may treat the procedure name as either a command name or as an argument to another command name. A function procedure returns an explicit result through its name; a proper procedure does not. You can pass arguments to DG/L procedures; the values given as actual parameters in a procedure call are substituted into the appropriate formal parameter positions in the procedure's body. A function procedure returns its result in its first argument.

DG/L procedures are re-entrant, and you may define recursive procedures. Both kinds of procedures can access globally defined data. Also, you can compile procedures separately from your main program. These procedures are called EXTERNAL procedures; all others are called internal.

Using Procedures

You declare a DG/L internal procedure with a PROCEDURE declaration. It has the general form

[type-dec [(precision)]] PROCEDURE proc-id *[(id-list)]* ;

If you use the *type-dec* field, you automatically specify a function procedure. You may include a precision specifier only for data types STRING, REAL, and INTEGER. You cannot specify any shape other than the declaration of the shape PROCEDURE itself; you cannot, for example, declare an ARRAY PROCEDURE.

The proc-id is the procedure's name, and can be any legal identifier.

Declaring Procedure Parameters

The optional *id-list* following the procedure's name contains the names of the procedure's formal parameters, if any.

If the procedure has formal parameters, the PROCEDURE declaration must be followed by declarations of the formal parameters' shapes and types, before the procedure's statement. For example, in:

```
PROCEDURE EXAM (PAR1, PAR2);
    INTEGER PAR1;
    REAL (4) ARRAY PAR2[0:5,0:5];
    WRITE (PAR1, PAR2);
```

you declare two parameters, PAR1 and PAR2. Declarations in this position can declare only attributes of formal parameters. If you wish to declare local identifiers for the use of a procedure, the procedure must constitute a block in which you declare the identifiers. Formal parameter declarations within the procedure body take the form:

type-dec *[shape-dec]* id-list ;

If the shape for the local identifier being declared is ARRAY, the type-dec is optional, and as usual defaults to REAL. For ARRAYS, the id-list field may take the form of an array-id-list , with dimension specifiers. If the shape for the local identifier being declared is PROCEDURE (within the outer procedure), the type-dec is included only for a function procedure. You cannot declare a storage class in a parameter declaration. After the declarations of formal parameters comes the procedure's statement, which may be simple, compound, or a block.

Declaring a Proper Procedure

The form for declaring a proper procedure is

PROCEDURE proc-id *[(id-list)]* ;

The following example of a proper procedure declaration determines whether a character is alphabetic.

```
PROCEDURE ALPHA (C, RESULT);

    STRING (1) C;
    BOOLEAN RESULT;

    RESULT := (C >= "A") AND (C <= "Z");
```

To call this procedure, list its name followed by the two actual parameters, enclosed in parentheses:

```
ALPHA ("Q", RES);
```

This call determines whether the string "Q" is a letter and returns with RES set to the appropriate Boolean value.

Declaring a Function Procedure

The form for declaring a function procedure is

type-dec PROCEDURE proc-id *[(id-list)]* ;

followed by type declarations of the formal parameters in the *id-list*. The type-dec field describes the type of result that the procedure returns.

The following function procedure sums the elements in an array passed to it. The procedure contains a block.

```
INTEGER PROCEDURE SUM_ARRAY (A);
     INTEGER ARRAY A;
    BEGIN INTEGER I, J;
    J := 0;
    FOR I := LBOUND (A,1) STEP 1 UNTIL HBOUND (A,1) DO
         J := J + A[I];
    SUM_ARRAY := J;
    END;
```

This procedure uses built-in functions HBOUND and LBOUND (described in Chapter 8) to determine the dimensions of the one-dimensional array parameter A. It also uses local variables I and J to form the sum. The procedure's name, SUM_ARRAY, receives the result.

Calling a Proper Procedure

A procedure is a *proper procedure* if you do not explicitly declare its type. A proper procedure does not return a result through its name. If, for example, you declare a proper procedure GOUT and then specify statements like

```
IF GOUT = Q THEN GOTO CRASH;
A := GOUT;
```

you will receive an error message.

Although a proper procedure can modify nonparameter values, the conventional and suggested way for passing data to and from proper procedures is through its list of parameters.

A call to a proper procedure has the statement form

proc-id *[(arg-list)]* ;

After a proper procedure is called and returns, control continues with the statement following the call.

The proc-id is the procedure's name, and the *arg-list* is a parameter list. Generally, the number of values in the *arg-list* should match the number declared within a procedure. If this is not the case, missing or superfluous arguments are dealt with according to the rules described later in this chapter.

If the name of a proper procedure occurs within its body other than as an unparenthesized parameter passed to another procedure, recursion results.

As an example of a proper procedure, consider the program in Figure 6-1, which declares and uses a procedure named PHYPOTENUSE.

```
 1 BEGIN REAL A, B, C;
 2   EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
 3   PROCEDURE PHYPOTENUSE (SIDE1, SIDE2, HYP);
 4
 5     REAL SIDE1, SIDE2, HYP;
 6
 7     HYP := SQRT ((SIDE1^2)+(SIDE2^2));
 8
 9   COMMENT BEGINNING OF MAIN PROGRAM;
10
11   OPEN (0, (GETCINPUT));
12   OPEN (1, (GETCOUTPUT));
13
14   WRITE (1, "ENTER TWO SIDES:  <NL>");
15
16   READ (0, A, B);
17   PHYPOTENUSE (A, B, C);
18
19   WRITE (1, "HYPOTENUSE, GIVEN:  ", A, " AND ", B, " IS:  ", C, "<NL>");
20
21 END;
```

Figure 6-1. Proper Procedure

Line 1 begins the program. Line 2 refers to the external procedures GETCINPUT and GETCOUTPUT in the runtime library. These external procedures resolve the names of the terminal input and output files under all operating systems. Lines 3-7 declare procedure PHYPOTENUSE. Note that, since it is a proper procedure, there is no type declaration in its header. Line 5 declares the type of the procedure's three formal parameters, and line 7 expresses the procedure body as a single simple statement.

Lines 11-16 open the terminal for input and output and send a prompt to request input. When the READ routine has read two input values, the program calls procedure PHYPOTENUSE (line 17). The two input values are the first two parameters, and the third parameter will receive the procedure's result. Line 19 writes a result to the terminal, and the program ends.

Calling a Function Procedure

The reference to the function procedure takes the form

proc-id *[(arg-list)]*;

A DG/L function procedure returns a value that you can access with the name of the procedure.

A call to a function procedure uses the function procedure's identifier as an expression that has a value. This value can be assigned to a variable, can itself be a parameter passed to another procedure, or can be part of an expression. For example, if you declare a real function procedure GOKAY, and Q and A are real variables, then statements like

```
IF GOKAY = Q THEN GOTO WIN;
A := GOKAY;
```

are valid.

After a function procedure returns with its value, control passes to the next evaluation step in the expression referring to it in the calling program. The value a function procedure returns is assigned to the procedure's name within the procedure, as in:

```
REAL PROCEDURE FHYPOTENUSE (SIDE1, SIDE2);

    REAL SIDE1, SIDE2;
    FHYPOTENUSE := SQRT ((SIDE1^2)+(SIDE2^2));
```

This function procedure returns the same result as procedure PHYPOTENUSE, which was described in the last section. You could call this procedure with

```
C := 2 * FHYPOTENUSE (2, X);
```

The value of FHYPOTENUSE is itself an expression, and can be used as a component of more complex expressions.

If you assign a value to a function procedure's name more than once, the last value assigned is the one returned on exit from the procedure.

If the procedure name occurs within the procedure body, except on the left side of an assignment symbol or as an unparenthesized parameter passed to another procedure, recursion results.

You can call a function procedure within a statement if you provide an identifier to receive the result returned, as in:

```
C := FHYPOTENUSE (A, B);
```

You may also use function procedures within expressions, and as arguments to other functions or routines. This allows you to define complex operations in single statements. For example,

```
SINE := FHYPOTENUSE (A, B) / A;
```

returns the result of the procedure to FHYPOTENUSE, divides the result by the value for the first side, and assigns the quotient to SINE.

You may call a function procedure as a statement with one more argument than is declared in its header. In this case, the leftmost argument will receive the procedure's result. For example, the two calls:

```
C := FHYPOTENUSE (A, B);
```

```
FHYPOTENUSE (C, A, B);
```

have the same effects. Errors may occur if you access a function procedure without providing some logical entity to receive its result, either through an assignment or as a leftmost parameter.

The VALUE Declarator and Temporary Copying

A procedure can modify the values of the actual parameters with which you call it (as opposed to the formal parameters in its declaration), whether the parameters are variables or constants. Often, this is a desirable means of returning a result from a procedure, but sometimes it has unfortunate results, as in

```
PROCEDURE DISTURB (ARG);
    INTEGER ARG;
    ARG := 0;
```

Calling this procedure with

```
DISTURB (1);
```

has the disturbing effect of changing the value of the integer constant 1. This call results in a global change to the value of the integer constant 1. Procedure parameters are passed by address, and literal constant references are optimized so that all references to the same constant value point to the same address.

The DG/L language has two methods for protecting against undesirable modification of global values: the VALUE declaration and *shielding*. Both methods make a local, working copy available upon entry to the procedure, but retain the global value upon return to the calling program.

The VALUE declaration has the form

```
VALUE id-list;
```

VALUE declarations are used only for procedure parameters and must precede any other declarations in the procedure. You cannot declare labels, switches, or identifiers of procedures as VALUE.

Use a VALUE declaration to make a local copy of the identifiers in its id-list every time that the program calls the procedure. Program execution accesses the copy at each point that the procedure refers to the identifier. If you had written the procedure DISTURB instead as

```
PROCEDURE DISTURB (ARG);
    VALUE ARG;
    INTEGER ARG;
    ARG := 0;
```

it would not have modified the value of a constant passed to it.

Shielding lets you protect a value from undesired modification by enclosing the value in an extra set of parentheses. You may do this independently at each particular procedure call. The copy is made at the point of call. This is particularly useful for protection of constants. For example,

```
DISTURB ((1));
```

does not disturb the global value of numeric constant 1.

Parameter-Passing Rules

You can substitute the actual parameters that you specify in a procedure call for the appropriate formal parameters in the body of the procedure's declaration. This section presents a set of rules for actual parameters, formal parameters, and parameter passing in DG/L programs.

1. Unless you specify otherwise, with shielding or the VALUE declaration, all DG/L variable or constant parameters are passed through call by reference. Program execution evaluates all actual parameter compound expressions (except trivial expressions like -1) at the point of call and passes their results in temporaries. A result can be returned only through a stored parameter, not a compound expression.

2. With certain exceptions (see Rule 8), the types, shapes, storage classes, and precisions of the actual parameters passed to a procedure must match the corresponding specifications in the procedure's formal parameter declarations. Otherwise, the DG/L compiler generates errors. When you pass a BASED string to a procedure, DG/L passes it using a string specifier and treats it for all intents and purposes as a string.

3. If the number of parameters passed to a procedure is less than the number of parameters declared in the procedure's header, DG/L gives any additional scalar formal parameters zero values. DG/L gives additional array or string parameters null array or null string specifiers, respectively. A null string specifier has a zero maximum size and length. A null array has a dimension count of zero. If you omit a label or switch expression parameter, an attempt to branch to it results in a normal exit from the procedure.

4. If the number of actual parameters passed to a procedure is greater than the number of formal parameters declared in the procedure's header, the DG/L program ignores the additional actual parameters. The function procedure is exempt from this rule; you may use an additional parameter to store the result of the procedure.

5. You need not include a specification of bounds for an array or a precision specification for a formal string parameter. Unless specified otherwise, the formal array or string parameter will assume the same size as its corresponding actual parameter.

6. You may redimension array parameters by putting explicit dimension specifications in formal array parameter declarations. These dimensions are local to the procedure declaring them. Similarly, you may include a maximum length specification for a formal string parameter. This string length will be local.

7. You may direct that selected bounds of a formal array parameter's dimension list be determined from the dimensions of the corresponding actual parameter. Do this by putting an asterisk (*) in the appropriate bounds specification field. For example:

```
BEGIN REAL ARRAY A[10, -1:8, 1:6];
.
.
PROCEDURE BOUND (X);
REAL ARRAY X [5, *:*, *:3];
.
.
END OF PROCEDURE BOUND;
BOUND (A);
```

has the same effect within procedure BOUND as:

```
REAL ARRAY X [5, -1:8, 1:3];
```

8. DG/L makes type conversions of actual scalar parameters so that they match the types of the corresponding formal parameters of non-EXTERNAL procedures. DG/L performs no type conversions for parameters of EXTERNAL procedures. If an actual parameter needs to be type-converted to match a formal parameter type, execution stores and accesses the converted value in a temporary, which is lost on exit from the procedure. It is the converted parameter, not the original, which is passed by value. This means that you must assure that the type of an actual parameter is correct before calling a procedure, if you use that parameter to return a result.

9. In parameter lists, and in fact anywhere in DG/L programs, you can replace commas by

)spaces string:spaces(

   Spaces are any number of blank spaces. The string must be made up wholly of alphabetic characters; no numbers, spaces, or control characters are permitted. The colon must immediately follow the string, with no intervening spaces.

   This construction lets you replace

   ```
   PROCEDURE ABSMAX (A,N,M,Y);
   ```

   with the more descriptive

   ```
   PROCEDURE ABSMAX (A) SIZE: (N,M) RESULT: (Y);
   ```

10. If an actual parameter is an assignment expression, DG/L performs the assignment before calling the procedure. If a value is returned through the parameter, only the variable to the left of the leftmost := will have its value changed.

Side Effects and Global Data

DG/L procedures can refer to identifiers declared in the blocks that enclose them. A *side effect* occurs when a procedure modifies a variable declared outside itself. Since the compiler does not analyze what side effects a procedure may have, procedure calls restrict optimization. They are a useful means of returning multiple results, however.

If a procedure that redeclares an existing identifier calls another procedure, as in

```
BEGIN INTEGER J, K;

PROCEDURE SAM1 (ARG);
    INTEGER ARG;

    BEGIN INTEGER K;

    K := ARG + 2;
    J := K * 2;

    END OF PROCEDURE SAM1;

PROCEDURE SAM2 (ARG);

    INTEGER ARG;

    BEGIN INTEGER J;

        J := ARG;
        K := J * 2;
        SAM1 (K);
    END OF PROCEDURE SAM2;
SAM2 (5);

END;
```

it is easy, at first glance, to lose track of what identifiers the procedures refer to. DG/L follows a simple and consistent set of rules in cases like these.

The identifiers that a procedure accesses are those valid for the block enclosing the procedure declaration. The identifiers are not necessarily the same as the valid identifiers set at the point that calls the procedure. If an identifier is redeclared within a procedure, the new declaration is valid within that procedure, but not necessarily in any procedures that it calls.

It is the *enclosing* block and its declarations, rather than the current block, that determines valid identifiers for the called procedure. Even for complex or recursive calling sequences, you determine the identifiers that a procedure refers to by inspecting the block in which the procedure's declaration nests.

Following these rules, it is easy to analyze the operation of the example in this section. Procedure SAM2 is called with an argument of 5. SAM2 redeclares J, which exists as a global identifier, and assigns the argument value (5) to its local J. J is multiplied by 2, and the result is assigned to the global variable K. SAM2 ends after calling SAM1 with an argument equal to 10. SAM1 declares its own local K, and computes the value of ARG + 2 (which, in this case, is 12) to the local K. This does not affect the global K, which maintains a value of 10. SAM1 then multiplies K by 2, giving a result of 24, and assigns that result to (global) J. Note that SAM1 refers to the globally declared J, not the local one declared within SAM2.

Recursive Procedures

Any DG/L procedure, whether proper or function, can be recursive. It can call itself either directly or with a series of procedure calls eventually leading to a call for it to perform its operation or to generate its result. You declare a recursive procedure in precisely the same way as a nonrecursive procedure. The only limit on procedure recursion level in DG/L is the size of the runtime stack. To call a procedure recursively, simply specify the *name* of the procedure within the *body* of the procedure's declaration.

Conversely, unless you *want* a recursive call, do not specify a procedure's name *within* its declaration, except for function procedures on the left side of an assignment symbol. Unintentional recursion is a common cause of stack overflow errors.

The computation of the factorial of an integer uses a recursive procedure, as in

```
INTEGER PROCEDURE FACTORIAL (N);

    INTEGER N;

    IF N <= 1 THEN FACTORIAL := 1
    ELSE FACTORIAL := N * FACTORIAL (N-1);
```

Another example using recursion is the program, INSERT, running under RDOS, in Figure 6-2. It could insert the program listing examples used in this manual into the text. It uses a recursive procedure, and the VALUE declarator.

```
1   BEGIN STRING (32) SBUF, TBUF;
2           POINTER BPTR;
3           BASED STRING LBUF;
4           EXTERNAL STRING PROCEDURE NAMEGROUND, GETCINPUT, GETCOUTPUT;
5   COMMENT THE READTEXT PROCEDURE IS THE HEART OF THE INSERT PROGRAM.
6           IT PROCESSES LINES FROM THE INPUT FILE, WRITING ORDINARY LINES
7           TO THE OUTPUT FILE. WHEN IT FINDS A LINE BEGINNING WITH A "%",
8           IT PROCESSES THE LINE AS A FILE NAME, CALLING ITSELF RECURSIVELY
9           TO INSERT THE CONTENTS OF THAT FILE IN THE TEXT STREAM;
10
11
12  PROCEDURE READTEXT (IN);
13          VALUE IN;
14          STRING IN;
15
16          BEGIN INTEGER BCNT; INTEGER (2) FPOS;
17
18          OPEN (4, IN, NOPE);
19          DO BEGIN
20
21           LINEREAD (4, BPTR, BCNT, RDER);
22           IF SUBSTR (BPTR->LBUF, 1) = "%" THEN BEGIN
23
24               FILEPOSITION (4, FPOS);
25               CLOSE (4);
26               READTEXT (SUBSTR (BPTR->LBUF, 2, BCNT-2));
```

Figure 6-2. Recursive Procedure (continues)

```
27              OPEN (4, IN);
28              POSITION (4, FPOS);
29              END
30
31          ELSE LINEWRITE (3, BPTR, BCNT, WTER);
32
33      END OF DO STATEMENT;
34
35  RDER: CLOSE (4);
36      GO TO LEAVE;
37
38  NOPE: FORMAT (2, "IMPOSSIBLE TO OPEN: #<NL>", IN);
39      GO TO LEAVE;
40
41  WTER: CLOSE (4);
42      FORMAT (2, "CAN'T WRITE TO: #<NL>", TBUF);
43      GO TO LEAVE;
44
45  LEAVE: END OF PROCEDURE READTEXT;
48
48  COMMENT THE MAIN PROGRAM BEGINS HERE. THE FIRST CALL TO
49          THE COMARG ROUTINE IS A DUMMY, ITS PURPOSE BEING TO
50          READ PAST THE FIELD IN THE COMMAND LINE CONTAINING
51          THE NAME OF THIS PROGRAM. THE NEXT TWO FIELDS ARE
52          THE INITIAL INPUT FILE NAME AND THE OUTPUT FILE NAME,
53          RESPECTIVELY. THE MAIN PROGRAM FIRST CALLS READTEXT
54          WITH THE INITIAL INPUT FILE NAME, AND SUBSEQUENT CALLS
55          TO READTEXT ARE RECURSIVE;
56
57          ALLOCATE (BPTR, 80);
58
59          OPEN (0, NAMEGROUND("COM.CM"));
60          OPEN (2, (GETCOUTPUT));
61
62          COMARG (0, SBUF); /* READ PAST FIRST FIELD */
63          COMARG (0, SBUF);
64          COMARG (0, TBUF);
65
66          OPEN (3,TBUF);
67          READTEXT (SBUF);
68
69          END;
```

Figure 6-2. Recursive Procedure (concluded)

Program INSERT is built around a recursive procedure, READTEXT. The following is a line-by-line explanation of its operation.

Lines 1-3 declare global identifiers: two strings, a pointer, and a based string. The based string will be used to refer to characters in the lines read from the input files. Line 4 refers to runtime EXTERNAL procedures NAMEGROUND, GETCINPUT, and GETCOUTPUT, to get the filenames for command file, terminal input, and terminal output. Lines 5-9 form a descriptive COMMENT statement.

Line 12 begins the declaration of PROCEDURE READTEXT. READTEXT takes one argument, IN, which represents the name of the text file to be read from. IN is a string, and is declared as VALUE. The VALUE declaration makes a local copy of IN when the program calls the procedure; without this, the line containing the filename would be lost when overwritten by subsequent LINEREADs.

Line 16 declares variables local to the procedure; BCNT will hold byte counts of lines read, and FPOS, a double-precision integer, will store file position information.

Line 18 opens the file whose name is contained in IN. If it is impossible to open this file, the procedure will branch to NOPE, issue an error message, and return. OPEN and the other built-in functions and routines used in this example are described in Chapters 8 and 9.

Line 19 begins the main statement of the procedure, a DO statement that executes the following compound statement indefinitely unless the compound statement branches outside.

Line 21 reads a line from the input file to the memory area pointed to by BPTR. The number of bytes read is returned in BCNT. If an error occurs in reading (most likely due to an end of file), the procedure exits, returning to the statement following the one that called the procedure. If the procedure was called recursively, the calling statement itself is in the procedure body, but the call was from a previous activation.

Line 22 checks whether the first byte of the line read is a "%," indicating a filename to be inserted. If it is not, the procedure continues at line 31, writing the line out to the output file. The DO clause at line 19 then repeats the process, reading a new line. If the first character is "%", the procedure executes the compound statement starting with the BEGIN at line 22 and continuing to the END at line 29.

Line 24 records the current position in the current input file, and line 25 closes that file. Line 26 calls READTEXT recursively, with a new input filename made up of the last line read with its delimiting "%"s and NEW LINE characters removed. When a recursive call terminates, it will return to line 27, which reopens the last input file. Line 28 sets that file's position to the position last recorded for that file. Note that the DG/L program did not have to explicitly create temporary variables to hold the values of string IN and integer FPOS, and that recursive calls can continue to arbitrary depth. Lines 35-43 deal with the various conditions that branch outside the compound statement, and line 45 ends the procedure. Note the comment between the END keyword and the terminating semicolon. The main program, lines 48-69, is quite simple. Line 57 allocates a memory area to the pointer used to reference text lines, line 59 opens the command file, and line 60 opens the terminal for output. With the command file opened, lines 62-64 can read successive filenames from the command line, and place them into strings. Line 66 opens the output file, and line 67 calls READTEXT with the first input filename. From that point on, control lies within READTEXT.

Chapter 7 describes DG/L recursion and block structure.

EXTERNAL Procedures

DG/L lets you compile procedures without enclosing them in the blocks that call them. Then any DG/L program can declare these procedures EXTERNAL and access them as needed.

The advantages of using EXTERNAL procedures include the fact that several DG/L programs can use a single procedure, once debugged and compiled. Also, you reduce net compilation and debugging time by the simple fact that EXTERNAL procedures comprising modular parts of program packages are shorter than the entire packages, and you can therefore compile and test them faster.

You declare EXTERNAL procedures in the EXTERNAL storage class. You can declare an EXTERNAL procedure in the declaration part of any block in a DG/L program. The ordinary rules of scope and block structure apply to EXTERNAL procedures, just as they do to all other procedures.

The EXTERNAL procedure declaration has the format

EXTERNAL *[type-dec]* PROCEDURE *[id-list]*

The *type-dec* is included only for function procedures. The examples in Figures 6-3 and 6-4 illustrate the use of an EXTERNAL procedure. The SORT procedure sorts a series of numbers; the calling program uses the EXTERNAL procedure.

```
1  BEGIN  INTEGER I;
2         INTEGER ARRAY A[10];
3         EXTERNAL PROCEDURE SORT;
4         EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
5         OPEN (1, (GETCINPUT));
6         OPEN (2, (GETCOUTPUT));
7
8         FOR I := 0 STEP 1 UNTIL 10 DO BEGIN
9
10           WRITE (2, "> ");
11           READ (1, A[I]);
12           END;
13
14        WRITE (2, "START SORT<NL>");
15        SORT (A);
16        WRITE (2, "END SORT<NL>");
17        FOR I := 0 STEP 1 UNTIL 10 DO
18           WRITE (2, A[I], "<NL>");
19
20 END;
```

Figure 6-3. Program SORTER

The following is EXTERNAL procedure SORT:

```
1 PROCEDURE SORT (A);
2
3         INTEGER ARRAY A;
4
5 BEGIN INTEGER I, T;
6         BOOLEAN DONE;
7
8 BUBBLE:
9         DONE := TRUE;
10        FOR I := LBOUND(A,1) STEP 1 UNTIL HBOUND(A,1) DO
11
12           IF A[I] > A[I+1] THEN BEGIN
13
14                T := A[I];
15                A[I] := A[I+1];
16                A[I+1] := T;
17                DONE := FALSE;
18                END;
19
20        IF NOT DONE THEN GO TO BUBBLE;
21
22 END;
```

Figure 6-4. EXTERNAL Procedure SORT

The combination of program SORTER and EXTERNAL procedure SORT requests 11 numeric values from the terminal and prints them out in ascending order.

SORTER declares an integer, I (at line 1), and an integer array, A (at line 2). At line 3, it declares EXTERNAL procedure SORT. You compile SORT separately, and load it along with SORTER; see Chapter 11. Line 4 refers to the EXTERNAL procedures GETCINPUT and GETCOUTPUT from the runtime library to resolve the names of the terminal input and output files under all operating systems. Lines 5 and 6 open the terminal for input and output. The FOR statement in lines 8-12 sends, prompts, and accepts terminal input. Since array A is declared as type integer, any other types of input data given to the program will be type-converted. Line 14 sends a message, and line 15 calls the procedure to sort the array. Lines 16-18 send a message, followed by the sorted numbers, and line 20 ends the main program.

The procedure, SORT, uses the bubble sort algorithm to sort the array passed to it as a parameter. A bubble sort interchanges adjacent elements that are out of order, continuing until it scans through the entire array, and finds no elements out of order. Value positions gradually "bubble" to where they belong, hence the name.

At line 1 in Figure 6-4, procedure SORT is declared as a proper procedure; line 3 declares its one parameter as an integer array. Since this line does not redimension A, A takes the same dimensions as the actual parameter passed to the procedure. Lines 5 and 6 declare variables to be used within the procedure. Line 8 starts the bubbling loop, and line 9 initializes the DONE flag to true. The FOR statement beginning at Line 10 and continuing to Line 18 sequentially checks each adjacent pair of array elements, switching them if they are not in increasing order. If such a swap occurs, DONE is set to false, meaning that another pass through the array will be needed. Line 20 goes back to the bubble loop if DONE is false; otherwise, the procedure ends, and returns.

The LABEL Declaration

Normally, when a called procedure reaches its END, the calling program continues evaluating the expression that contains the (function) procedure name (if you used the name as an argument), or continues to the next statement (if you used the procedure name as a command name).

With the LABEL declarator, you may specify branching from a procedure into any point in its calling program. If you wish to branch from a DG/L procedure to a point outside the procedure, you must pass the branch label to the procedure as a parameter. You must declare the procedure in the form

LABEL id-list;

You may use branching freely within the statements making up the body of a procedure; you need not declare these internal labels as type LABEL. You need the LABEL declarator only to describe label parameters passed to procedures.

You may declare SWITCH expressions and subscripted labels type LABEL and branch to them from a procedure. Their subscript expressions must be evaluable. Runtime evaluates them at the point of call, not within the called procedure.

Passing Procedures by Name

DG/L lets you declare procedure parameters as type PROCEDURE. This allows you to pass procedures by name to other procedures. If the procedure being passed is a function procedure, it must have a type specification in its parameter declaration. A procedure parameter declaration occurs in the called procedure and has the form

[type-dec] PROCEDURE *[id-list];*

The procedure APPLY illustrates passing a procedure by name.

```
REAL PROCEDURE APPLY (FNAME, DATA1, DATA2);
    REAL PROCEDURE FNAME;
    REAL DATA1, DATA2;
      APPLY := FNAME (DATA1, DATA2);
```

Real procedure APPLY accepts three parameters. The first is the name of a function procedure whose type is real; the other two are parameters to be passed to the procedure named by the first parameter. APPLY applies the function procedure to the two parameters, and returns the function procedure's result as its own. Therefore, if you had declared procedure FHYPOTENUSE as you did earlier in this chapter,

```
APPLY (FHYPOTENUSE, 3, 4);
```

would return the same result as

```
FHYPOTENUSE (3, 4);
```

Parameters passed to procedures passed by name do not type-convert; all parameters passed to such a procedure must have the appropriate data types, or errors result. Since the compiler must be able to determine all transfer addresses in a program, you cannot pass a variable as the name of a procedure; all procedure names passed as parameters must be fixed names.

Since a procedure name passed as a parameter to a procedure represents the procedure's address, a conflict arises with function procedures taking no arguments. A commonly used example is the built-in function MEMORY, which is discussed in Chapter 8. To pass the procedure's *result*, place the parameter in parentheses in order to treat the parenthesized name as an expression, as in

```
APRO ((MEMORY));
```

where APRO is a procedure taking one argument. To pass a procedure's address, so as to be able to call it from within the procedure's body, omit parentheses around the parameter, as in

```
APRO (MEMORY);
```

You may pass DG/L's built-in functions by name; Chapter 8 describes the mechanisms and considerations for doing so.

The CLUSTER Declaration

The CLUSTER declaration, a separate compilation facility, allows you to group together several procedures and their common data declarations. A cluster may contain any number of GLOBAL or local procedures, and any number of OWN or GLOBAL variables of any type. You may also declare LITERALs and BASED variables, and use INCLUDE files.

You *must not* place any executable statements in the main body of the CLUSTER. GLOBAL declarations must reside in block level 1.

A CLUSTER declaration has the form

```
CLUSTER cluster-name [arg1, arg2, ... , argn;
arg-dec1; arg-dec2; ... ; arg-decn ];
BEGIN
    [variable-declaration ...];
    [procedure-declaration ...];
END;
```

If a cluster allocates any data, as in the example below, you must call the cluster before you call any of the procedures within the cluster, in order to allocate variables properly. All arrays, strings, and bits require this allocation. In this example, the argument STACKSIZE no longer exists after the call to the cluster returns, so you must save the value passed if you want to use it again. Arguments that no longer exist are not checked for reference.

```
CLUSTER STACKSTUFF (STACKSIZE);
INTEGER STACKSIZE;
BEGIN
     OWN INTEGER STACKPOINTER, STACKLIMIT;
     OWN STRING ARRAY STACK[0:STACKLIMIT:=STACKSIZE];

      GLOBAL PROCEDURE PUSH(ARG);
      STRING ARG;
      BEGIN
           STACK[STACKPOINTER] := ARG;
           STACKPOINTER := STACKPOINTER+1;
           IF STACKPOINTER > STACKLIMIT THEN
              ERROR ("Stack Overflow");
      END;

      GLOBAL PROCEDURE POP;
     BEGIN
           STACKPOINTER := STACKPOINTER-1;
           IF STACKPOINTER < 0 THEN ERROR ("Stack Underflow");
           POP := STACK[STACKPOINTER];
      END;

      GLOBAL PROCEDURE STACKINIT;
      BEGIN
           INTEGER I;
           FOR I := 0 STEP 1 UNTIL STACKLIMIT DO
              SETCURRENT (STACK[I],0);
           STACKPOINTER := 0;
      END;
END;
```

A cluster with no arguments has the syntax

```
CLUSTER FOO;
BEGIN
     .
     .
     .
END;
```

If a procedure defined within a cluster does not contain the keyword GLOBAL, the procedure is visible only within the cluster, following normal DG/L rules of scope. A cluster resembles a procedure except that you must not declare local variables and code at the top level.

End of Chapter

Chapter 7
Block Structure

The addition of one or more declarations to the beginning of a compound statement turns it into a *block*. Every DG/L program consists of at least one block, and so must contain at least one declaration. The general syntax of a DG/L block is

BEGIN declaration-part *[statement-part]* END *[comment]*;

You can nest blocks within other blocks just as you can compound statements. Nested inner blocks must appear in the *statement part*, rather than the declaration part, of the enclosing block. For example,

```
BEGIN INTEGER A;
      REAL FLOAT;
      BEGIN STRING CHAR;
            CHAR := "I am a string.";
      END;
END;
```

is a legal nesting, since the inner block follows the only declaration of the enclosing block. But

```
BEGIN INTEGER A;
      BEGIN STRING CHAR;
            CHAR := "I am a string.";
      END;
      REAL FLOAT;
END;
```

is not a legal nesting, since a declaration of the outer block follows the inner block. Blocks are *statements*, and must occur in a statement part of a program, unlike procedures, which are *declarations* and must occur within a declaration part of the program.

Aside from the declaration part, a block retains the syntax of a BEGIN...END Compound statement. A block is *dominant* to one it contains, and is *subordinate* to one that contains it. It is *parallel* to any other block which is subordinate to the same dominant block. In

```
B0:     BEGIN
        dec-part
        B1:     BEGIN
                dec-part
                stat-part
                B2:     BEGIN
                        dec-part
                        stat-part
                        B3:     BEGIN
                                dec-part
                                stat-part
                                END of B3;
                        stat-part
                        END of B2;
                B4:     BEGIN

                        dec-part
                        stat-part
                        END of B4;
                stat-part
                END of B1;
        END of B0;
```

B0 encloses the entire program and is dominant to all other blocks. B3 is subordinate to B2, both B3 and B2 are subordinate to B1, and all three of B1, B2, and B3 are subordinate to B0. B2 is dominant to B3, but not to B4. B2 and B4 are parallel. Dominance and subordination are a function of the nesting of the BEGIN...END blocks above. The indentation above is merely a visual aid highlighting that structure.

Identifiers in a DG/L program have *scope*. This property means that an identifier is valid only for the block containing the declaration, or for blocks subordinate to that block. Other parts of the program--dominant or parallel blocks--cannot refer to them. Identifiers declared in parallel blocks, or in both dominant and subordinant blocks, are distinct, even if they consist of the same string of symbols, and their current values are independent of each other. For example,

```
BEGIN STRING REFER;
    REFER := "This sentence is false.";
    BEGIN STRING REFER;
        REFER := "This sentence is not about itself.";
        REFER := "Prove: "!!REFER;
        END;
    END;
```

has two distinct legal declarations using two different identifiers REFER. The concatenation specified in the subordinate block yields

"Prove: This sentence is not about itself."

and not

"Prove: This sentence is false."

As with compound statements, you may include comments between the END keyword of a block and its terminating semicolon. The comment must not include a semicolon or the keywords ELSE, END, UNTIL, or WHILE, even if they are enclosed in quotation marks.

Blocks, Scope, and Program Execution

Block structure is a key principle of DG/L programming. Although you can write relatively complex DG/L programs using only single blocks, use of inner, nested blocks makes for more comprehensible and memory-efficient programming. The DG/L runtime environment allocates storage for a block when it enters the block, and frees that storage when it leaves the block.

Statements in a block can access an identifier declared in that same block, or declared in a block dominant to that block. An identifier is local to the block you declare it in. Statements in dominant or parallel blocks cannot access it.

When a DG/L program starts running, no user-defined identifiers are valid or accessible. When program execution enters a block, the identifiers there become part of a list of currently valid identifiers. This list changes each time execution enters or leaves a block. Since no block is dominant to the outermost block, identifiers declared in that block are valid everywhere in the program; they are program-wide *global* identifiers.

If you redeclare an identifier in a subordinate block, the DG/L runtime environment suspends the global declaration and value of that identifier while in that block, and assumes the local declaration and value. The declaration in the declaration part of the subordinate block remains valid through the END of the block. The runtime environment restores the global declaration and value when it leaves the subordinate block.

You implicitly declare a label when you insert it in your program. The declaration is valid for the innermost block enclosing the label. Since you cannot access the label from a dominant or parallel block, you cannot jump to the label from outside the block.

You can, however, label the BEGIN of a block and jump to it, since that label is valid for the enclosing block. The BEGIN...END Compound statement (with no declarations) and a block (with at least one declaration) differ in this respect: you may jump to a label within a compound statement from outside the statement, but not to a statement within a block from outside that block. In the program fragment

```
B1:     BEGIN INTEGER A;
        .
        .
        .
        B2:     BEGIN INTEGER B;
                .
                .
                .
                B3:     BEGIN INTEGER C;
                        REAL ARRAY A[19];
                        .
                        .
                        .
                        END of B3;

                .
                .
                .
                END of B2;

        B4:     BEGIN BOOLEAN B;
                INTEGER D;
                .
                .
                .
                END of B4;

        .
        .
        .
        END of B1;
```

the identifier A represents a valid integer everywhere in block B1, except in the subordinate block B3, where it represents a 20-element real array. Program execution suspends the global integer value and declaration of A when it enters B3, and restores the global integer value and declaration when it leaves B3. Identifier B represents a valid integer in all of B2, including B3, and represents a valid Boolean value in B4. Outside B2 or B4, B is undefined. Identifier C is valid only in B3, and identifier D is valid only in B4.

When program execution reaches the END of a block or leaves the block with a GOTO statement, the storage space assigned to the identifiers declared in its header is released. (You also leave the block when a GOTO statement in it jumps you to the BEGIN of the block.) The identifier ceases to exist when program execution leaves the block that declares it. This permits efficient storage. In the example above, the DG/L program may store the values of C and D in the same location at runtime, since they never both exist at the same time. Similarly, the program can store the integer value of B defined in B2 in the same location used to store the Boolean B defined in B4. But since B3 needs both B and C, they cannot share storage in a single place. Storage requirements for small arrays and scalar values need not concern you greatly, but the runtime space efficiency of DG/L block structure becomes very important for programs with large arrays or many variables.

For effective and elegant use of DG/L block structure, you should declare temporary, local identifiers in the most subordinate blocks, so as to free their storage space outside those blocks. Declare identifiers needed over wider ranges in the headers of the more dominant blocks. Declare identifiers needed everywhere in a program in the outermost block of the program.

Blocks, Scope, and Recursion

The DG/L language lets you define recursive procedures. During recursive execution, the scope of an identifier is a function of its block nesting level, as given in the listing of the program. Scope is not a function of block recursion as it appears at runtime. An identifier redefined in a recursive procedure accesses a variable based on the blocks enclosing the procedure, not on the block that calls the procedure. In the program fragment

```
BEGIN INTEGER J, K, L;
      PROCEDURE PRO1;
          BEGIN INTEGER J;
                J := J+10;
                K := K-10;
                L := L+3;
                IF L < 11 THEN PRO2;
                END;
      PROCEDURE PRO2;
          BEGIN INTEGER K;
                J := J-10;
                K := K+10;
                L := L-2;
                IF L < 11 THEN PRO1;
                END;
J := 0;
K := 0;
L := 0;
PRO1;
END;
```

the enclosing block declares three integers. Procedure PRO1 redefines J, and reassigns values to it and to the global variables K and L. Procedure PRO1 calls procedure PRO2 if L is in the range 0 through 10.

Procedure PRO2 redefines K, and reassigns values to it, to the global variable J (and not the variable J declared in PRO1), and the global variable L. Procedure PRO2 calls procedure PRO1 if L is in the range 0 through 10.

The main program initializes the three global variables to 0 and calls PRO1 unconditionally. PRO1 increments its local variable J and decrements the global variable K. Since PRO1 does not redeclare a variable L, the increment in the assignment statement for L is to the global variable L of the main program. Since L is less than 11, control passes to procedure PRO2. PRO2 increments the global variable J and decrements its local variable K. Since PRO2 also does not redeclare L, the decrement in the assignment statement is to the global variable L of the main program. Since L is still less than 11, control passes back to PRO1. The net effect of the two procedures on the variable L is to increment it by 1. Control keeps passing back and forth between PRO1 and PRO2 until the value of L exceeds 10. The end values of the global variables J and K are negative, since PRO1 always decrements global K and has no effect on global J, and PRO2 always decrements global J and has no effect on global K.

End of Chapter

Chapter 8
Built-in Functions

DG/L supplies you with a wide variety of standard built-in functions that perform many commonly needed operations. DG/L functions fall into two main classes, arithmetic functions and general functions. The compiler recognizes both types; there is no need for you to declare them in your programs.

A function, like a function procedure, returns an explicit result through its name, allowing you to use it in expressions. A routine, unlike a function, does not return an explicit result through its name. It returns its results through parameters. Built-in routines are described in Chapter 9. The built-in function LOG10, for example,

```
B := 2 + (A := LOG10 (17));
```

gets the common logarithm of 17 and assigns it to variable A. Then, it adds 2 to this result, and stores the sum in B. Therefore, this function performs a series of operations within one statement. The use of the assignment statement format in this example and in the individual function descriptions does not imply that the results of built-in functions must be assigned to variables. You may use the result of a function as an operand in an expression or as an argument to another function. You must, however, remember that each function has a data type associated with its result (REAL, INTEGER, and the like), which will be used for any necessary type conversions. If you call a function when there is no place for the result to go, or if you call it with an incorrect number of arguments, errors may occur.

If you wish, you can write your own function procedures using the same names as the built-in functions. In this case, you must make declarations as with any other function procedures. Then DG/L will execute your function when the name is called, rather than the built-in function with the same name. This feature lets you revise basic functions, which you may need for certain specialized programs. For example, you could define your own function procedure DSIN, to compute a double-precision sine (or, for that matter, any other function to replace double-precision sine computations). You would declare this procedure as

```
REAL(4) PROCEDURE DSIN;
```

Any statement that calls DSIN, even when generically selected (discussed later in Chapter 8), calls your procedure rather than the built-in version.

You may pass the built-in functions by name. For example, Figure 8-1 contains a program fragment that defines a procedure, APPLY, which takes three arguments. The first is a function name, which may be a built-in. The second is an argument to be given to that function; and the third is a variable to receive the result:

```
.
PROCEDURE APPLY (FNC, DATA, RESULT);
REAL PROCEDURE FNC;
REAL DATA, RESULT;
RESULT := FNC (DATA);
.
.
APPLY (SQRT, 4.0, X);
COMMENT X WILL GET THE VALUE 2.0 PLUS OR MINUS A FUDGE FACTOR-THE SQRT OF 4.0 ;
.
```

Figure 8-1. Passing Built-in Functions by Name

You can pass any of the built-in functions described in this chapter by name. No generic selection of functions (discussed later in this chapter) occurs if built-in functions are passed by name; passing the name SIN will always call internal function SIN, never DSIN, regardless of the data type of the argument. If, then, you pass SIN by name and give it REAL(4) (double-precision) arguments, you may get erroneous results. To successfully pass the built-in functions by name, you must declare them in your accepting procedure and give them types as in Table 8-2 under OUTPUT DATA. For example, you would declare DABS as a REAL (4) PROCEDURE. You must give arguments matching the data types listed in the INPUT DATA column of Table 8-2. In addition,

- You must type ADDRESS as a POINTER procedure; it accepts any argument type.
- ROTATE is a single-precision INTEGER procedure which takes two single-precision integer arguments in both 16-bit and 32-bit environments. SHIFT differs from ROTATE in that, in 32-bit environments, SHIFT is a double-precision INTEGER procedure which takes two double-precision integer arguments.
- If you pass SIZE as an argument to a function, it is an INTEGER procedure which only accepts STRING arguments (not ARRAYs).

Names of the built-in functions consist exclusively of uppercase letters, and you must access them that way. For example, an attempt to call

```
z := mod (z, 6);
```

will succeed only if you have defined a function procedure mod, in lowercase letters.

Generic Selection of Arithmetic Functions

Table 8-1 shows the built-in arithmetic functions supplied with DG/L. As the table implies, the data type and precision of the operands force the compiler to choose which of its internal routines should be used for an operation. This process is called generic selection. For example, calling for the "SIN" of a single-precision real calls internal function SIN. If the real were double precision, though, DSIN would be called. All arithmetic operations supplied in DG/L can be performed with automatic internal function selection by calling mathematical functions with their generic names, as shown in the left column of Table 8-1.

The compiler uses two basic steps to select the appropriate internal function and convert data types. If you call a function by generic name, DG/L checks the correspondences shown in Table 8-1 and chooses an internal name based on the data type of the argument(s). If an internal name (one not listed in the leftmost column of Table 8-1) is used directly in the call, this step is omitted. In either case, the data type of the argument(s) is converted to the type the internal function requires. Table 8-2 lists these types. Whether called using a generic or an internal name, a called function executes, and yields an explicit result of type and precision as shown in Table 8-2.

DG/L allows you to give to an arithmetic function the parameters of any type that can convert to the type that the internal function requires. DG/L lets you use a wide range of data types, including reals, integers, pointers, and even strings, as arguments to arithmetic functions.

Table 8-1. DG/L's Generically Selected Arithmetic Functions

Generic Name	Internal Name (Based on Argument Type)							
	Int (1)	Int (2)	Rea (2)	Rea (4)	String	Point	Bool	Bit
ABS	IABS	ABS	ABS	DABS	DABS	IABS	IABS	DABS
ARCCOS	ACOS	DACOS	ACOS	DACOS	DACOS	ACOS	ACOS	DACOS
ARCSIN	ASIN	DASIN	ASIN	DASIN	DASIN	ASIN	ASIN	DASIN
ARCTAN	ATAN	DATAN	ATAN	DATAN	DTAN	ATAN	ATAN	DTAN
ATAN2	ATAN2	DATN2	ATAN2	DATN2	DATN2	ATAN2	ATAN2	DATN2
COS	COS	DCOS	COS	DCOS	DCOS	COS	COS	DCOS
COSH	COSH	DCOSH	COSH	DCOSH	DCOSH	COSH	COSH	DCOSH
ENTIER	FLOOR	DFLOOR	FLOOR	DFLOOR	DFLOOR	FLOOR	FLOOR	DFLOOR
EXP	EXP	DEXP	EXP	DEXP	DEXP	EXP	EXP	DEXP
FIX	IFIX	IFIX	IFIX	IFIX	IFIX	IFIX	IFIX	IFIX
FLOAT	FLOAT	DFLOAT	FLOAT	DFLOAT	DFLOAT	FLOAT	FLOAT	DFLOAT
LN	ALOG	DLOG	ALOG	DLOG	DLOG	ALOG	ALOG	DLOG
LOG10	ALOG10	DLOG10	ALOG10	DLOG10	DLOG10	ALOG10	ALOG10	DLOG10
MAX	MAX0	DMAX1	AMAX1	DMAX1	DMAX1	MAX0	MAX0	DMAX1
MIN	MIN0	DMIN1	AMIN1	DMIN1	DMIN1	MIN0	MIN0	DMIN1
MOD	MOD	DMOD	AMOD	DMOD	DMOD	MOD	MOD	DMOD
SGN	SGN	SGN	SGN	SGN	SGN	SGN	SGN	SGN
SIGN	SIG	SIG	SIG	SIG	SIG	SIG	SIG	SIG
SIN	SIN	DSIN	SIN	DSIN	DSIN	SIN	SIN	DSIN
SINH	SINH	DSINH	SINH	DSINH	DSINH	SINH	SINH	DSINH
SQRT	SQRT	DSQRT	SQRT	DSQRT	DSQRT	SQRT	SQRT	DSQRT
TAN	TAN	DTAN	TAN	DTAN	DTAN	TAN	TAN	DTAN
TANH	TANH	DTANH	TANH	DTANH	DTANH	TANH	TANH	DTANH

Table 8-2. Built-In Function Input/Output Data Types

Internal Name	Input Data		Output Data	
	Type	Precision	Type	Precision
ABS	Real	Single	Real	Single
ACOS	Real	Single	Real	Single
ALOG	Real	Single	Real	Single
ALOG10	Real	Single	Real	Single
AMAX1	Real	Single	Real	Single
AMIN1	Real	Single	Real	Single
AMOD	Real	Single	Real	Single
ASIN	Real	Single	Real	Single
ATAN	Real	Single	Real	Single
ATAN2	Real	Single	Real	Single
COS	Real	Single	Real	Single
COSH	Real	Single	Real	Single
DABS	Real	Double	Real	Double
DACOS	Real	Double	Real	Double
DASIN	Real	Double	Real	Double
DATAN	Real	Double	Real	Double
DATN2	Real	Double	Real	Double
DCOS	Real	Double	Real	Double
DCOSH	Real	Double	Real	Double
DEXP	Real	Double	Real	Double
DFLOOR	Real	Double	Real	Double
DLOG	Real	Double	Real	Double
DLOG10	Real	Double	Real	Double
DMAX1	Real	Double	Read	Double
DMIN1	Real	Double	Real	Double
DMOD	Real	Double	Real	Double
DSIN	Real	Double	Real	Double
DSINH	Real	Double	Real	Double
DSQRT	Real	Double	Real	Double
DTAN	Real	Double	Real	Double
DTANH	Real	Double	Real	Double
EXP	Real	Single	Real	Single
FIX	Real	Single	Integer	Single

(continues)

Table 8-2. Built-In Function Input/Output Data Types

Internal Name	Input Data		Output Data	
	Type	Precision	Type	Precision
FLOAT	Integer	Single	Real	Single
FLOOR	Real	Single	Real	Single
IABS	Integer	Single	Integer	Single
MAX0	Integer	Single	Integer	Single
MIN0	Integer	Single	Integer	Single
MOD	Integer	Single	Integer	Single
SGN	Real	Single	Real	Single
SIGN	Any	Any	Integer	Single
SIN	Real	Single	Real	Single
SINH	Real	Single	Real	Single
SQRT	Real	Single	Real	Single
TAN	Real	Single	Real	Single
TANH	Real	Single	Real	Single

(concluded)

Return the absolute value of expr.

Format

result := ABS (expr)

Example

VAR := ABS(-63)

gives VAR a value of 63.

ADDRESS *General Function*

Create a POINTER value to point to the address of a variable.

Format

pointer := ADDRESS (var)

Var is a subscripted or unsubscripted variable valid in the current block.

Remarks

1. In the 32-bit environment, the pointer returned is a double word.
2. The address function lets you address variables from a DG/L program on an assembly language level, permitting greater object code efficiency. Refer to Chapters 4 and 5 for further discussion of POINTER and BASED variables.
3. If you give ADDRESS an unsubscripted array argument (in the example, P := ADDRESS(B);), where, for example, B is a one-dimensional, ten-member array, ADDRESS returns the address of the array's first element.
4. If var is a string, the address returned is the address of the data, not the address of the string descriptor. If var is a substring, a nonfatal error will result if address is an odd-byte boundary, or a bit that does not start on a word boundary.

Example

If you execute the following DG/L program, it will give POINTER P the address of array element B[I].

```
BEGIN POINTER P; INTEGER ARRAY B [0:10];
INTEGER I;
.
.
.
P:=ADDRESS(B[I]);
.
END;
.
.
```

ARCCOS

Arithmetic Function

Return the value in radians of the angle having a cosine equal to the expression.

Format

```
result := ARCCOS (expr)
```

Remarks

1. If you give the ARCCOS function an argument with an absolute value greater than 1, the program issues a nonfatal error message and returns a value of 0.
2. ARCCOS returns a result from 0 to pi.

Example

```
VAR:=ARCCOS(.866)
```

gives VAR a value of .523649.

ARCSIN *Arithmetic Function*

Return the value in radians of the angle having sine equal to expression.

Format

```
result := ARCSIN (expr)
```

Remarks

1. If you give the ARCSIN function an argument with an absolute value greater than 1, the program issues a nonfatal error message and returns a value of 0.
2. ARCSIN returns a result from -(pi/2) to +(pi/2).

Example

```
VAR := ARCSIN(0.5)
```

gives VAR a value of .523579, the radian equivalent of 30 degrees.

ARCTAN

Arithmetic Function

Return the value in radians of the angle having a tangent of expression.

Format

result := ARCTAN (expr)

Remark

ARCTAN returns a result from -(pi/2) to +(pi/2).

Example

VAR:=ARCTAN(1.0)

gives VAR a value of .785398, the radian equivalent of 45 degrees.

ASCII or BYTE

General Function

Read or store the byte specified and return a 16-bit integer value, the left byte being all zero.

Formats

```
result := ASCII(variable,byte-num)
ASCII(variable,byte-num) := expr
```

The variable is any expression except an array without subscripts, byte-num is a single-precision integer that specifies the position of the desired character or byte in variable, and expr is a single-precision integer.

Remarks

1. To maintain compatibility with previous Data General ALGOL compilers, this function has two names. The two names are equivalent and you can use them interchangeably.
2. If a noninteger value is given as byte-num, the program converts it to an integer.
3. If you use BYTE or ASCII characters with parity bits, the program does not clear them.
4. You can now use the ASCII or BYTE function on the left-hand side of an assignment statement. It stores the bottom 8 bits of value in the specified byte number of the variable, without disturbing the other byte.
5. This function reads the byte from the variable specified by byte-num, returning a 16-bit integer with the left byte a zero, or stores the expression in a variable at the byte specified by byte-num.

Example

```
BEGIN LITERAL S ("ABCD"); INTEGER CHARVAL;
BASED INTEGER BI; POINTER P; INTEGER A;
.
.
.
A := ASCII (S,3);
/* A IS ASSIGNED THE VALUE 103R8, THE ASCII VALUE OF THE CHARACTER C */
.
.
CHARVAL := ASCII (P->BI, 2);
/* CHARVAL GETS THE VALUE OF THE SECOND
BYTE IN THE STRING POINTED TO BY P */
.
.
```

Return the value in radians of the angle whose tangent is expr1/expr2.

Format

result := ATAN2 (expr1, expr2)

Remark

ATAN2 returns a result from -(pi/2) to +(pi/2).

Example

VAR := ATAN2(1,2)

gives VAR a value of 9.99314E-8.

BYTE or ASCII

General Function

Read or store the byte specified and return a 16-bit integer value, the left byte being all zero.

Formats

```
result := BYTE(variable,byte-num)
BYTE(variable,byte-num) := expr
```

The **variable** is any expression except an array without subscripts, **byte-num** is a single-precision integer that specifies the position of the desired character or byte in variable, and **expr** is a single-precision integer.

Remarks

1. To maintain compatibility with previous Data General ALGOL compilers, this function has two names. The two names are equivalent and you can use them interchangeably.
2. If a noninteger value is given as **byte-num**, the program converts it to an integer.
3. If you use BYTE or ASCII characters with parity bits, the program does not clear them.
4. You can now use the ASCII or BYTE function on the left-hand side of an assignment statement. It stores the bottom 8 bits of value in the specified byte number of the variable, without disturbing the other byte.
5. This function reads the byte from the variable specified by **byte-num**, returning a 16-bit integer with the left byte a zero, or stores the expression in a variable at the byte specified by **byte-num**.

Example

```
BEGIN LITERAL S ("ABCD"); INTEGER CHARVAL;
BASED INTEGER BI; POINTER P; INTEGER A;
.
.
.
A := BYTE (S,3);
/* A IS ASSIGNED THE VALUE 103R8, THE ASCII VALUE OF THE CHARACTER C */
.
.
CHARVAL := BYTE (P->BI, 2);
/* CHARVAL GETS THE VALUE OF THE SECOND
BYTE IN THE STRING POINTED TO BY P */
.
.
```

CLASSIFY

General Function

Return an integer indicating the class that the entry belongs to, as defined in the class table.

Format

result := CLASSIFY (entry,class-table-pointer)

Entry is a single-precision signed integer. If entry is a byte value, it is treated as a full word, with the more significant byte being all zero. The class-table-pointer is the name of a table you define to group ranges of integers or ASCII characters.

Remarks

1. In the 32-bit environment, the pointer to the table is a double word.

2. All entries must be full words; you must define the class table in the following form:

```
least-value-#1
greatest-value-#1range-#1
result-#1
.
.
.
least-value-#n
greatest-value-#nrange-#n
result-#n
```

   CLASSIFY returns result-n if character is in the range between least-value-n and greatest-value-n. You can define any number of ranges, but you must include in some range all integers or ASCII characters that can be given to CLASSIFY. For example, thoughtful use of CLASSIFY on characters would provide table entries covering the entire ASCII range. If this is not done CLASSIFY will scan through all of memory in quest of a table entry image. Ranges may overlap; if they do, CLASSIFY will return the result defined for the first range the classified datum falls into.

Example

```
 1 BEGIN INTEGER I, J, K;
 2 EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
 3    BASED STRING LINBUF;
 4    BASED INTEGER PTABL;
 5    POINTER TPTR, BUFPTR;
 6
 7    ALLOCATE (BUFPTR,72);
 8    ALLOCATE (TPTR,33);
 9
10    FOR I := 0 STEP 1 UNTIL 9 DO BEGIN
11    (TPTR+(3*I))->PTABL := BYTE("A",1) + 3*I;
12    (TPTR+(1+(3*I)))->PTABL := BYTE("C",1) + 3*I;
13    (TPTR+(2+(3*I)))->PTABL := (I+1);
14    END;
15
16    (TPTR+30) -> PTABL := 0;
17    (TPTR+31) -> PTABL := 177;
18    (TPTR+32) -> PTABL := 10;
19
20    OPEN (0, (GETCINPUT));
21    OPEN (1, (GETCOUTPUT));
22
23 GET:       WRITE (1, "CHARACTER? ");
24    LINEREAD (0,BUFPTR,K,EXIT);
25    IF K=1 THEN GOTO EXIT ELSE
26    K := CLASSIFY (BYTE(BUFPTR -> LINBUF,1),TPTR);
27    WRITE (1, "CLASSIFICATION IS:  ",K,"<NL>");
28    GOTO GET;
29
30 EXIT:     END;
31
32
```

Figure 8-2. CLASSIFY Function

The DG/L program in Figure 8-2 requests characters from the terminal and classifies them. It divides the alphabet into triplets: "A", "B", and "C" return a value of 1, "D", "E", and "F" return 2, and so on. All nonalphabetic characters return a 10. A blank line (carriage return only), or a read error terminates the program.

In the declaration section (lines 1-5), BASED STRING LINBUF is defined as a template to store the lines read from the terminal. BASED INTEGER PTABL describes the class table as type integer, as it should be. Lines 10-18 initialize the class table; lines 16-18 define the default range (the entire ASCII set) for any characters not found in preceding ranges. Incidentally, the compiler will remove the invariant computation of BYTE("A",1) from the FOR loop to speed program execution. Lines 20-21 open the terminal for input and output. At line 23, the program requests a character, and reads the input at line 24. If there is a read error, or if the line length is 1 (<NL> only), the program branches to EXIT. At line 26, the BYTE function selects the first character from the line buffer, and this is presented to CLASSIFY, along with TPTR, which is a pointer to the class-table.

COS

Arithmetic Function

Give the cosine of an angle in radians.

Format

result := COS (expr)

Example

VAR := COS(.1)

assigns VAR a value of .995004.

COSH

Arithmetic Function

Return the hyperbolic cosine of an angle in radians.

Format

result := COSH (expr)

Example

VAR := COSH (3.14159)

gives VAR a value of 11.5919.

ENTIER *Arithmetic Function*

Give the largest integer value not greater than expr.

Format

result := ENTIER (expr)

The result data type for ENTIER is INTEGER.

Examples

VAR := ENTIER(3.125)

returns 3, and

VAR := ENTIER(-3.125)

returns -4.

EXP *Arithmetic Function*

Return the value of e (2.71828) raised to the expr power.

Format

result := EXP (expr)

Example

VAR := EXP(1.5)

returns a value for VAR of 4.48169, equivalent to (2.71828) ^ (1.5).

Return an integer value formed by truncating the real value expr.

Format

result := FIX (expr)

Examples

VAR := FIX (15.8)

returns a value of 15, but

VAR := FIX (-15.8)

returns -15.

FLOAT *Arithmetic Function*

Convert an integer value to a real value.

Format

result := FLOAT (expr)

Examples

```
VAR := FLOAT (35) / 10
```

gives VAR the value 3.5, but

```
VAR := FLOAT(35 / 10)
```

gives VAR the value 3, since parenthesized expressions are evaluated before arithmetic functions.

Get the currently declared upper bound of the dimension at the position of the integer expression in the dimension list of the array.

Format

result := HBOUND (array,integer-expr)

Array names an array, and integer-expr is a position in array's dimension list.

Remarks

1. In the 32-bit environment, the result is a double word.
2. If integer-expr has a value greater than the number of dimensions in the dimension list, HBOUND generates a fatal error.

Example

```
REAL ARRAY MAT[1: 9, 25, -2:4];
     /*ARRAY MAT HAS 3 DIMENSIONS*/
.
.
.
H1 := HBOUND (MAT,1)          /* HBOUND RETURNS 9*/
H2 := HBOUND (MAT,-2+4)       /* HBOUND RETURNS 25*/
H3 := HBOUND (MAT,H1/3)       /* HBOUND RETURNS 4*/
H4 := HBOUND (MAT,4)          /* THIS GENERATES AN ERROR*/
```

Find the first occurrence of substring in string-var, and return an integer index.

Format

result := INDEX (string-var,substring)

Remarks

1. The index points to the position in string-var where the match of string begins.
2. INDEX returns a value of 0 if the string is not in string-var.

Example

```
STRING (10) A; INTEGER I;
A := "ABCDEFGHC";
I := INDEX(A, "DE");
          /* "DE" is found in string A starting at position 4. INDEX returns a
          value of 4. */
I := INDEX (A,"CE");
          /* "CE" cannot be found. INDEX returns 0. */
I := INDEX (A,"C");
          /* "C" occurs first at position 3; INDEX returns 3. */
```

LBOUND

General Function

Get the currently declared lower bound of the dimension at the position of the integer expression in the dimension list of the array.

Format

result := LBOUND (array,integer-expr);
Array names an existing array, and integer-expr is a position in array's dimension list.

Remarks

LBOUND returns the currently declared lower bound of the dimension at position integer-expr of array's dimension list.

If integer-expr evaluates to a value greater than the number of dimensions in the dimension list, a fatal error occurs.

Example

```
REAL ARRAY MAT[1: 9, 25, -2: 4];
/* ARRAY MAT HAS 3 DIMENSIONS*/
.
.
L1 := LBOUND (MAT,1);          /* LBOUND RETURNS 1*/
L2 := LBOUND (MAT,-2+4);       /* LBOUND RETURNS 0*/
L3 := LBOUND (MAT,L1*3);       /* LBOUND RETURNS -2*/
L4 := LBOUND (MAT,4);          /* LBOUND GENERATES AN ERROR*/

.
.
.
```

LENGTH

General Function

Return the (integer) value of the number of characters currently in the string string-var.

Format

result := LENGTH (string-var)

Remarks

1. Do not confuse this function with the maximum size declared for string-var's dimensions; to get that value, use the SIZE function.

2. Since the current length of a BASED STRING is not stored, the LENGTH returned for a BASED STRING is always the maximum declared length, as would be returned by the SIZE function.

Example

```
BEGIN STRING (10) X; INTEGER I;
X := "ABCD";
I := LENGTH(X);
/* LENGTH RETURNS THE VALUE 4 */
```

Return the natural logarithm (to the base e) of expr.

Format

result := LN (expr)

Remarks

Logarithms are not defined for negative numbers. If you give LN a nonpositive argument, it will return a 0 value and print a nonfatal error message.

Example

VAR := LN (959)

assigns VAR a value of 6.86589.

Return the common logarithm (to the base 10) of expr.

Format

```
result := LOG10 (expr)
```

Remarks

Logarithms are not defined for negative numbers. If you give LOG10 a nonpositive argument, it will return a 0 value and print a nonfatal error message.

Example

```
A :=LOG10 (959)
```

returns a value of 2.98182.

Select the largest value in the parameter list.

Format

result := MAX (expr1,expr2,...,exprn)

Example

VAR := MAX (-13,0,2.6,-4)

gives VAR a value of 2.6.

MEMORY *General Function*

Give number of words of memory currently available for program use.

Format

result := MEMORY

Remarks

1. In the 32-bit environment, the function result is a double word.
2. MEMORY takes no arguments.

Example

```
IF MEMORY < 50 THEN GO TO DANGER;
/* BRANCHES TO "DANGER" IF LESS THAN 50
WORDS ARE AVAILABLE*/
```

Select the smallest value in the parameter list.

Format

result := MIN (expr1,expr2,...,exprn)

Example

VAR := MIN (30/10,2,23.6,8)

gives VAR a value of 2.

Return the value of expr1 modulo expr2.

Format

result := MOD (expr1,expr2)

Expr1 and expr2 are any valid DG/L expressions. Necessary type conversions are automatic. The defining expression is:

MOD := expr1 - (FIX (expr1/expr2))*expr2.

Examples

VAR := MOD (24,10)

returns the value 4, and

VAR := MOD (24,-10)

also returns 4.

ROTATE

General Function

Rotate the contents of a variable a number of bit positions left or right as the integer expression specifies.

Format

result := ROTATE (var,integer-expr)

Var is a 16-bit datum, and integer-expr is a signed integer.

Remarks

1. If integer-expr is positive, var is rotated right. Bits rotated out of bit 15 go to bit 0.
2. If integer-expr is negative, var undergoes a left rotate, with bits shifting out of bit 0 into bit 15.
3. If var is a numeric value, the internal representation of that quantity rotates.

Examples

```
R := ROTATE (R,1); % Rotates R one position to the right.
S := ROTATE (100017R8P1,-6); % S gets the value 1710R8.
```

In the second ROTATE, the initial value 100017R8P1 is represented internally as

1000000000001111

When rotated left six places, it becomes

0000001111100000

Return the value of expr1 with sign controlled by the sign of expr2.

Format

result := SGN (expr1,expr2)

If expr2 is negative, then the result is -expr1. Otherwise, the result is expr1.

Examples

A := SGN (-30,2)

returns -30 as a result, but

A := SGN (-30, -2)

returns 30.

SHIFT

General Function

Shift the contents of variable a number of bit positions left or right as specified in the integer expression.

Format

result := SHIFT (var,integer-expr)

Var is a 16-bit or 32-bit datum, and integer-expr is an integer expression.

Remarks

1. In the 32-bit environment, you can shift both 16-bit and 32-bit quantities.
2. SHIFT fills the leftmost or rightmost bits, as appropriate, with zeros.
3. If integer-expr is positive, then SHIFT moves the contents of var to the right. SHIFT fills the leftmost bits with zeros.
4. If integer-expr is negative, then SHIFT moves the contents of var to the left. SHIFT fills the rightmost bits with zeros.
5. If var is a numeric value, then that quantity's internal representation shifts.

Example

```
I := SHIFT (X,+4);
/* SHIFTS THE VALUE X 4 BIT POSITIONS
TO THE RIGHT, STORES THE RESULT INTO I.*/

Z := SHIFT (1000017R8P1, -6);
/* Z GETS THE VALUE 1700R8*/
```

Determine the sign of the expr.

Format

result := SIGN (expr)

If the expr is > 0, the result is 1. If the expr = 0, the result is 0. If the expr is < 0, the result is -1.

Examples

```
I := SIGN (0.5);          % I will be 1 5
I := SIGN (-0.5);         % I will be -1
```

Return the sine of an angle in radians.

Format

result := SIN (expr)

Example

VAR := SIN (60*3.14159/180)

assigns VAR a value of .866.

Get the hyperbolic sine of an angle in radians.

Format

result := SINH (expr)

Example

VAR := SINH (.5*3.14159)

returns a value of 2.302176.

Give the declared number of elements in an array or the declared maximum number of characters in a string.

Format

result := SIZE (array-or-string-var)

Remarks

1. In the 32-bit environment, the result of SIZE of an array returns a double word.
2. Do not confuse the declared maximum size of a string with its current dimension, which is available by using the LENGTH function.

Example

```
INTEGER ARRAY MAT [4,4,4] ; STRING (10) SNAME ;
STRING (2) ARRAY ANAME [3,3];
Z := SIZE (MAT);
          /*The function will return 125, since DG/L arrays have
          default origin 0.*/
Y := SIZE (SNAME);
          /* Y will receive 10*/
Z1 := SIZE (ANAME);
          /* Z1 will receive 16*/
```

Get the square root of expr.

Format

result := SQRT (expr)

Remark

Since real arithmetic does not permit taking square roots of negative numbers, a nonfatal error is generated if SQRT is given a negative argument. SQRT returns a value of 0 if such an attempt is made.

Example

VAR := SQRT (36)

assigns VAR a value of 6.

SUBSTR

General Function

Select a string (or byte) value from a string (or integer), for reference or storage.

Formats

result := SUBSTR(arg1,integer-expr1 *[,integer-expr2]*);
SUBSTR(arg1, integer-expr1 *[,integer-expr2]* := expr;

Arg1 is either an integer variable, a string, a pointer expression representing an integer or a string, or a literal.

Integer-expr1 and *integer-expr2* are the first and last character positions, respectively, of a substring in arg1, or the first and last bytes of the memory locations containing the integer value or the based integer.

Remarks

1. If *integer-expr2* is not specified, the SUBSTR returns the string value of the single byte specified by integer-expr1.
2. If integer-expr1 is less than 1, or if *integer-expr2* is greater than the length of a string (or number of bytes in the representation of an integer), SUBSTR fetches its result from memory outside the range of the string. No error will be flagged if this occurs, however.
3. You may use SUBSTR on the left side of an assignment statement to select positions in a string (or integer) to be respecified.
4. If you use a SUBSTR clause to put characters into positions in a string beyond the string's current length, the string will first be extended with spaces up to but not including the specified character positions, and then any specified respecifications will be performed.

SUBSTR (continued)

Example

```
LITERAL X ("ABCDEFGH");
STRING A, B, C, Z;

A := "JKLMNOP"; B:= "XXX";
C := "QQQ";
Z := SUBSTR (X,1,8); % Z entirety of X
Z := SUBSTR (X,5,7); % Z gets "EFG"
Z := SUBSTR (X, 4); % Z gets "D"
SUBSTR (A, LENGTH (A) +1, LENGTH(A) + LENGTH(B)) :=B;
       /* String B concatenated to string A.  A is now
       equal to "JKLMNOPXXX"*/
SUBSTR (A, LENGTH (A) -5, LENGTH (A)-3) := C;
       /* Replace part of String A with C.  A now equal
       to "JKLMQQQXXX".*/

INTEGER I; STRING J;

I := (102R8*400R8)+101R8;
J := SUBSTR (I,2); % Example of SUBSTR of an integer.
```

Figure 8-3. SUBSTR Function

The example in Figure 8-3 uses SUBSTR to return the string value of byte 2 of the single-precision word containing I. Byte 2's value is 101R8 (ASCII "A"), as can be seen from the following sketch of I's representation.

BYTE 1	BYTE 2
102R8	101R8

Return the tangent of an angle in radians.

Format

```
result := TAN (expr)
```

Example

```
VAR := TAN (45*3.14159/180)
```

gives VAR a value of .999999 (not quite 1, due to internal representations).

TANH *Arithmetic Function*

Return the hyperbolic tangent of an angle in radians.

Format

result := TANH (expr)

Example

VAR := TANH (.75*3.14159)

returns a value of .982193 for VAR.

End of Chapter

Chapter 9
Built-in Routines

DG/L provides you with a wide variety of built-in routines. Like the built-in functions described in Chapter 8, the compiler recognizes built-in routines without declaration.

With the exception of the FETCH and NODESIZE functions, described in Chapter 10, the built-in routines return values through their parameters rather than through their names. You call them as statements.

If you wish to override any of the built-in routines, you can define a routine with the same name as a built-in routine and declare it as a procedure; this directs all references to your version.

The routines are divided into three categories: General, Operating System Interface, and Cache Memory Management. The General routines perform a variety of operations, including maintaining the runtime environment, simple memory management, and numeric calculations providing more than one return value. The Operating System Interface routines provide a method for file handling and I/O operations. The Cache Memory Management routines, listed in Chapter 10, provide a unique and efficient method for handling large files effectively.

When a routine has an optional *err* field in its format, you may specify an address in your program to return to if an error occurs in the routine. If an error occurs and you do not specify the *err* field, the error may be fatal and will terminate your program. When you use the abbreviations **filenumber** or **integer-expr**, they are taken as integer quantities. However, if you give arguments of other types, they convert to the appropriate type. Similarly, DG/L assumes filenames to be strings. The names of the built-in routines are exclusively uppercase, and you must use uppercase to refer to them. For example, if you specify

```
open (1, "AFILE");
```

you call your own routine "open" rather than the built-in routine OPEN. If you have not defined a procedure called "open" (in lowercase characters), you generate an error.

Working with Files and the Environment

Many of the routines discussed in the General and Operating System Interface sections perform operations on files. For a full discussion of files and file management, you should refer to manuals describing the operating system that you are using. The remainder of this section sketches file management through DG/L.

CHAIN, OPEN, APPEND and ACCESS are the only built-in routines that can refer to files that are not currently open. You can use CHAIN to transfer from a DG/L program and its environment to another save file (under RDOS or DOS) or program file (under AOS or AOS/VS). OPEN and APPEND open files. These routines take a filename and a file number as arguments, and associate the file number with the named file. You use ACCESS only with Cache Memory Management, which is discussed separately in Chapter 10.

Once you have opened a file, you can use several routines to read data from a file or to write data to it. The detailed differences between routines are discussed with the individual routines.

BYTEREAD and BYTEWRITE transfer a specified number of bytes to or from a file. Use FORMAT, OUTPUT and WRITE to transfer data to a file in various formatted fashions: WRITE is the simplest, and FORMAT and OUTPUT provide more sophisticated output format control. READ lets you read formatted information from a file. LINEREAD and LINEWRITE transfer single lines (which may have varying numbers of bytes) of data to and from files.

Various routines let you perform other file operations. CLOSE lets you explicitly close a file (which is not usually necessary, since open files are automatically closed when a DG/L program terminates). DELETE lets you delete a disk file. FILEPOSITION lets you determine the current file point position relative to the beginning of the file, and POSITION lets you modify that position. FILESIZE lets you determine the current size of a file.

Certain files have special names and special meanings. Under RDOS, DOS, and RTOS, terminal input comes from file $TTI and terminal output goes to $TTO. Under AOS and AOS/VS, you refer to terminal input and output through a single name, @CONSOLE. You can avoid operating system-dependent programming by using two procedures in the DG/L runtime library, GETCINPUT and GETCOUTPUT. To use these procedures, declare:

```
EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
```

and open the channels with

```
OPEN (0, (GETCINPUT));
OPEN (1, (GETCOUTPUT));
```

The sample programs in this manual use this format. Other special filenames are used to communicate with devices such as printers and tape equipment.

Your DG/L programs can read command lines typed into the CLI, using the COMARG routine. It isn't necessary to open any files to use COMARG under AOS and AOS/VS. Under RDOS, DOS, and RTOS, however, you must open COM.CM (if running in the background) or FCOM.CM (if operating in the foreground). As with terminal input and output, you can use a runtime procedure to gain program portability: in this case, the procedure is NAMEGROUND. If you declare

```
EXTERNAL STRING PROCEDURE NAMEGROUND;
```

and then follow the declaration with

```
OPEN (2, NAMEGROUND ("COM.CM"));
```

the command file will be opened as unit 2 in either RDOS environment.

ALLOCATE

General Routine

Allocate a contiguous core block of words.

Format

ALLOCATE (point, size*/, [flag] err]*);

Point is a pointer. Size is an integer (1) if *flag* is not present. If *flag* is present, size is integer (2) when bit 9 in *flag* is 1, and integer (1) otherwise.

Flag requires the presence of *err*. The format of *flag* is an inclusive OR of the fields

- bits 12-15 (1R88-17R8). This alignment factor ensures that the block returned aligns on at least a 2^n boundary.
- bit 10 (40R8). The program zeroes this block before returning if this bit is 1.
- bit 9 (100R8). If this bit is 1, the size field is a double-word quantity (integer (2)); otherwise it is a single-word quantity (integer (1)). All code except 32-bit AOS/VS ignores this bit.
- bit 8 (200R8). If this bit is 1, there is no search of the free list. Instead, it takes core from the top of the stack.

The flag argument defaults to 40R8: no alignment, zeroed block, single-word size field, and free list search.

Err is an optional error label.

Description

Use this routine to allocate a contiguous core block integer-expr words long, and return its address in pointer. If the integer-expr is nonpositive, or if there is not enough available memory to fulfill the allocation request, you generate a fatal error. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors are fatal.

ALLOCATE (continued)

Remarks

1. Memory you allocate by calling ALLOCATE is not freed until you issue a call to FREE with the associated pointer.

2. If, in connection with ALLOCATE, you declare any of the standard routines

```
EXTERNAL INTEGER BESTFIT;
EXTERNAL INTEGER EXACTFIT;
EXTERNAL INTEGER FIRSTFIT;
```

   you can specify which method the local allocation routines should use to manage the free chain, and whether to merge adjacent free blocks or not. BESTFIT looks for the smallest block in the free chain to fit the size and alignment requirements. FIRSTFIT looks for the first block to fit the size and alignment requirements. Both BESTFIT and FIRSTFIT split a free block if the block is much larger than the request, and merge adjacent free blocks when you release the block. EXACTFIT, the default, searches the free chain for blocks that meet the alignment request and equal exactly the size requested. All three routines adjust the stack limit down to allocate a block if the requested size or alignment does not occur in the free chain. The cost, above the size of the block allocated, is

Method	16-bit code	32-bit code
BESTFIT	2 words	4 words
FIRSTFIT	2 words	4 words
EXACTFIT	1 word	4 words

Examples

```
ALLOCATE (PTR, 400);
```

allocates a 400-word block of memory to be used in pointer expressions, and puts its address in PTR.

```
ALLOCATE (PTR, 32000, ELABEL);
```

probably causes a transfer to ELABEL, since it is highly unlikely that 32000 contiguous words of memory is available (in a 16-bit environment).

APPEND

Interface Routine

Open a file as a filenumber and set fileposition to end of file.

Format

APPEND (filenumber,string-expr */,err/*);

Description

This routine directs the operating system to open the file named **string-expr** as **filenumber**, and set the current fileposition to the end of that file. The file named **string-expr** must not be open, and **filenumber** must not be in use when you try to APPEND, or an error will occur. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, APPEND creates an empty random file and appends it. If you omit *err*, any other errors are fatal. Initial fileposition is at the end of the file.

Remarks

1. If the *err* field is included, and the file can't be appended, the calling program will branch to the *err* location. No new file will be created.
2. APPEND generates no automatic leading form feeds under RDOS, even if the file being appended is a character device. See also OPEN.
3. APPEND sets the initial fileposition at the end of the file. Subsequently, you may reposition, read from, or write to the file.

Example

```
APPEND (1, "MT0:0", IFERR);
```

opens RDOS file MT0:0, and positions the tape at the end of MT0:0. If errors occur, the program will branch to IFERR.

BYTEREAD

Interface Routine

Read a number of bytes from filenumber into memory.

Format

BYTEREAD (filenumber,pointer,integer-var *[,err]*);

Description

This routine reads integer-var bytes from the filenumber, which must already be open, into the memory area pointed to by pointer. BYTEREAD returns with integer-var set to the number of bytes actually read, even if an error occurs. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors, except end of file, are fatal.

Remarks

1. In the 32-bit environment, the pointer to the area must be a double word.
2. Integer-var should not be a constant, since BYTEREAD returns with integer-var set to the actual number of bytes read if an error (most likely, an end of file) prevents reading the full number specified.
3. If you use BYTEREAD to read into a string, you should follow it with a call to SETCURRENT to set the length of the string to the number of bytes actually read, as in

```
BYTEREAD(1,ADDRESS(CSTR),BCNT);
SETCURRENT(CSTR,BCNT);
```

Example

```
BYTEREAD (0, ADDRESS(STR), COUNT, ERR);
```

reads COUNT bytes from file 0 to the memory area STR, unless an error prevents BYTEREAD from doing so.

BYTEWRITE *Interface Routine*

Write a number of bytes from memory to filenumber.

Format

BYTEWRITE (filenumber, pointer, integer-expr, *[err]*);

Description

This routine writes integer-expr bytes to filenumber, which must already be open, from the memory area pointed to by pointer. Integer-expr receives the number of bytes actually written, if an error prevents BYTEWRITE from writing the full integer-expr number of bytes. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any error is fatal.

Remarks

1. In the 32-bit environment, the pointer to the area to be written must be a double word.
2. Since integer-expr is modified to the number of bytes actually written if an error occurs in the write process, integer-expr should not be a constant.

Example

```
BYTEWRITE (1, ADDRESS(STR), I := LENGTH (STR));
```

writes the bytes of string STR to filenumber 1. Since no provision is made for errors, all errors generate fatal error messages. If an error occurs in this example, the actual byte count will not be lost, since LENGTH(STR) uses I to pass and return the length. You can avoid losing this value if, as in the example, you use an integer variable instead of a function clause for integer-expr.

CHAIN

General Routine

Terminate the current process and execute the program file.

Format

CHAIN (filename, *[,integer-expr [,err]]*);

Integer-expr under AOS and AOS/VS is a pointer to an interprocess communication packet (IPC) that defines the command line passed to the program.

Description

This routine stops the current program, and loads and starts execution of another program (under RDOS, save) file from disk. The filename must include any extension. You pass a datum to the program or save file with *integer-expr*. If errors occur in the transfer process, and you specify *err*, the calling program branches to *err*. This is the only condition for returning to the calling program after a CHAIN call.

Remarks

1. Filename must be available as a string at runtime: it may be any string expression.
2. If you pass *integer-expr* to a procedure with one or more arguments, DG/L places the datum into the first argument position.
3. If you call CHAIN with only two arguments, the second is assumed to be *integer-expr*, not *err*.
4. RDOS save filenames conventionally have .SV extensions; AOS and AOS/VS program files conventionally have .PR extensions.

Example

```
CHAIN ("PROG.SV", 5, ELABEL);
```

causes the save file PROG.SV to be executed. PROG.SV receives the value 5 as a datum. If the transfer fails, the calling program goes to ELABEL.

CLOSE

Interface Routine

Close filenumber.

Format

CLOSE (filenumber, *[,err]*);

Description

This routine directs the operating system to close **filenumber**, which must currently be open. Errors, such as those caused by the file not being open, cause a branch to *err*, or a fatal message if you do not specify *err*.

Remarks

1. You do not need to close files expressly before program termination, since the operating system automatically closes them for you.
2. Under RDOS (but not AOS or AOS/VS), you must close a file before you can delete it.

Example

```
CLOSE (0);
```

closes file 0, with no error transfer location.

COMARG

General Routine

Read a command line from the CLI.

Formats

COMARG (filenumber,string-var *[,err]*);
COMARG (filenumber,string-var *[,[,boolean-array] err]*);

Description

This routine reads a command line from the CLI. The command file must be open as **filenumber**, if you are running under RDOS, and the command is read into **string-var**. *Boolean-array* must have at least 26 elements. The first 26 will receive the values of each of the 26 possible switches set in the command line. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors are fatal.

Remarks

1. *Boolean-array* must have at least 26 elements. You can specify it only if you use all four argument positions, including *err*. Positions in *boolean-array* correspond to set switches alphabetically; for example, a /A in the command line would set array element 1 to TRUE (a numeric 1), a /B would set element 2 to TRUE, and so on.

2. Before using COMARG under RDOS, you must explicitly open the command file, "COM.CM" (or "FCOM.CM" in the foreground environment), as **filenumber**. The first call to COMARG returns the name of your DG/L program, complete with its accompanying global switches. Successive calls to COMARG return successive arguments, with their associated switches. Under AOS, you need no open command file to use COMARG. If you supply a command file number in an AOS COMARG call, it is ignored.

3. If a call to COMARG generates an end-of-file condition, this means that the last CLI command line argument has already been read.

Example

```
COMARG(0,RECEIVER,BOOLMAT,ELAB);
```

reads the next command from file 0. The command is stored in the string RECEIVER, and Boolean elements BOOLMAT[1] through BOOLMAT[26] are set conditionally, depending on the switches set in the command line. Errors cause transfer to ELAB.

DELETE

Interface Routine

Delete a file from disk.

Format

DELETE (string-expr *[,err]*);

Description

This routine deletes the file named string-expr from disk. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors are ignored.

Remarks

1. Under RDOS (but not AOS or AOS/VS), you must close a file before you can delete it.
2. If an error occurs while trying to DELETE a file, and *err* is specified, the calling program branches to *err*. Otherwise, errors are ignored, and the calling program continues with its next statement.

Example

```
DELETE ("OLDFILE");
```

ERRINTERCEPT *General Routine*

Establish a procedure for intercepting certain errors.

Format

ERRINTERCEPT(*[procedure, datum, mode, [[,error code, [error mask]] label]]*);

Description

The ERRINTERCEPT routine invokes your defined procedure when it encounters an error of the specified class. The routine is sensitive to block structure, so you may redefine intercepts at lower block levels, or replace intercepts in the same block level.

ERRINTERCEPT without arguments removes any active ERRINTERCEPT.

Procedure is the name of the handling procedure. You must declare the procedure either globally or locally to the ERRINTERCEPT call. Declare the procedure in the format

```
PROCEDURE procedure (datum [,error label]);
  procedure body
```

where ERRINTERCEPT passes datum and *error label.*

Datum is an integer or integer variable (but not an expression) whose value passes to the procedure. It is a 16-bit quantity in all environments.

Mode is an integer whose bit values define the manner of error handling: printing, severity, type, and code for the procedure to handle. That is, if you specify an *error label* and *error code,* the routine ignores the code (bits 8 through 15) in mode. See Table 9-1 for details. *Mode* is a 16-bit quantity in all environments. If the bottom 8 bits of mode are 377R8, -1 is used both for *error code* and *error mask.* If the bottom 8 bits of mode are not 377R8, then the bottom 8 bits of mode are used for *error code* and 777R8 is used for *error mask.*

Error code is a valid member of the class of errors you wish to specify. It is a 32-bit quantity in the 32-bit environment, and a 16-bit quantity elsewhere.

Error mask specifies which bits to retain in *error code* and in the actual error encountered when trying for a match, permitting the procedure call for any of a class of errors. Error mask is a 32-bit quantity in the 32-bit environment, and a 16-bit quantity elsewhere. If you specify five arguments for ERRINTERCEPT, -1 is used for *error mask.*

Error label is a statement label that you pass to the procedure.

An error report (with no label passed) triggers a search of the chain of ERRINTERCEPT and ERRTRAP calls, from most to least recent, looking for one that handles both its severity and type. When the search finds a match for both severity and type, it checks *error code.* If *error code* is -1, it calls the defining ERRINTERCEPT routine or ERRTRAP label. If *error code* is not -1, it calls the defining ERRINTERCEPT routine or ERRTRAP label if the result of doing an AND on *error code* and *error mask* matches the result of doing an AND on *error mask* and the actual error. Otherwise the search continues until it either exhausts the list of ERRINTERCEPT and ERRTRAP calls or finds one to handle the error.

Examples

```
ERRINTERCEPT (MYERRS,0,72377R8);
        /*INTERCEPT USER-DEFINED ERRORS WITH "MYERRS"*/

ERRINTERCEPT (SYSERRS,0,74777R8);
        /*INTERCEPT SYSTEM ERRORS WITH "SYSERRS"*/
```

Table 9-1. Error Handling Modes

Code	Octal Equivalent	Meaning
	Printing:	
0B0	000000R8	Suppress printing of error message.
1B0	100000R8	Print error message before handling.
	Severity:	
1B1	040000R8	Handle errors fatal to process.
1B2	020000R8	Handle errors fatal to task.
1B3	010000R8	Handle nonfatal errors.
	Type:	
1B4	004000R8	Handle system errors.
1B5	002000R8	Handle user error codes.
1B6	001000R8	Handle user error text.
1B7	000400R8	Handle DG/L runtime errors.
	Code:	
NOTE: Bits 8 through 15 either specify an error code for the procedure or system to handle, or give 377R8 for handling any code.		

Create a message for the error handler to write.

Format

ERROR (string-variable);

Description

This routine writes a fatal error message to the terminal and returns you to the operating system. The message to be written comes from **string-variable**. If you create an error message at runtime, you must append a null ("<NUL>") to the string.

You must not specify **string-variable** with SUBSTR. You may use instead a literal string.

The error signalled is a "User error text" type of error.

Examples

```
ERROR ("ILLEGAL DATA FORMAT!")
```

would print the quoted message and return you to the operating system.

```
        ALLOCATE (PTR, 32000, ELAB);
ELAB: ERROR ("INSUFFICIENT MEMORY <NL>");
```

would branch from the ALLOCATE to ELAB if an error occurred in ALLOCATE. Then, ERROR would be called.

```
ERROR("CANNOT START TASK "!!NUM!!".<NUL>");
```

You create an error message at runtime, which requires a final null.

ERRTRAP

General Routine

Establish a label as a location for trapping errors.

Format

ERRTRAP(*[error label, mode, [error code [,error mask]]]*);

Description

This routine transfers control to the ERRTRAP label when it encounters an error of the specified class. The routine is sensitive to block structure, so you can redefine intercepts at lower block levels, or replace intercepts in the same block level.

ERRTRAP without arguments removes any active ERRTRAP established in the current block. It does not carry over to surrounding blocks.

Error label is a statement label to which control transfers when an error occurs.

Mode is an integer whose bit values define the manner of error handling: printing, severity, type, and code for the procedure to handle. If you specify an *error label* and *error code,* the routine ignores the code (bits 8 through 15) in mode. See Table 9-1 for details. *Mode* is a 16-bit quantity in all environments. If the bottom 8 bits of *mode* are 377R8, -1 is used both for *error code* and *error mask.* If the bottom 8 bits of *mode* are not 377R8, then the bottom 8 bits of *mode* are used for *error code* and 777R8 is used for *error mask.*

Error code is a valid member of the class of errors you wish to specify. It is a 32-bit quantity in the 32-bit environment, and a 16-bit quantity elsewhere.

Error mask specifies which bits to retain in error code and in the actual error encountered when trying for a match, permitting the procedure call for any of a class of errors. *Error mask* is a 32-bit quantity in the 32-bit environment, and a 16-bit quantity elsewhere. If you specify three arguments for ERRTRAP, -1 is used for *error mask.*

An error report (with no label passed) triggers a search of the chain of ERRINTERCEPT and ERRTRAP calls, from most to least recent, looking for one that handles both its severity and type. When the search finds a match for both severity and type, it checks *error code.* If *error code* is -1, it calls the defining ERRINTERCEPT routine or ERRTRAP label. If *error code* is not -1, it calls the defining ERRINTERCEPT routine or ERRTRAP label if the result of doing an AND on *error code* and *error mask* matches the result of doing an AND on *error mask* and the actual error. Otherwise the search continues until it either exhausts the list of ERRINTERCEPT and ERRTRAP calls or finds one to handle the error.

Example

```
ERRTRAP(EOF,74030R8)
/*TRAPS ALL SEVERITY LEVELS (4+2+1) OF SYSTEM ERRORS (4)
WITH "END OF FILE" (30) CODE*/;
```

FILEPOSITION *Interface Routine*

Get the current byte position in filenumber.

Format

FILEPOSITION (filenumber,position */,err/*);

Position is a double-word aggregate.

Description

This routine gets the current byte position in the file that is open as filenumber, and returns the result in position. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any error is fatal.

Remark

A two-word aggregate can be any two consecutive words of memory. For example, it can be a double-precision integer, two-element integer array, two words allocated to a pointer, or a single-precision real number. However, the format will always be the same as a double-precision integer.

Example

```
FILEPOSITION (1, DI, ERRLBL);
```

gives DI a value equal to the current byte position of file 1.

FILESIZE

Interface Routine

Store the size of filenumber.

Format

FILESIZE (filenumber,byte_count */,err/*);

where byte_count is a double-word aggregate.

Description

This routine gets the size of filenumber and stores the size in byte_count. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any error is fatal.

Remarks

A two-word aggregate can be any two contiguous words in memory. For example, it can be a double-precision integer, a two-element integer array, or a two-word area allocated to a pointer. However, the format is always the same as a double-precision integer.

Example

```
FILESIZE (0, DI, IFERR);
```

finds the size of file 0, and stores that value in the two words described by DI. If errors occur, they cause a branch to IFERR.

FORMAT

Interface Routine

Specify editing features when writing to filename.

Format

FORMAT (filenumber,form*[,list][,err]*);

Filenumber is the number of an open file, form is a string specifying the output format, and *list* is an optional list of variables to be written out according to the specified format.

Description

FORMAT lets you write the list data to the output file in flexible fashion, rather than with the default formats associated with the WRITE routine. FORMAT provides built-in editing features, different from those available with the OUTPUT routine. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors will be fatal.

Remarks

1. The form string must be a quoted string in the FORMAT statement, a string literal, or a string variable containing data ending with a null character.

2. The formatter interprets three characters, #, &, and %, as editing characters.

   - The # character causes the datum to be output in signed decimal notation, without leading zeros.
   - The & character causes the datum to be output as an unsigned octal value, with leading zeros.
   - The % character causes the datum to be output as a signed octal value, with no leading zeros.

   With all editing characters, strings are printed to their full length, Booleans are printed as TRUE or FALSE, and reals are always printed as signed decimal notation without leading zeros.

3. Since all output data is retained in its full precision by FORMAT, you need use only one editing character to format an entire datum. Table 9-2 shows the sizes of output fields for each possible data type and editing character; N represents the number of character positions needed to print the value without leading zeros, as WRITE would do. Generally, numbers are printed in the most compact form that will fully represent their values.

Table 9-2. Field Lengths

Data Types	Editing Characters		
	#	&	%
INTEGER	N	6	N
INTEGER (2)	N	11	N
REAL	N	N	N
REAL (4)	N	N	N
POINTER	N	6	N
BOOLEAN	N	N	N
STRING	N	N	N
BIT	N	N	N

4. You can use FORMAT to output more data items than there are editing characters in the formatting string. FORMAT works as follows: When FORMAT is ready to write out an item, FORMAT looks for the next editing character in the format string. If the format string ends before one is found, FORMAT goes back to the format string's beginning and continues its search, outputting nonediting characters as it searches. When FORMAT finds the character, it outputs the data item according to the specified format. Keeping this in mind, you could also write the example as

```
FORMAT (1,"#<HT>#<NL>", U(1), T(1), U(2),
T(2));
```

5. You can output entire arrays with a single FORMAT statement. The elements of the array are output one at a time, using the format string as described in remark 3. Using this feature, you could output a 4-by-5 array as 4 rows of 5 values with:

```
INTEGER ARRAY [3,4] IRVING;
.
.
.
FORMAT (1,"#<HT>#<HT>#<HT>#<HT>#<NL>", IRVING);
```

6. Both the *list* and the *err* are optional arguments. If you give FORMAT three arguments, the compiler decides whether the third supplied argument is a list or an err from context.

Example

```
FORMAT (1, "UNIT: <HT>TOTAL:<NL><NL>");
FORMAT (1, "#<HT>#<NL>#<HT>#<NL>", UA(1),TA(1),UA(2),TA(2));
```

generates output with the format

```
UNIT:       TOTAL:
4328        1
33          4068
```

FREE

General Routine

Free a previously allocated area.

Format

FREE (pointer, *[err]*);

Description

This routine frees the previously allocated section of memory pointed to by pointer. FREE causes an error return if it is given a pointer to a block of memory that is not freed. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors are fatal.

Remarks

1. In the 32-bit environment, the pointer is a double-word quantity.
2. FREE places the memory block on a linked list, available for future allocation. The block is returned to the stack, if possible.
3. Contiguous freed blocks are not merged together into larger blocks, unless you specify a nondefault allocation method (see the ALLOCATE routine).

Example

```
FREE (PTR);
```

frees the block of memory pointed to by PTR.

GTIME

General Routine

Get the current time.

Format

```
GTIME (int1,int2,int3,int4,int5,int6);
```

Description

This routine gets the current time from the system clock, setting the argument values as

year	=	int1
month	=	int2
day	=	int3
hour	=	int4
minute	=	int5
second	=	int6

Remarks

The year value int1 is returned less the century; 1976 would appear as 76.

See also STIME.

Example

```
GTIME (YEAR,MONTH,DAY,HOUR,MINUTE,SECOND);
```

sets the integer variables as indicated.

LINEREAD

Interface Routine

Read a line or number of bytes of data from filenumber into memory.

Formats

LINEREAD (filenumber,pointer,integer-var*[,err]*);
LINEREAD (filenumber,pointer,integer-var,limit-count,err);

Description

In the first format, this routine reads a line or integer-var bytes of data from filenumber into the area of memory pointed to by pointer. The byte count of the line read is returned in integer-var. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors are fatal.

In the second format, the integer expression limit-count defines a maximum count for LINEREAD to use. If the routine does not reach a line delimiter before that number of characters, it terminates the read and goes to err.

Remarks

1. In the 32-bit environment, the buffer address is a double-word quantity.
2. A line is terminated by a carriage return, NEW LINE, form feed, or a null character. Under AOS, <LF> or <NL> (octal 12) will also be treated as a line terminator. A line containing one of these termination characters, and nothing else, returns a byte count of 1.
3. Integer-var should not be a constant, since its value will be replaced by the returned byte count.
4. If you use LINEREAD to read data into a string, be sure to follow it with a call to SETCURRENT to set the current length of the string correctly. For example,

   ```
   LINEREAD(0, ADDRESS(STR), J);
   ```

   should be followed by

   ```
   SETCURRENT(STR,J);
   ```

5. RDOS ignores limit-count. Its functionality is .RDL (limit 133 characters).
6. The err label is not optional after limit-count.

Example

```
LINEREAD (0, ADDRESS(DATA),I,IFERR);
```

reads one line of data from file 0 into the buffer DATA. For details of the mechanism used to get the buffer's address, see the ADDRESS function. When the routine is executed, variable I receives number of bytes in the line. If errors occur, the calling program branches to IFERR.

LINEWRITE *Interface Routine*

Write a line or number of bytes of data from memory to filenumber.

Formats

LINEWRITE (filenumber,pointer,integer-var*[,err]*);
LINEWRITE (filenumber,pointer,integer-var,limit-count,err);

Description

In the first format, this routine writes a line or integer-var bytes of data from the memory area pointed to by pointer out to filenumber, which must be open. The number of bytes actually written is returned in integer-var. The *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any error is fatal.

In the second format, limit-count returns the maximum number of characters for LINEWRITE to write out. If LINEWRITE does not encounter a line delimiter before that number of characters, it goes to err; otherwise it gives a normal return.

Remarks

1. In the 32-bit environment, the buffer address is a double-word quantity.
2. You must terminate the line with a terminator recognized by the operating system (NULL, carriage return, or form feed on all operating systems; line feed under AOS and AOS/VS). Compiling for AOS and AOS/VS resolves the DG/L character <NL> to a line feed; compiling for RDOS resolves <NL> to a carriage return.
3. Integer-var should not be a constant, since its value is replaced by the byte count.
4. RDOS ignores limit-count. Its functionality is .WRL, a line limit of 133 characters.
5. The err label is not optional following limit-count.

Example

```
LINEWRITE(1,ADDRESS("DONE<NL>"),I,IFERR);
```

writes the line "DONE<NL>" to file number 1. Its byte count (5) is returned in integer-var. If any errors occur, the calling program branches to IFERR.

OPEN

Interface Routine

Open a file as filenumber and set fileposition to zero.

Format

OPEN (filenumber,string-expr/,*err*/);

Description

This routine opens the file named string-expr as filenumber. If the attempt fails, the calling program branches to label *err*. If either the file named string-expr is open or filenumber is in use when you attempt the OPEN, an error occurs. Initial fileposition is zero.

Remarks

1. If *err* is not specified and the file doesn't exist, OPEN first tries to create the file and then open it. Any errors following this attempt are fatal. Any errors other than "file does not exist" are immediately fatal.

2. Since the initial fileposition is at the beginning of the file, sequentially writing to the file overwrites any data already there.

3. If the file is a character device (such as a printer), OPEN automatically sends a form feed to the device. To avoid this form feed, use the APPEND routine instead of OPEN.

4. In order to get input from and send output to the terminal, you may declare the built-in EXTERNAL procedures

   ```
   EXTERNAL STRING PROCEDURE GETCINPUT, GETCOUTPUT;
   ```

   and then set up input and output channels with

   ```
   OPEN (0, (GETCINPUT));
   OPEN (1, (GETCOUTPUT));
   ```

   The interior parentheses are obligatory.

   These input and output procedures resolve to the default input and output files under RDOS, AOS, and AOS/VS.

Operating system	GETCINPUT	GETCOUTPUT
AOS	@INPUT	@OUTPUT
AOS/VS	@INPUT	@OUTPUT
RDOS foreground	$TTI1	$TTO1
RDOS background	$TTI	$TTO

Example

```
OPEN (1,"FILEA.DT",IFERR);
```

opens the (existing) file named "FILEA.DT" under file number 1, branching to IFERR if the OPEN is not possible.

OUTPUT *Interface Routine*

Output data to filenumber in an explicitly defined format.

Format

OUTPUT (filenumber,form*[,list] [,err]*);

Filenumber is the number of an open file, form is a string specifying the output format, and *list* is a list of variables to be output according to the specified format.

Description

This routine allows you to output the list data to filenumber with an explicitly specified format, rather than with the default values used by the WRITE routine. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err* any errors are fatal.

Remarks

1. Form must be specified either as a quoted string in the OUTPUT statement or as a string terminated with a null character. DG/L string literals always terminate with nulls.

2. The character # represents one digit position in the output field.

3. You may use a radix indicator as part of a format specification for integers, for example #R2. The radix indicator is not printed, but is used to format the data. You can specify any integer in the range 2-10. If you specify no radix, the default is radix 10 (decimal).

4. You may include a decimal point in a format specification for floating-point numbers. In this case, the number of # signs to the right of the format field decimal point determines the number of positions to the right of the number's decimal point that will be displayed. The values will be rounded to as many places as specified by the output field specification. For example:

   ```
   OUTPUT(1,"###.#",W,X);
   ```

5. You may also put a sign on a format field for floating-point numbers. The formatter interprets signs, or the lack thereof, as follows:

   - No sign: Positive numbers are printed without signs, but negative numbers are printed with - signs before them. The minus sign requires a digit position, so format field ###.### can print only values from -99.999 to 999.999.
   - + sign: Both positive and negative numbers are printed with appropriate signs. The signs require no field positions: +###.### can print any quantity between -999.999 and 999.999.
   - - sign: Negative numbers are printed with sign, positive numbers without sign, but with a space before the number. However, the sign requires no field position: -###.### can print all values from -999.999 to 999.999.

6. You may include an E in format fields for floating-point numbers to direct that their values be printed using exponential notation. # signs following the E determine the number of fields reserved for the exponent. If the exponent is not as big as the field reserved for it, it is right-justified, as in

   ```
   OUTPUT(1,"-#####.##E##",A,B);
   ```

   which would generate

   ```
   12345.25E-4        99.04E  0
   ```

   as output.

7. If you use OUTPUT to print strings, they are left-justified in their output fields, but are always printed in full, even if they are longer than the format field. To take advantage of this feature for numeric data items, convert them to strings.

8. You may include character representations within angle brackets in format specifications to send carriage control and other characters. You may use the octal integer ASCII equivalent of a character, or certain abbreviations (see Chapter 2 for a listing).

9. To print all of the elements of an array, give OUTPUT the array identifier, omitting subscripts.

10. If a numeric data item is too large to fit in the specified field (overflow), then an asterisk (*) is printed, followed by the data item in exponential format. For example, 453.75638 cannot be printed in format field ##.###, so it appears as *4.5E02. In other words, it appears with as many digits as will fit and still leave room for the * and the Enn notation.

OUTPUT (continued)

Example

Figure 9-1 shows how the OUTPUT routine works and what its generated output looks like.

```
 1 BEGIN INTEGER I, J;
 2   REAL ARRAY RARR[4,4];
 3
 4   OPEN (1, "AFILE");
 5
 6   OUTPUT(1,"THIS IS THE FIRST CALL TO OUTPUT<NL><NL>");
 7
 8   I := 3462;
 9   OUTPUT(1,"NOTE WHAT HAPPENS WITH OVERFLOW:<NL>
10   ##<HT>###<HT>####<HT><NL><NL>",I,I,I);
11
12   OUTPUT(1,"<NL>");
13   FOR J := 1 STEP 5 UNTIL 100 DO BEGIN
14     FOR I := J STEP 1 UNTIL J+4 DO
15     OUTPUT(1,"####",I+J);
16     OUTPUT(1,"<NL>");
17     END;
18
19   FOR J := 0 STEP 1 UNTIL 4 DO
20     FOR I := 4 STEP -1 UNTIL 0 DO
21     RARR[I,J] := (FLOAT(I)/4)*J;
22
23   OUTPUT(1,"<NL>  THE ARRAY IS:");
24   OUTPUT(1,
25     "<NL>-###.###<HT>-###.###<HT>-###.###<HT>-###.###<HT>-###.###<HT>",
26     RARR);
27 END;
```

Figure 9-1. OUTPUT Routine

The program in Figure 9-1 generates the contents of Figure 9-2 on file AFILE.

```
THIS IS THE FIRST CALL TO OUTPUT

NOTE WHAT HAPPENS WITH OVERFLOW:
  *3E3  *3E3    3462


  2   3   4   5   6
 12  13  14  15  16
 22  23  24  25  26
 32  33  34  35  36
 42  43  44  45  46
 52  53  54  55  56
 62  63  64  65  66
 72  73  74  75  76
 82  83  84  85  86
 92  93  94  95  96
102 103 104 105 106
112 113 114 115 116
122 123 124 125 126
132 133 134 135 136
142 143 144 145 146
152 153 154 155 156
162 163 164 165 166
172 173 174 175 176
182 183 184 185 186
192 193 194 195 196

 THE ARRAY IS:
  0.000     0.000     0.000     0.000     0.000
  0.000     0.250     0.500     0.750     1.000
  0.000     0.500     1.000     1.500     2.000
  0.000     0.750     1.500     2.250     3.000
  0.000     1.000     2.000     3.000     4.000
```

Figure 9-2. Output

This program uses the OUTPUT routine several times. It begins by declaring two integer variables, I and J, and a 5-by-5 element real array, RARR. At line 4, it opens file AFILE, which it will use for output. Line 6 calls OUTPUT to send a message, followed by NEW LINE characters. Line 8 assigns a value to integer I, and the OUTPUT routine prints it, with three different format specifications. The first two aren't big enough to display all four digits of the number, so OUTPUT prints an asterisk followed by an order-of-magnitude approximation of I. See Remark 10 for discussion of OUTPUT's actions on overflow.

Line 12 sends an extra NEW LINE character, and lines 13-17 generate and print a table. Note that one NEW LINE is sent each time J is given a new value in the outer FOR loop.

Lines 19-21 set up values in RARR. Line 23 prints a header, and then a NEW LINE. Lines 24-26 print the entire array, with one call to the OUTPUT routine. The format specifier directs that 5 values be printed per line with 3 positions on either side of the decimal point. Leading zeros aren't printed, but there are spaces in their positions. The - signs before each format field direct that only negative values be printed with signs. Line 27 ends the program.

POSITION

Interface Routine

Reset the fileposition for filenumber.

Format

POSITION (filenumber,arg2*[,err]*);

Arg2 is a two-word aggregate (see Remarks).

Description

This routine resets the position of open filenumber to a new position contained in arg2. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors will be fatal.

Remarks

A two-word aggregate can be any two contiguous words in memory. For example, it can be a double-precision integer, a two-element, single-precision integer array, or two words allocated to a pointer. However, the format is always the same as a double-precision integer.

Example

```
POSTION(0,DI+200,IFERR);
```

Sets the position of file 0 to position DI+200. (The position is specified in bytes.) If errors occur, the calling program branches to IFERR.

READ

Interface Routine

Read data from an input file to program variables.

Format

READ (filenumber, list*[,err]*);

Filenumber is an open filenumber, and list is a list of variables to be read in from filenumber.

Description

This routine lets you read data into a DG/L program from an outside file. It reads ASCII characters, and converts them to the appropriate target data type. You must follow each value in the file by a delimiter. The format used in reading depends on the data types of the variables to be read in. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors are fatal.

Remarks

1. Input data is read in accordance with the input variable types. Delimiters for numeric data are commas, semicolons, carriage returns, or NEW LINEs, in any combination. Line terminators are carriage returns or NEW LINEs. Where space-dlm is any number of spaces or tabs, READ reads numeric data as

   space-dlm number space-dlm delimiter

   and string data as either

   space-dlm "any-string" space-dlm delimiter

   or

   space-dlm string line-terminator

2. By default, the delimiter for string reads is a carriage return, NEW LINE, or null character. If you wish to use spaces, tabs, commas, and semicolons as additional delimiters, you must modify the EXTERNAL word to which .SPREAD (defined in file DGLPARAM.SR) points to a nonzero value.

3. If you wish to include line terminators in a string variable, use the quoted format (first string format pictured above).

4. If you put an unsubscripted array name in the list, READ reads sequentially, filling every element of the array, stopping only when it has read every array element.

5. If you give the READ routine more data items as input than are specified in its call, any extra items are ignored, not carried on to subsequent READs.

6. To read input from the terminal to a program variable, see the OPEN routine.

Example

```
READ(0,I,R,A[10],IFERR);
```

reads variables I, R, and A[10] from file 0 according to their declared data types. If errors occur, the calling program branches to IFERR.

READSTRING *Interface Routine*

Put a data-sensitive read into a string variable.

Format

READSTRING (filenumber, stringvar, *[,err]*);

Stringvar is the variable that receives the data.

Description

READSTRING resembles LINEREAD, and performs a data-sensitive read. It takes the maximum length it will read from the string's maximum length and sets the length read to the string's current length. Thus, unlike LINEREAD, no subsequent SETCURRENT call is necessary.

The string argument can also be a substring.

Example

```
READSTRING (3, CALL_IN, IF_ERR);
```

reads a string from filenumber 3 into the variable CALL_IN. The label IF_ERR provides a branch in case an error occurs.

Store the result and remainder of a division.

Format

REM (int1,int2,int3,*[int4]*);

Int1 and int2 are integer expressions, and int3 and int4 are integer variables.

Description

This routine divides (using unsigned arithmetic) int1 by int2, returning the result in int3 and the remainder in *int4*.

Example

```
REM (DATA, 10, J, DIGIT);
```

divides the contents of DATA by 10, using unsigned arithmetic, and puts the quotient into J. The remainder is stored in DIGIT.

RENAME *Interface Routine*

Change the name of a file.

Format

RENAME (string1,string2*[,err]*);

Description

This routine changes the name of the file whose name is in string1 to string2. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors are fatal.

Remark

The file to be renamed, string1, must NOT be currently open, and file string2 must not already exist.

Example

```
RENAME("OLDNAME","NEWNAME");
```

changes the name of file OLDNAME to NEWNAME.

SETCURRENT

General Routine

Set the current length of a character or bit string.

Format

SETCURRENT (string-var,integer-expr);

Description

Sets the current length of string-var to integer-expr. If the new length is greater than the declared maximum length of string-var, a fatal error is generated.

SETCURRENT has no effect on the length of a BASED string.

Example

```
SETCURRENT (STR, LENGTH (STR) - 1);
```

uses the LENGTH function (see Chapter 8) to truncate the last character from string-var.

STIME

General Routine

Set the system clock.

Format

STIME(int1,int2,int3,int4,int5,int6 *[,err]*);

Int1 through int6 are integers or integer expressions.

Description

This routine sets the system clock, as

year	=	int1
month	=	int2
day	=	int3
hour	=	int4
minute	=	int5
second	=	int6

The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err* any errors will be fatal.

Remarks

1. As with GTIME, the year is represented with a base year of 1900; e.g., 1976 is represented as 76.
2. System constraints will not let you set a date before January 1, 1968, or after the year 2099.
3. Under AOS, only a Superuser may use STIME. Other users will receive error returns if they attempt a call to STIME.

Example

```
STIME(76,8,8,17,17,3);
```

sets the system clock to 17:17:03, August 8, 1976.

Store the result and overflow of a multiplication.

Format

UMUL (int1,int2,int3,int4,int5);

Int1 through int3 are integer expressions, and int4 and int5 are integer variables.

Description

This routine does unsigned multiplication of int1 and int2. It then adds int3, places the result in int5, and puts any overflow in int4.

Example

```
UMUL (DATA, 10, DIGIT, OVER, DATA);
```

does an unsigned multiplication of the contents of DATA by 10. DIGIT is added to the result, and the sum is returned to DATA, with overflow put in OVER.

WRITE *Interface Routine*

Output a list to filenumber.

Format

WRITE(filenumber,list*[,err]*);

Filenumber is the number of an open file and list is a list of variables or expressions to be written.

Description

This routine outputs the list values to filenumber using default formats based on the list data types. It writes ASCII characters, with intervening delimiters. The optional *err* field lets you specify a label in your program to branch to if an error occurs. If you omit *err*, any errors are fatal.

Remarks

1. The output list is written as a contiguous set of characters, with no format control characters unless you explicitly specify them.

2. Data output format is based on data type, as follows (see FORMAT for a fuller explanation)

INTEGERS	Brief format, decimal
POINTERS	Brief format, octal
STRINGS	Literally (in full)
REALS	General format
BOOLEANS	As "true" or "false"

   Generally, use WRITE as a simple means of getting data to the outside world without much concern for formatting. If you want explicit control over formatting, use the routines OUTPUT or FORMAT.

3. WRITE is equivalent to

   FORMAT (filenumber,"#",list,*[err]*);

4. To write output from the program to the terminal, see the OPEN routine.

Example

```
WRITE (1,NUM1, "<HT>", NUM2, "<NL>");
```

writes the datum in variable NUM1, a tab, the datum in variable NUM2, and a NEW LINE to file 1.

WRITESTRING *Interface Routine*

Write a string to a file.

Format

WRITESTRING (filenumber, stringvar, *[,err]*);

Stringvar is a variable containing the string you wish to write.

Description

WRITESTRING uses the current length of the string to determine how many characters to write. READSTRING and WRITESTRING are not exactly complementary, since WRITESTRING does a dynamic write on AOS and AOS/VS, and a binary write on RDOS.

The string argument may also be a substring.

Example

```
WRITESTRING (3, PUT_OUT, GO_HERE);
```

writes the string in PUT_OUT to the file open as filenumber 3. The label GO_HERE provides a branch in case an error occurs.

End of Chapter

Chapter 10
Applications Development Aids

This chapter describes Cache Memory Management (CMM), suggested data reference methods, the Include facility, and overlay support.

Cache Memory Management

Cache Memory Management is a DG/L feature for programmers who need to access considerably more computer memory than is physically available. You can think of CMM as a software virtual memory, addressable with logical addresses that point to cache memory elements. A cache memory element is an array of words, and can be any size.

CMM creates additional effective memory space by using disk files and buffers in main memory. The buffers store 256 word blocks, or 1024-word pages for the shared-page cache on AOS and AOS/VS. This block size allows many successive references to data within a given block without need for frequent disk fetches. If you attempt to refer to a CMM address not currently in main memory, CMM writes the Least Recently Used (LRU) block back to disk and reads the newly required block into its buffer area.

You can define more than one cache memory file in a single program. Files are differentiated in usage by their file numbers, and on the disk by their filenames.

CMM Initialization

When you call the buffer allocation routine (BUFFER), it establishes the buffer pool, using the appropriate buffer area parameters. It allocates two areas, the actual buffer area and an area containing buffer descriptors. For example, the call BUFFER(BPT,1000) allocates at most a 1000-word buffer pool and sets the pointer BPT to point to that pool. The area containing the buffer descriptors is always resident in main memory, but you may specify (for mapped RDOS) that the buffers themselves be kept in extended memory. BUFFER returns a pointer to the descriptor area as its result. If you initialize CMM for a mapped system in RDOS, it can also operate on an unmapped system or AOS and AOS/VS. If you initialize it for an unmapped RDOS system or AOS, AOS/VS, it can also operate on a mapped system, but will not take advantage of the map.

After establishing the buffer pool, you initialize the cache memory with the ACCESS routine. This routine associates a file number, a filename and an element size with the buffer pool area. You cannot use a CMM file until you have opened it with ACCESS. Once you have opened it, you can read from it and write to it.

The disk file does not necessarily represent the status of the cache buffers at any given moment, since the contents of a buffer are not ordinarily written to disk until a new block needs the buffer space for storage. If you want the disk file to reflect the cache buffers accurately, use the FLUSH routine. FLUSH writes all modified buffers to the disk. You should use it before terminating any program that uses CMM, or you may lose data.

Element Access (Word Access)

The simplest way to access information in the cache memory is with the routines WORDREAD and WORDWRITE. To use them, specify the file number assigned to the cache memory file, the logical address of the desired element, and a pointer to the area of main memory you want to use for transmission of data. Optionally, you may include a word count in the routine call. If you do not include a word count, the contents of the first word of the element you refer to functions as the number of words

to read, not including the count word. For example, if the first word of an element contained the value 16, and you called WORDREAD without a word count parameter, WORDREAD would read the 16 words immediately following the count word. You may use WORDREAD and WORDWRITE to access multiple elements with one call.

Node Access

You can use a special form of cache memory access on the file opened by ACCESS as file number zero, but only on file zero. This form, node access, lets you access series of words similar in structure to DG/L arrays. Node access resembles WORDREAD and WORDWRITE, but without specified word counts, and with no explicitly specified file numbers. Conventionally, nodes are defined as arrays with a lower bound named MINRES. MINRES has the default value -3 (to allow one word for the node size and two words for user-defined information, such as forward and back pointers) before the storage for the data begins at element 0. You can give MINRES other values in your program by declaring it an external integer and assigning the desired value to it. You can also change its value in the file DGPARAM.SR.

You can give the upper bound of the node any value you want, but the array that corresponds to the node should be declared with MINRES as the lower bound and SIZE + MINRES as the upper bound, where SIZE is one larger than the maximum number of data words to be stored in a node. The extra word is the first word of the array. It has the subscript MINRES and contains the size specification for the node.

To access nodes, use the NODEREAD and NODEWRITE routines. These routines take the logical node element address and the name of the array that contains or will receive the data as their arguments. Since the first word of the array contains the size specification for the node, there is no need to specify the number of words in these routines. If you want to transmit data to or from single words of file 0 instead of a whole node, use the STASH and FETCH routines instead of NODEREAD and NODEWRITE. Both take logical addresses and offsets as arguments, and act as follows: STASH stores one word at the location pointed to by the logical address plus the specified offset. The default logical address is the file's beginning address. Similarly, FETCH reads a word from the location specified by the logical address and offset. STASH and FETCH use MINRES as the node's lower bound, so the value of MINRES determines the value of the offset. You should be aware that if the logical address is explicitly included in a call to STASH or FETCH, logical addresses 0-255 are inaccessible. Refer to the notes under FETCH and STASH for more details about this case.

HASH Access

HASH access lets you follow one call to the CMM package with subsequent non-CMM references to the same buffer. HASHREAD and HASHWRITE are highly efficient for repeated accesses to information in the same buffer area, but they run the risk of losing data if not used very carefully. They are not, therefore, recommended to the beginning CMM user. If you know that two data words reside in the same block (that they have the same block address), and you want to access both of them, you can proceed by

1. reading the first word
2. computing the offset into the block, since you know the block size (256 words) and the logical address of the second word,
3. accessing the second word.

HASHREAD requires its own buffers, and an element size of 1. HASHREAD is called with a file number and logical address, and returns a pointer to the main memory location of the appropriate buffer to find that logical address, as well as an offset within the buffer to the actual word. HASHREAD uses CMM's buffers, and CMM needs to be informed when HASHREAD modifies them. The routine HASHWRITE informs CMM that a buffer has been modified. You call it with the buffer pool pointer value that was returned by BUFFER and used by ACCESS.

When you use HASHREAD and HASHWRITE under virtual memory, it may happen that HASHREAD points to a buffer that has been mapped out of memory, following another CMM access. If there is danger of this occurring, you should execute the HASHBACK routine, which will force the last-mapped buffer back into your address space, making the pointer valid again.

Closing CMM Files

Use the CMCLOSE routine to close explicitly files in buffers. You need not issue a FLUSH routine before CMCLOSE, since CMCLOSE contains an implicit FLUSH.

If you do not use CMCLOSE, you must use the FLUSH routine before terminating a program. The system's ordinary file closing procedures do not provide for writing modified buffers to disk. The FLUSH routine puts an accurate buffer image on disk. FLUSH is also useful during long programs to guarantee disk accuracy should a system failure occur.

Buffer Organization

The following sections sketch the internal structure of cache memory.

The Cache Memory Buffers

Cache memory contains a linked list that consists of one global header for the buffer pool, and a separate local header for each buffer in it. Figure 10-1 sketches the headers.

For RDOS, unshared-page AOS, and unshared-page 16-bit AOS/VS, a 20-word area in memory is allocated for each buffer pool (not for each individual buffer). It contains descriptors and a pointer to the buffer chain. Therefore, CMM requires a minimum of 280 words of buffer space to operate: 260 words for a buffer, and 20 words for the pointer and descriptor area. Subsequent increases to the size of the buffer pool take place in 260-word increments (a buffer at a time).

For 32-bit unshared-page AOS/VS cache, an 8-word area in memory is allocated for each buffer pool (not for each individual buffer). It also contains descriptors and a pointer to the buffer chain. Each individual buffer uses 12 words in addition to the 256 words for the contents of the buffer. Therefore, CMM requires a minimum of 276 words of buffer space to operate: 268 words for a buffer, and 8 words for the pointer and descriptor area. Subsequent increases to the size of the buffer pool take place in 268-word increments (a buffer at a time).

For the shared-page caches, the space for buffer contents comes from unallocated memory, and the headers come from the heap. For AOS and 16-bit AOS/VS, the global header is four words, and the local header is 11. For 32-bit AOS/VS, the global header is eight and the local header is 12 words.

Several CMM files can share the same buffer pool, unless you wish to access them with HASH. However, buffer pool sharing increases access time.

On an unmapped RDOS system, or unshared cache on AOS, or AOS/VS, you define the number of blocks allotted to CMM by declaring the number of words in memory that CMM can use for buffers. CMM calculates the number of buffers that it can allocate based on this specification. On a mapped system, you need only state explicitly how many buffers should be reserved, since memory mapping permits less restrictive use of space.

On a mapped RDOS system, the use of buffers is controlled by two runtime variables, which are used by the initializer. .UMEM represents the number of 1K blocks allotted for your program and the stack. If .UMEM is 0, no mapping will be used. By specifying .UMEM, you can limit the stack, thus allowing more CMM buffer blocks. If you specify .VMIN, you can direct that CMM use mapping when, and only when, enough buffers are available to make mapping useful. To define .UMEM or .VMIN, you must specify them in an assembly language routine, and load that routine along with your DG/L program, or modify them with SEDIT or OEDIT. For more information, consult the parameter file DGLPARAM.SR on the DG/L product tape.

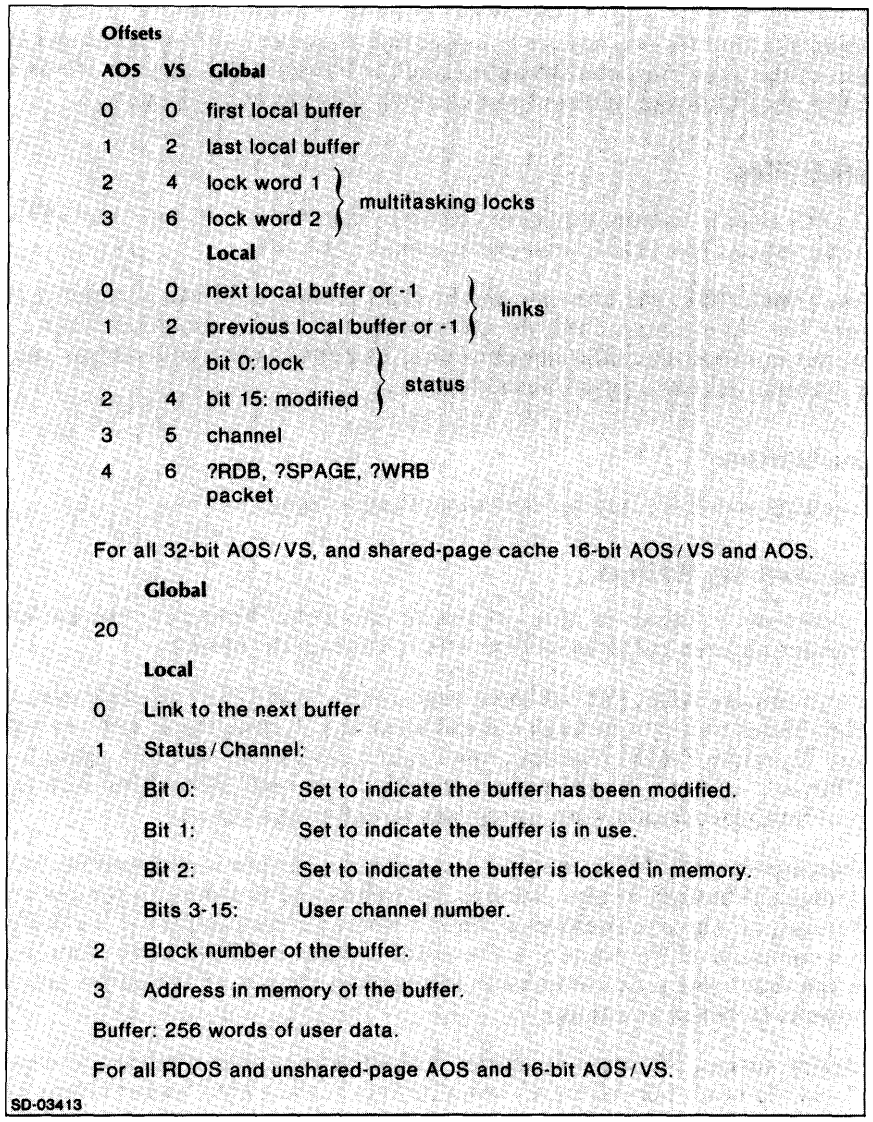


Figure 10-1. Cache Headers

On AOS and AOS/VS, you have the choice of using either shared-page or unshared-page cache. You determine which you want to do by your selection of libraries (the default is unshared cache). You must also call BUFFER with three arguments for shared-page cache. The unshared cache call to BUFFER ignores the third argument for system compatibility, as Table 10-1 indicates.

Table 10-1. Types of Cache Memory

RDOS	RDOS, AOS, AOS/VS	AOS, AOS/VS
Unshared Cache Buffers in Extended Memory	Unshared Cache Buffers in Memory	Shared-page Cache Buffers in Shared Pages
Buffer must have three arguments. User must be running on a mapped system and set up the locations .UMEM .VMIN .VMAX	Buffer has two or three arguments. Default if no options specified, or if running unmapped RDOS	Buffer has three arguments. User must use a special library and set up one of the locations. .VMAX .NMAX
Ignores second argument.	Ignores third argument.	Ignores second argument.
Load with normal libraries: DGLIB DGLIBE DGLIBN	Load with normal libraries: DGLIB DGLIBE DGLIBN DGLIBA DGLIBF DGLIB16 DGLIB32	Load with special libraries: DGLIBAS DGLIB16S DGLIB32S

When you use shared-cache memory on AOS or AOS/VS, the third argument to BUFFER controls the number of buffers. It specifies the maximum number of pages to use as shared pages. To reserve pages for use as shared pages, you must tell the initializer to leave some pages available, instead of allocating everything to the stack and heap. To do this, set the value at location .VMAX to the number of pages you wish to reserve. The initializer leaves that number of pages unallocated. (The location .VMAX is a double-word quantity for 32-bit AOS/VS.) The default for .VMAX is 0. To define .VMAX, you must specify it in an assembly language routine, and load that routine along with your DG/L program, or modify it with FED (AOS/VS) or DEDIT (AOS).

As you allocate more buffers by using BUFFER, less stack space becomes available. Conversely, allocating more buffers reduces the number of disk accesses required to write out least-recently-used blocks and read in newly needed blocks. If you want to access a file randomly, with multiple accesses referring to the same block on the disk, it is desirable to allocate a large buffer pool. If, however, you are going to access a file sequentially, there is little advantage to be gained by providing it with many buffers.

Each call to BUFFER sets up one global header and n local headers and buffers, as Table 10-2 indicates.

Table 10-2. Cache Buffer Sizes

Sizes	RDOS Extended Memory	RDOS Normal	16-bit AOS, Shared	16-bit AOS, Normal	32-Bit AOS/VS Shared	32-Bit AOS/VS Normal
Global header	20	20	4	20	8	8
Local header	4	4	11	4	12	12
Buffer	1024*	256	1024	256	1024	256

* The 1024 words for the buffer reside in extended memory, not in the logical address space.

Reading or Writing Multiple CMM Elements

Using WORD I/O and NODE I/O, you can access multiple CMM elements for reading or writing with a single CMM call. If, for example, you want to create a CMM file where all but a few elements are 5 words long and the remainder are 13 words long, you should call ACCESS with an element size specification of 5. Whenever you call WORDWRITE or NODEWRITE, the word count (or MINRES word) should be set to 5 or 13, depending on which length element you are accessing. CMM will write the appropriate number of words to the CMM file. In the example, writing a 13-word element will actually write three 5-word elements (the length specified in ACCESS). The two logical addresses following the 13-word element address are then unusable, and the two stray words following the element are wasted.

Logical Addresses in CMM

Logical addresses are mapped to physical disk word addresses as follows:

```
physicaladdress := elementsize * logicaladdress;
```

The CMM block's disk address is

```
blocknumber := physicaladdress/256;
```

The offset of the physical address into the block is

```
offset := MOD(physicaladdress,256);
```

Since logical addresses must lie in the range 0-65535 (because they are stored in single words), if you need a database larger than 65536 words, you must use an element size greater than 1, as the table indicates.

File Size (in Words)	Minimum Element Size
0-65K	1
0-131K	2
0-196K	3
0-262K	4

and so on....

Cache Memory Management Example

```
BEGIN

/* CACHE MEMORY EXAMPLE using NODEREAD and NODEWRITE
 *
 * This program reads in strings from the terminal and sorts
 * them, storing the strings in cache memory and building
 * a binary tree.  The node is organized as follows:
 *
 *       -4:    Length of the node      (MINRES)
 *       -3:    Length in characters of the string
 *       -2:    Node pointer to left subtree
 *       -1:    Node pointer to right subtree
 *       0..65: String read in from the terminal
 */


LITERAL LEN             (-4),           /* Node length in words */
  CHAR_LEN              (-3),           /* Length in characters */
  LEFT          (-2),                   /* Left subtree node */
  RIGHT         (-1),                   /* Right subtree node */
  STR                   (0),            /* Start of string datum */

  START_NODE            (400R8),        /* Start of cache nodes */

  INP                   (1),            /* Input channel */
  OUT                   (2),            /* Output channel */

  MAX_LEN               (133),          /* Maximum string length */
  MAX_SIZE      ((MAX_LEN/2)-LEN),      /* Maximum number of words
                                        in node */

  ELEMENT_SIZE  (8);                    /* Element size to use */


EXTERNAL INTEGER (1) MINRES;            /* Lower bound of node */
EXTERNAL PROCEDURE CMCLOSE;
EXTERNAL STRING PROCEDURE GETCOUTPUT, GETCINPUT, NAMEGROUND;


                                        /* Node array */

INTEGER (1) ARRAY NEW_NODE [LEN:MAX_SIZE];

                                        /* Comparison node */

INTEGER (1) ARRAY COMPARE  [LEN:MAX_SIZE];

INTEGER (1) NEXT_NODE;                  /* Next node to be allocated */

STRING (MAX_LEN) INPUT;                 /* String to add to tree */
STRING FILENAME;                        /* Filename for cache memory */

POINTER BUF;                            /* Pointer to cache buffers */

```

Figure 10-2. Cache Memory Management Example (continues)

```
56  BASED STRING (MAX_LEN) BS;              /* Allow storing string
57                                                 in array.  */
58
59
60
61  /*
62   * Recursive procedure to insert the string in the correct
63   * alphabetical order
64   */
65
66  PROCEDURE SORT ( NODE );
67    INTEGER (1) NODE;                     /* Current node pointer */
68  BEGIN
69    INTEGER I, J;
70
71    NODEREAD ( NODE, COMPARE );           /* Read the node.  */
72    J := COMPARE[ CHAR_LEN ];             /* Size of string */
73
74      /* Now set up I to have right or left pointer.  */
75      /* Return if the string is already in the tree.  */
76
77    IF SUBSTR ( COMPARE[STR], 1, J ) <> INPUT THEN BEGIN
78
79      I := IF SUBSTR ( COMPARE[STR], 1, J ) < INPUT
80          THEN RIGHT                      /* Right subtree */
81          ELSE LEFT;                      /* Left  subtree */
82
83        /* Now see if subtree exists or not.  */
84
85      IF COMPARE[I] <> 0
86            /* Descend to compare again.  Note that COMPARE[I] must
87             * be passed by value because COMPARE is overwritten in
88             * in recursive calls to SORT.  */
89
90        THEN SORT ( (COMPARE[I]) )
91
92        ELSE BEGIN
93
94                                          /* Put node in tree.  */
95
96          STASH ( NEXT_NODE, NODE, I );
97          NODEWRITE ( NEXT_NODE, NEW_NODE );
98
99                                          /* Update NEXT_NODE now.  */
100
101         NEXT_NODE := NEXT_NODE +
102           (NEW_NODE[LEN]+ELEMENT_SIZE)/ELEMENT_SIZE;
103
104         IF (NEXT_NODE >= 0) AND
105            (NEXT_NODE < START_NODE) THEN
106           ERROR ("TABLE OVERFLOW");
107       END;
108   END;
109 END OF SORT;
110
```

Figure 10-2. Cache Memory Management Example (continued)

```
111
112
113  /*
114   * Recursive program to print sorted list of strings
115   */
116
117  PROCEDURE PRINT (NODE);
118    INTEGER NODE;
119  IF NODE <> 0 THEN BEGIN
120
121    PRINT ( FETCH ( NODE, LEFT ) );        /* Print left subtree.  */
122
123    NODEREAD ( NODE, COMPARE );            /* Print out node.  */
124
125    FORMAT ( OUT, "#<NL>",
126      SUBSTR ( COMPARE[ STR ], 1, COMPARE[ CHAR_LEN ]));
127
128    PRINT ( FETCH ( NODE, RIGHT ) );       /* Print right subtree.  */
129
130  END OF PRINT;
131
132
133
134  /*
135   * Procedure to read in a string, and set it up in the node
136  */
137
138  PROCEDURE READ_NODE ( DONE );
139    LABEL DONE;
140  BEGIN
141
142                                           /* Prompt user.  */
143
144    FORMAT ( OUT, "Type a string: " );
145
146                                           /* Read string.  */
147
148    READ ( INP, INPUT, DONE );
149
150                                           /* Done yet?  */
151
152    IF LENGTH ( INPUT ) = 0 THEN GOTO DONE;
153
154                                           /* Zero subtrees */
155
156    NEW_NODE[ LEFT ] := NEW_NODE[ RIGHT ] := 0;
157
158                                           /* String byte length */
159
160    NEW_NODE[ CHAR_LEN ] := LENGTH (INPUT);
161
162                                           /* Set up node word length.  */
163
164    NEW_NODE[ LEN ] := (NEW_NODE[ CHAR_LEN ]+1)/2 - MINRES;
165
```

Figure 10-2. Cache Memory Management Example (continued)

```
166                                         /* Store the string.  */
167
168     SUBSTR ( NEW_NODE[ STR ], 1, MAX_LEN ) := INPUT;
169
170   END OF READ_NODE;
171
172
173   /* * * * * * * * * * * * * * * * *
174    * MAIN PROGRAM follows below. *
175    * * * * * * * * * * * * * * * * */
176
177   OPEN ( OUT, (GETCOUTPUT) );
178   OPEN ( INP, (GETCINPUT) );
179
180   FORMAT ( OUT, "<NL>" );
181   FORMAT ( OUT, "Welcome to the DG/L cache memory string sorter.<NL>");
182   FORMAT ( OUT, "Type one string per line, ending with a NEW LINE.<NL>");
183   FORMAT ( OUT, "Terminate your list with an End of File (^D).<NL><NL>");
184
185   MINRES := LEN;                        /* Bottom of node */
186   FILENAME := NAMEGROUND ("CACHE");
187
188   DELETE ( FILENAME );
189
190                                         /* Set up the cache file.  */
191
192   BUFFER ( BUF, 2000 );
193   ACCESS ( 0, FILENAME, BUF, ELEMENT_SIZE);
194
195                                         /* Set up the first node at START_NODE.  */
196
197   READ_NODE ( ERR );
198   NODEWRITE ( START_NODE, NEW_NODE );
199   NEXT_NODE := START_NODE + (NEW_NODE[ LEN ] + ELEMENT_SIZE)/ELEMENT_SIZE;
200
201                                         /* Now read the other strings.  */
202
203   DO BEGIN
204     READ_NODE ( DONE1 );                /* Read */
205     SORT ( START_NODE );                /* and insert it.  */
206   END;
207
208
209
210 DONE1:     FORMAT ( OUT, "<NL><NL>The sorted strings are:<NL>");
211   PRINT ( START_NODE );
212   GO TO DONE;
213
214
215 ERR:       FORMAT ( OUT,
216       "<NL>ERROR IN FIRST READ, NO STRINGS TO PRINT.<NL>");
217
218
219
220 DONE:      CMCLOSE ( 0 );               /* Close CM file */
221   DELETE ( FILENAME );                  /* and delete it.  */
222
223 END OF CACHE MEMORY EXAMPLE;
```

Figure 10-2. Cache Memory Management Example (concluded)

The main program in Figure 10-2 begins on line 177. Lines 177 and 178 open channels for input and output, using the literals declared in lines 25 and 26 for channels, and the built-in external string procedures declared in line 38. Lines 180 through 183 send an introductory message to the terminal, telling what the program is for and how to proceed. Line 185 sets the lower bound of the cache memory file, or offset of the first word in the node, to -4, the value of the literal LEN declared in line 17. Line 186 sets the string variable FILENAME, declared on line 52, to CACHE, using the external procedure NAMEGROUND declared in line 38. NAMEGROUND sets up a unique filename based on the process identification (AOS and AOS/VS) or the ground (RDOS). Line 188 deletes a previously created file with the unique name created in line 186, if there is one, so that ACCESS in line 193 will create a new file with the unique name created in line 186 for each execution of the program.

To set up the cache file, line 192 allocates 2000 words to the buffer pool and stores the address in the pointer variable BUF, declared in line 54. Line 193 opens file CACHE on channel 0, associates it with the buffer pool that BUF points to, and declares an element size of ELEMENT_SIZE, which the compiler resolves to 8 on line 32.

With the cache file set up, line 197 calls the internal procedure READ_NODE, declared on line 138, with the actual label ERR of line 215 substituting for the formal parameter label DONE, declared within the procedure on line 139. The READ_NODE procedure stores input from the terminal with the Compound statement on lines 140-170. Line 144 prompts the user for an input string, and line 148 reads the response into the string variable INPUT, declared in line 51 with a precision (maximum length) of MAX_LEN, which the literal on line 28 resolves to 133 characters. If an error occurs, the program branches to line 215, sends an error message, and subsequently terminates. If there is no error, line 152 checks the length of INPUT. If the length is 0, the user entered nothing, and control passes as before to line 215.

If the length of INPUT is greater than 0, the user made an entry; the Boolean on line 152 fails and control passes to line 156, which sets two elements of the array NEW_NODE to 0. The integer array NEW_NODE, declared on line 43, has elements indexed from LEN, or -4, to the value of the literal (line 29) MAX_SIZE = ((MAX_LEN/2)-LEN) = (133/2)-(-4) = 70. Line 19 resolves the array index LEFT, for the left subtree node, to -2, or the third array element, and line 20 resolves the index RIGHT, for the right subtree node, to -1, or the fourth array element. So line 156 assigns the value 0, representing zero subtrees, to the third and fourth elements of the array NEW_NODE.

Since, by line 18, CHAR_LEN resolves to -3, line 160 assigns to the second element of NEW_NODE the length of the input string. Line 164 assigns to the first (-4) element of NEW_NODE the word length of the node, which it derives from the value in NEW_NODE[CHAR_LEN] (= NEW_NODE[-3] = LENGTH (INPUT)) plus 1 (which will round up, in case the length is odd) divided by 2 (since each word contains two bytes, or two characters), plus 4 (to compensate for the offset). If, for example, the input string is 12 characters long, the value assigned is ((12+1)/2)-(-4) = (13/2)-(-4) = 6-(-4) = 6+4 = 10.

Line 128 assigns the input string in INPUT to a subarray of NEW_NODE, which the SUBSTR routine converts from integer to string type. The substring begins at element STR, which line 21 resolves to 0. So the array NEW_NODE contains the input string, with its first character in element 0, or the fifth element of the array. Line 170 terminates the Compound statement and the procedure READ_NODE, and control passes to line 198.

Line 198 reads the array NEW_NODE into memory beginning at START_NODE, which line 23 resolves to 400R8. Line 199 sets the integer NEXT_NODE, declared on line 49, to a value derived by adding the value in NEW_NODE[LEN] to ELEMENT_SIZE, dividing by ELEMENT_SIZE, and adding the result to START_NODE. If NEW_NODE[LEN] is 10, the value assigned is 400R8+((10+8)/8) = 402R8. NEXT_NODE is the location in the tree of the next node.

Lines 203 through 206 contain an uncontrolled DO statement that reads and sorts any additional input strings. Line 204 calls procedure READ_NODE again, this time with DONE1 as the actual label for the formal parameter DONE. On this and successive reads, an error at line 148 or true Boolean at line 152 will lead to a branch to line 210, rather than line 215 as on the first procedure call. If there is no branch, READ_NODE will reset the array NEW_NODE as the new input string requires. Control then passes to the procedure call on line 205.

Line 205 calls SORT with START_NODE = 400R8 for the formal integer parameter NODE, declared on line 67. Line 71 transfers the node beginning at START_NODE into the array COMPARE, declared on line 47 with the same dimensions as NEW_NODE. Line 72 stores in the integer variable J, declared in the procedure at line 69, the length of the input string as given in COMPARE[CHAR_LEN]. Line 77 checks to see whether the substring of COMPARE from the fifth element STR = 0 to the length of the string is equal to the string in INPUT (some previous input string).

If the input string in COMPARE is equal to the one in INPUT (that is, it is a duplicate entry), the Boolean condition on line 77 fails, and control passes through line 109 and exits procedure SORT.

If the Boolean condition on line 77 suceeds, the two strings are not equal and control passes the Compound statement on lines 77 through 108, and first to the Assignment statement on line 79. Line 79 assigns RIGHT (-1) to I if the string in COMPARE is less than (precedes alphabetically) the INPUT string, and it assigns LEFT (-2) if the string in COMPARE is greater than (follows alphabetically) the INPUT string. This assignment specifies later, lines 90 through 107, whether to use the right subtree or the left subtree.

Line 85 tests whether the subtree specified (LEFT or RIGHT) exists (is not equal to 0). If it exists, then line 90 recursively calls SORT with the value of that subtree (LEFT or RIGHT) substituting for the formal parameter NODE. Line 71 reads a new string into COMPARE, and line 77 checks it again. Recursive calls of SORT continue until, on line 86, COMPARE[I] = 0, and the procedure has found the place in the tree at which it will attach the node for the INPUT string. At this point, the Boolean in line 85 fails and control passes to line 92.

The ELSE clause on line 92 contains a Compound statement on lines 92 through 107. Line 96 goes to the address specified by NODE (depending on cases, some previous value of COMPARE[I] or START_NODE), and stores the value of NEXT_NODE at the offset I (which has the value of either LEFT or RIGHT). Line 97 writes the INPUT line into the address specified by NEXT_NODE. Lines 101 and 102 increment the value in NEXT_NODE for the next new value of NEXT_NODE. That value is the node address of the next node that SORT will append to the cache.

Lines 104 through 107 ensure that SORT has not exceeded the limits of the cache memory address space, and abort the program with a message if it has.

Control passes to the FORMAT statement on line 210 and then to the procedure call PRINT, with the constant START_NODE for the formal parameter NODE, on line 211. On the first call to PRINT, START_NODE is greater than 0, so the Boolean on line 119 is true and control passes to the THEN clause, which contains the remainder of the procedure. Line 121 calls PRINT recursively with the value in the offset of the address in element -2 (LEFT) of the previous node. The calls continue until the procedure reaches a LEFT with 0 in the offset. At this point it had reached the lowest, leftmost node on the previous call. The previous call continues with line 123 reading that node into COMPARE. Lines 125 and 126 print out the substring of COMPARE that consists of the character string.

Line 128 then calls the procedure with RIGHT. If RIGHT is 0, control passes to the previous call with LEFT. If RIGHT is not 0, line 121 makes a new call to PRINT with LEFT. If on this call LEFT is zero, then RIGHT is the next lowest, leftmost item in the list, and the previous call continues with a printout of RIGHT and a further call to PRINT with LEFT. The procedure continues calling itself until it has printed out all next lowest and leftmost nodes; the result is an alphabetic listing of the INPUT strings.

Control then passes from procedure PRINT to line 212, which specifies a skip to line 220. Line 220 closes CACHE, the cache memory file, and line 221 deletes it. Line 223 terminates the main program.

Cache Memory Routines

This section describes routines that allow you to manage cache memory.

ACCESS *Routine*

Open a cache memory file.

Format

ACCESS (filenumber,filename,buf-pool-ptr*[,elementsize]*);

Filenumber and filename are user-defined, buf-pool-ptr is the pointer returned by the BUFFER routine, and *elementsize* is an optional specification of logical element size, in words.

Description

The ACCESS routine opens a CMM file, associating a buffer pool and element size with the file.

Remarks

1. In the 32-bit AOS/VS environment, buf-pool-pointer must be a double word (INTEGER (2) or POINTER type).
2. The default element size value is 1.
3. You can access several files using the same buf-pool-ptr.

Example

```
ACCESS (1,FILE1,BUFF,3);
```

opens file number 1 as a CMM file, with the name FILE1, and a pointer named BUFF, and specifies the element size as 3.

BUFFER *Routine*

Assign a buffer pool in memory.

Format

BUFFER(buf-pool-ptr,poolsize,*[virtualbuf]*);

Buf-pool-ptr is the value returned, a pointer to the memory location of the CMM buffer, poolsize is the number of words to be allocated to buffers (for an unshared AOS/VS, unshared AOS, or unmapped RDOS system), and *virtualbuf* is the number of buffers to be created (for a mapped RDOS system), or the number of shared pages to use if you are taking advantage of the shared-page cache memory on AOS or AOS/VS.

Description

The BUFFER routine assigns a buffer pool area in memory, and yields a pointer to that area.

Remarks

1. In the AOS/VS 32-bit environment, buff-pool-ptr must be a double word (INTEGER (2) or POINTER type). The argument *virtualbuf* is ignored if you use unshared cache memory (default) routines. If you use shared-page cache (library DGL32SCACHE.LB), poolsize is ignored, and *virtualbuf* is the maximum number of shared pages for the cache memory routines, for this buffer pool.
2. Unmapped systems will ignore *virtualbuf*, and mapped systems will ignore poolsize, except that poolsize is used on a mapped system having fewer than 31+.VMIN available 1K blocks.
3. The EXTERNAL INTEGER NUMBUF gives the number of virtual memory buffers available at the start of a program on RDOS, and the number of shared pages available on AOS and AOS/VS.

Example

```
BUFFER(BUFF,2000,7);
```

will allocate 7 buffers (in either a mapped or an unmapped system). The entire buffer pool will be allocated at most 2000 words of memory.

BUFLOCK *Routine*

Lock a 256-word block into cache memory.

Format

BUFLOCK(file-number, logical-address);

File-number is an integer expression whose value is associated with the file, and logical-address is an integer expression whose value is an address within the block that will be locked in memory.

Description

The BUFLOCK routine locks a 256-word block into memory. The block containing logical-address is not written out to disk even if it becomes the least-recently-used one. If the block is currently not in memory, it is read in.

Remarks

1. If all blocks are locked into memory, the process or task terminates with a fatal error.
2. Since BUFLOCK is not a built-in routine, you must declare it as an external procedure.

Example

```
EXTERNAL PROCEDURE BUFLOCK;
.
.
.
BUFLOCK (MYBLOCK,DATA);
```

BUFUNLOCK *Routine*

Unlock a 256-word block in cache memory.

Format

```
BUFUNLOCK (file-number, logical-address);
```

Description

The BUFUNLOCK routine unlocks the block containing the element at logical-address. It may be written to disk if it becomes the least-recently-used block.

Remark

Since BUFUNLOCK is not a built-in routine, you must declare it as an external procedure.

Example

```
EXTERNAL PROCEDURE BUFUNLOCK;
.
.
.
BUFUNLOCK (FILE2, LOC);
```

CMCLOSE *Routine*

Close a cache memory file.

Format

CMCLOSE (file-number);

File-number is of type INTEGER (1).

Description

The CMCLOSE routine closes a cache memory file opened with the ACCESS routine.

Remarks

1. Since CMCLOSE automatically flushes the buffers, you need not precede it with a FLUSH.
2. This routine uses ?SCLOSE for shared-page cache, and ?GCLOSE (AOS/VS)/?CLOSE (AOS)/.CLOSE (RDOS) for unshared cache. You must use CMCLOSE on shared-page cache files instead of FLUSH and CLOSE.
3. Since CMCLOSE is not a built-in routine, you must declare it as an external procedure.

Example

```
EXTERNAL PROCEDURE CMCLOSE;
.
.
.
CMCLOSE (3);
```

FETCH *Function*

Retrieve a word from a cache memory file on file 0.

Format

I := FETCH(*[logical-address,]* subscript);

I is an integer variable that receives the word, *logical-address* is an integer (1) treated as unsigned, and subscript is an offset into the node of the word to fetch.

Description

The FETCH function obtains the contents of the word pointed to by the *logical-address* plus the offset subscript-MINRES.

Remarks

1. You can use FETCH only on file 0.
2. If you specify *logical-address* (you may omit it), it is not possible to refer to logical addresses 0-255. If you attempt this, the system returns a 0 value.
3. If you omit *logical-address*, it is assumed to be 0, so physical addresses 0-255 can be accessed, and subscript forms a word offset from the file's beginning.
4. See also STASH.
5. The displacement from the beginning of the node is

   subscript - MINRES
6. If MINRES was -3, a FETCH(500,0) call would get the fourth word of element 500, not the first. FETCH assumes the first node word has offset MINRES, the next MINRES+1, and so on. If you omit *logical-address* (fetching from the start of disk file, useful for global parameters) then offset counts from 0, not MINRES.
7. FETCH uses either shared or unshared cache.

Example

```
I := FETCH(P1,4);
```

fetches a word from the location offset (4 - MINRES) words from logical address P1.

FLUSH

Routine

Write out contents of all modified buffers in pool to disk.

Format

```
FLUSH (buf-pool-ptr);
```

Description

The FLUSH routine writes out to disk all CMM buffers that your program has modified since it read them.

Remarks

1. In the 32-bit environment, the pointer buf-pool-ptr must be a double word.
2. You do not need to use FLUSH before a CMCLOSE call, since CMCLOSE automatically flushes buffer files to disk. But if you do not use CMCLOSE, you should use FLUSH before a CLOSE on a modified CMM file, since CLOSE does not write out modified buffers.
3. Buf-pool-ptr is the same pointer returned by the BUFFER call.
4. FLUSH operates on both the shared and the unshared cache memory system.

Example

```
FLUSH (BUFF1);
```

FLUSH writes to the disk all modified buffers that BUFF1 points to.

HASHBACK *Routine*

Bring the last HASHREAD buffer back into memory in a mapped RDOS system.

Format

```
HASHBACK(buf-pool-ptr);
```

Description

In mapped RDOS systems, the HASHBACK routine brings the last HASHREAD buffer back into memory. On other systems, this routine is inoperative.

Remarks

1. HASHBACK has no effect, and is not needed, under AOS/VS, AOS, or unmapped RDOS.
2. You must use this call any time you use a HASHREAD block in a mapped RDOS system after an intervening CMM access, to guarantee that a block is still in memory.
3. The buf-pool-ptr is the same one returned by the BUFFER call.

Example

```
HASHBACK(BUFF);
```

assures that the buffer pointed to by BUFF is in main user memory.

HASHREAD *Routine*

Read a block of the hash file into memory.

Format

```
HASHREAD(file-number,logical-address,blockptr,offset);
```

Description

The HASHREAD routine locates a word in the specified **file-number**, at the specified **logical-address**, and returns a pointer to the CMM block buffer and an offset into that block, showing where you can find the specified **logical-address**.

Remarks

1. In the 32-bit environment, the pointer **blockptr** must be a double word.
2. Do not call HASHREAD or HASHWRITE under multitasking if more than one task is accessing the cache memory system.
3. The file's element size must be 1.
4. The combination of **blockptr** and **offset** forms a memory pointer, which you can use like any other pointer.

Example

```
HASHREAD(1,ELMT,BKNO,BKOFF);
```

returns pointer BKNO and offset BKOFF pointing to logical address ELMT in file number 1. The contents of this logical address could then be read into DATA with

```
DATA := (BKNO+BKOFF) -> BI;
```

HASHWRITE *Routine*

Mark the current buffer as modified.

Format

```
HASHWRITE(buf-pool-ptr);
```

Description

The HASHWRITE routine marks the last HASHREAD buffer as modified, so that the system will write it out to disk when necessary.

Remarks

1. In the 32-bit environment, the pointer buf-pool-ptr must be a double word.
2. Do not call HASHREAD or HASHWRITE under multitasking.
3. This routine operates on both the shared and unshared page cache memory.
4. Any modification in a HASHREAD file requires that you use this call, or the block will not be marked as modified.
5. The buf-pool-ptr is the same one returned by BUFFER.

Example

```
HASHWRITE (BUFF1);
```

MINRES *Function*

Access the default lower bound of a cache memory file.

Format

```
EXTERNAL INTEGER MINRES;
.
.
.
MINRES := value;
```

Value is the offset to use in FETCH and STASH routines to access the first word in a node. Subsequent words are value+1, value+2,....

Description

MINRES is a declaration to specify, modify, or otherwise access the default lower bound of a cache memory file. The FETCH and STASH routines use MINRES to determine the word offset from the beginning of the node, by assuming the first word in the node is offset MINRES.

Remark

The default value for MINRES is -3.

Example

```
BEGIN MINRES := 2;
    INTEGER ARRAY LOC_ARRAY[MINRES:MINRES+NODESIZE(EMPTR)];
    END;
```

NODEREAD *Routine*

Read a node from file 0 into memory.

Format

NODEREAD(logical-address,array);

Logical-address is an INTEGER (1) and array is an ARRAY.

Description

The NODEREAD routine reads a node from the specified logical address (in file 0) into the array you have defined.

Remarks

1. The number of words to be read is found in the first word of the node.
2. The element size specification includes MINRES.
3. Since a node always starts at an element boundary, you need no offset.
4. This routine operates on both shared and unshared page cache memory.

Example

NODEREAD(ELMT,CURNODE);

reads the node at ELMT into array CURNODE.

NODESIZE *Function*

Obtain the size of a node on file 0.

Format

```
I := NODESIZE(logical-address);
```

I is an INTEGER (1) and **logical-address** is an integer expression that specifies the logical element address of the first word of the node in the file.

Description

The NODESIZE routine gets the size (in words) of the node at the specified **logical-address** on file 0.

Remark

This routine operates on both shared and unshared page cache memory.

Example

```
INTEGER ARRAY R [MINRES:NODESIZE(P) + MINRES];
```

allocates an array large enough to hold the node at logical address P.

NODEWRITE *Routine*

Write a node from memory to file 0.

Format

NODEWRITE(logical-address,array);

Logical-address is an INTEGER (1) and array is an ARRAY.

Description

The NODEWRITE routine transfers a node from array to the CMM logical-address (on file 0).

Remarks

1. The first word of the array must contain the number of words to write (the nodesize).
2. The element size includes one word for the MINRES word.
3. This routine operates on both shared and unshared page cache memory.

Example

```
NODEWRITE(ELMT,MEMNODE);
```

writes array MEMNODE to logical address ELMT.

STASH *Routine*

Store a word into a node on file 0.

Format

STASH(I *[,logical-address]*, subscript);

I, *logical-address*, and subscript are of type INTEGER (1).

Description

The STASH routine stores the contents of word I at the offset subscript-MINRES from the specified logical address on file 0.

Remarks

1. You can use STASH only for file 0.
2. If you specify the optional *logical-address*, you cannot access logical addresses 0-255. Attempts to do so do not generate errors, but no operation is performed. If you do not specify *logical-address*, you can access locations 0-255, and subscript is used as a physical word offset from the beginning of the file.
3. This routine operates on both shared and unshared pages in cache memory.
4. The displacement, in words, from the beginning of the node is

   subscript - MINRES

Example

```
STASH(I,ELMT,4);
```

would store word I into the location described by ELMT plus an offset of 4.

WORDREAD *Routine*

Read words from a file using cache memory.

Format

WORDREAD(file-number,logical-address,memptr *[,words [,offset]]*);

Memptr is a pointer to the first memory location to receive data, *words* is the number of words to be read, and logical-address represents the logical starting address of the words to be read. You may include *offset* only if *words* is also included (it is the "most optional" parameter of the list). It specifies an offset from the beginning of the element from which the words are to be read.

Description

The WORDREAD routine reads a contiguous block of words from the specified logical-address and file.

Remarks

1. In the 32-bit environment, memptr must be a double word.
2. If you omit the optional *words* field, the number of words to be read comes from the first word at the logical address.
3. The offset specification is designed to allow an available address space, in conjunction with increased element sizes. The offset makes it easier to refer to consecutive elements.
4. This routine operates on both shared and unshared page cache memory.

Examples

```
WORDREAD(1,ELMT,COREPTR,3);
```

reads 3 words from file number 1, beginning at location ELMT, into the area in memory beginning at COREPTR.

```
WORDREAD (1, POS, PTR, 2, 2);
```

will first locate element POS in file 1. It will then advance two words from POS, and then read and return the two words starting at that address.

WORDWRITE *Routine*

Write words to a file using cache memory.

Format

WORDWRITE (file-number,logical-address,memptr *[,words [offset]]*);

Logical-address is the logical address of the words to be written, memptr is a user-defined pointer that specifies the starting address in memory of the data to be written, and *words*, if specified, is the number of words to be written. The "most optional" parameter in the list, *offset*, is an offset from the beginning of the element specified by logical-address at which writing is to begin.

Description

The WORDWRITE routine writes a contiguous block of words to the specified logical address in the specified file.

Remarks

1. In the 32-bit environment, the pointer memptr must be a double word.
2. If you omit words, the word count to be written comes from the first word at the user memory address specified.
3. See the description of WORDREAD for further discussion of offset.
4. This routine operates on both shared and unshared page cache memory.

Example

```
WORDWRITE (1,ELMT,MEMPTR,3);
```

writes 3 words to file number 1 at logical address ELMT. The words to be written start at memory location MEMPTR.

The INCLUDE Facility

The DG/L INCLUDE facility lets you incorporate DG/L source text from several files into a DG/L program. You use INCLUDE in the form

```
INCLUDE filename;
```

The filename is not quoted or parenthesized. Unless you explicitly specify an extension or follow the filename with a period to indicate that no extension is to be used, the compiler first searches for source text in a file named filename.DG. If this search fails, the compiler looks for filename with no extension. If you specify an explicit extension or that no extension be used, the compiler will look only for the exact filename you specify. For example, if a DG/L program contains

```
INCLUDE DECL;
```

the compiler will search for source text from file DECL.DG. If DECL.DG doesn't exist, the compiler will try for DECL. If neither file can be found, an error occurs. In contrast,

```
INCLUDE DECL.;
```

will cause the compiler to look for source text only in file DECL., not DECL.DG.

```
INCLUDE DECL.DG;
```

will search only for DECL.DG, not DECL. INCLUDE directives can occur anywhere within DG/L programs, and included files may contain both declarations and statements, so long as the entire resulting program conforms to DG/L structural rules.

As a simple example of the use of INCLUDE, consider file DECS.DG to contain

```
INTEGER INTE;
REAL REALLY;
```

If a program contains

```
BEGIN
    INCLUDE DECS.DG;
    REALLY := REALLY + INTE;
    END;
```

it will be processed equivalently to

```
BEGIN
    INTEGER INTE;
    REAL REALLY;
    REALLY := REALLY + INTE;
    END;
```

You may nest INCLUDE files (in other words, a file that you INCLUDE may contain an INCLUDE directive) to a depth of four, but the recursive use of the INCLUDE facility (doing an INCLUDE on a file within that same file) causes errors.

Overlay Support and the Overlay Declaration

An overlay is a section of code that resides on the disk, and comes into main memory only when requested. It takes the form of an EXTERNAL procedure. DG/L works with operating systems to let you divide large programs into multiple nodes that the operating system accesses by node numbers and overlay numbers. The node numbers refer to overlay areas, and the overlay numbers identify specific overlays.

AOS/VS does not support overlays in the 32-bit environment. For details on supporting overlays, if your operating system is

- 16-bit AOS/VS, consult the *Advanced Operating System/Virtual Storage (AOS/VS) Link and Library File Editor (LFE) User's Manual* and the *DG/L Runtime Library (AOS and AOS/VS) User's Manual*
- AOS, consult the *Advanced Operating System (AOS) Link User's Manual* and the *DG/L Runtime Library (AOS and AOS/VS) User's Manual*
- RDOS, consult the *Extended Relocatable Loaders User's Manual* and the *DG/L Runtime Library User's Manual (RDOS)*

DG/L overlays take the form of procedures. To implement an overlay under RDOS, the declaration part of the outermost block of the overlay must contain an OVERLAY declaration, which takes the form

```
OVERLAY oname;
```

This declaration causes DG/L to generate an entry symbol (.ENTO) to the overlay.

Under RDOS, you must do four things in your program before you can transfer the control to an overlay.

1. You must include in the main program the declarations

   ```
   EXTERNAL INTEGER oname;
   EXTERNAL PROCEDURE OVOPN, OVLOD, OVCLOSE;
   ```

   to access OVOPN, OVCLOSE and OVLOD from the DG/L runtime library. The EXTERNAL INTEGER declaration refers to the same identifier as specified in the OVERLAY declaration.

2. The procedure must call OVOPN to open the overlay file (already generated by RLDR). This call takes the form

   OVOPN (filenumber,filename *[,err]*);

   where **filenumber** is the file number under which you open the overlay file, **filename** is the name of the generated overlay file, and *err* is an optional error return label if the overlay file specified can't be opened.

3. Call OVLOD to load the appropriate overlay node into memory. A call to OVLOD takes the form

   OVLOD (oname *[,cond flag]*);

   where **oname** is the name referred to in the OVERLAY declaration and *cond flag* is a flag of 0 implying conditional load and -1 implying unconditional load.

   After the overlay has been loaded, it can be called like any other procedure, the call format depending on parameters to be passed to it and results to be returned.

4. When using overlays with multitasking, after you finish with an overlay, you must signal to the overlay manager that the overlay area can now be used for a different overlay. You do so by using OVREL. The syntax is

   ```
   OVREL(oname);
   ```

   where oname is the same as for the OVOPN call.

   Under AOS and 16-bit AOS/VS, all internal and external procedure calls use the AOS and AOS/VS ?RCALL mechanism, which provides load-on-call overlay capability. Overlays do not require or permit explicit overlay calls, such as OVLOD or OVRELEASE, or the OVERLAY declarator. You can decide at link time which of your routines to put in what overlays, without recompiling. Chapter 11, "Operating Procedures," specifies the form of LINK lines using overlays.

Additional Data Reference Methods

You can combine the DG/L POINTER and LITERAL facilities to create flexible methods for referring to data objects and items. How you use these facilities of course depends on your program. This section provides some guidelines on data references, showing some capabilities that may not be apparent at first glance.

To take a simple case, you can declare literals corresponding to subscripted array elements. This sometimes makes programs easier to read. For example, consider a group of 5 data words, named as

Word	Name
0	$DATUM
1	$TYPE
2	$INIT
3	$FINAL
4	$NEXT

Each of these items represents a word descriptor used as part of a database. You could define appropriate literals as

```
INTEGER   ARRAY IA [4];
LITERAL   $DATUM (IA[0]),
          $TYPE(IA[1]),
          $INIT(IA[2]),
          $FINAL(IA[3]),
          $NEXT(IA[4]);
```

Once you have made these declarations, you can make symbolic references to the elements in integer array IA, as in

```
$INIT := 0;
```

which the compiler resolves as equivalent to

```
IA[2] := 0;
```

You can also declare literals that represent pointer expressions. This lets you refer to data items with pointers while still maintaining easily comprehensible symbolic naming. For example, if you declare

```
INTEGER ARRAY IA [4];
POINTER IP1, IP2;
BASED INTEGER BI;
LITERAL     $DATUM (IP1 -> BI),
            $TYPE((IP1+1) -> BI),
            $INIT((IP1+2) -> BI),
            $FINAL((IP1+3) -> BI),
            $NEXT((IP1+4) -> BI);
```

and execute the statement

```
IP1 := ADDRESS (IA);
```

you can refer to the elements in array IA in much the same way as the last example referenced the elements of IA. For example,

```
$DATUM := $TYPE - 1;
```

is processed as equivalent to

```
(IP1 -> BI) := ((IP1+1) -> BI) - 1;
```

Remember that literals are always treated as parenthesized expressions where precedence is concerned.

You can expand the concept of pointer expression substitution to allow the substitution of embedded pointer expressions. For example, if you include the following statement in a DG/L program

```
IP2 := (IP1 -> $NEXT);
```

it is expanded to

```
IP2 := (IP1 -> ((IP1 + 4) -> BI));
```

Constructions like these let you refer to complex tree representation of data in easy-to-handle symbolic form.

BASED strings are templates referring to fixed numbers of contiguous bytes in memory. You can use them to refer to entire data constructs, which is desirable if you wish to move the constructs from one memory area to another. As an example, the lines

```
POINTER P1, P2;
.
.
.
BASED STRING (10) BS10;
LITERAL OLD (P1 -> BS10),
        NEW (P2 -> BS10);
NEW := OLD; /* MOVE RECORD */
```

move 10 bytes from the locations described by pointer P1 (which points to the first location in the BASED string) to the locations described by P2. When you pass a BASED string to a procedure, you can use a string specifier, and treat it as a string.

Another use of the LITERAL facility lets you simulate n-ary logic, as opposed to binary logic. That is, Boolean comparisons can be given symbolic names, as in

```
LITERAL     JAN (MONTH = "JANUARY"),
            FEB (MONTH = "FEBRUARY"),
            MAR (MONTH = "MARCH"),
            APR (MONTH = "APRIL"),
            MAY (MONTH = "MAY"),
            JUN (MONTH = "JUNE"),
            JUL (MONTH = "JULY"),
            AUG (MONTH = "AUGUST"),
            SEP (MONTH = "SEPTEMBER"),
            OCT (MONTH = "OCTOBER"),
            NOV (MONTH = "NOVEMBER"),
            DEC (MONTH = "DECEMBER");
```

Given this declaration, and the declaration at some appropriate place in the program of MONTH as a string variable, you could use the construct

```
IF JAN OR FEB OR MAR THEN GO TO WINTER;
```

The Boolean expression part of this statement can also be represented as a literal

```
LITERAL     WINTER (JAN OR FEB OR MAR),
            SPRING (APR OR MAY OR JUN),
            SUMMER (JUL OR AUG OR SEP),
            FALL (OCT OR NOV OR DEC);
```

This declaration would enable you to use the statement

```
IF WINTER THEN
    GO TO WARM_CLIMATE;
```

if you declared WARM_CLIMATE as a label.

As a generalization of the preceding methods, consider how you might create a data object. A data object is made up of one or more individual data items. Each item may itself contain an address pointing to another object, and the items within a given object may have different types and sizes. Once you have determined the items to be contained within an object, allocate the appropriate memory area for the object. Note each item's length, and its offset relative to the starting address of the object. Declare BASED identifiers with attributes corresponding to those of each data type that you want to reference.

Declare a literal, one for each item to which you want to refer, in the following form representing a pointer expression

```
LITERAL litname (name -> based-id);
```

where based-id corresponds to the item's type.

Then, declare the names from the last batch of literals as literals themselves, in the form

LITERAL name (obj-addr-offset);

where obj-addr-offset is the starting address of the object plus the offset of the item within the object.

If any of the items point to other objects, declare more literals, as

LITERAL ptd-to-name (litname-plus-offset −>
based-id);

where litname-plus-offset describes the location of an item within the pointing object, and ptd-to-name is a name for the object pointed to. Continue this process until all levels of the object and objects that it points to are described.

The following example illustrates this process. The example object is defined as follows. Its first two words are treated as a four-character string. The following word is a pointer, which points to another object containing two pointers, each of which in turn points to a five-character string. The following word contains a Boolean value. The remaining four words of the object are to be treated as a double-precision real number. The diagram in Figure 10-3 illustrates the sample object.

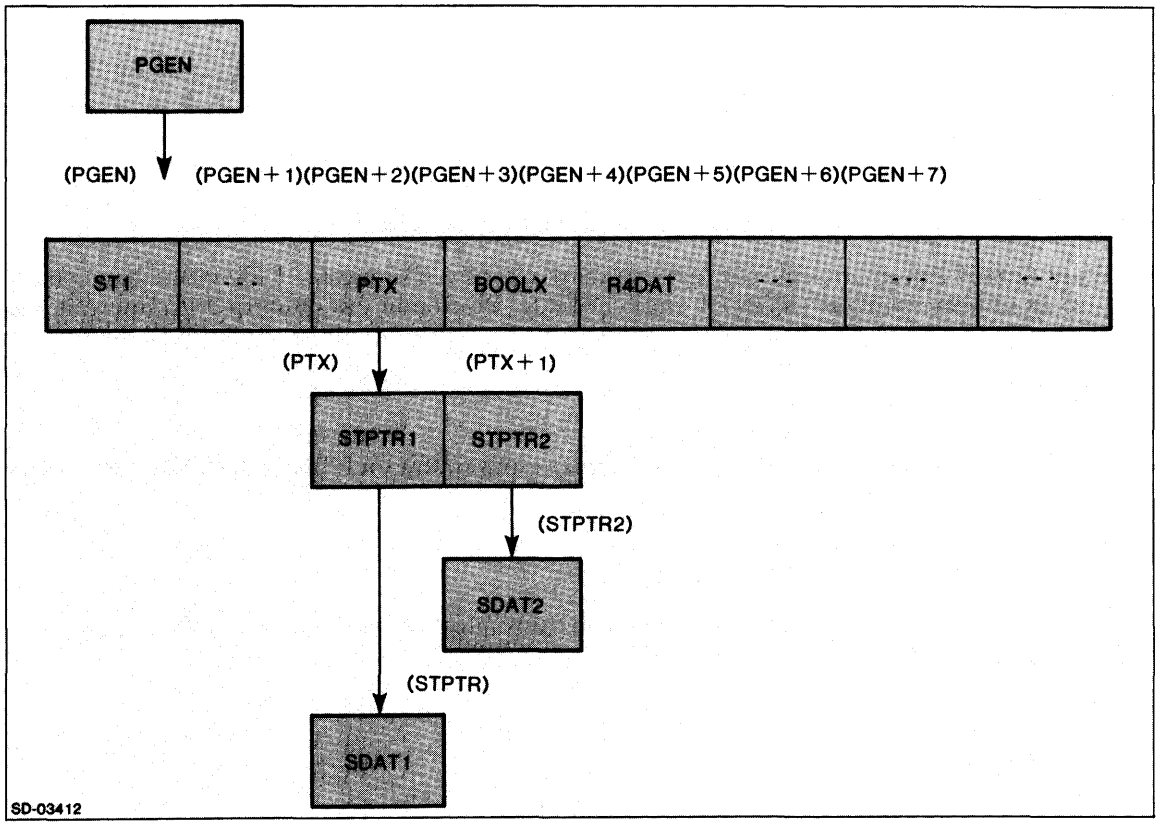


Figure 10-3. Sample Object

The following declarations serve to describe each type of data to be represented in the object

```
POINTER PGEN;
BASED STRING (4) BS4;
BASED STRING (5) BS5;
BASED POINTER BP;
BASED BOOLEAN BB;
BASED REAL (4) BR;
```

BS4 is a based string template, and defines a four-character (two-word) string. BS5 is another based string template, which will be used as a template to describe two five-character strings. BP, BB, and BR are used to describe the remaining data types referred to.

The following declarations take care of the requirement to declare a literal specifying a pointer expression that points to each type of data you want to access.

```
LITERAL ST1 (PS->BS4);
LITERAL PTX (PP->BP);
LITERAL BOOLX (PB->BB);
LITERAL R4DAT (PR->BR);
```

The next set of literals establishes a correspondence between each datum and its offset from the beginning of the data object.

```
LITERAL PS (PGEN);
LITERAL PP (PGEN+2);
LITERAL PB (PGEN+3);
LITERAL PR (PGEN+4);
```

Reference to the topmost level of the picture should make these offsets clear.

There are further objects to point out that you must describe. The pointer PTX in the basic object points to two further pointers (STPTR1 and STPTR2). Describing this requires declarations in the form

```
LITERAL STPTR1 (PTX->BP);
LITERAL STPTR2 ((PTX+1)->BP);
```

We still need to describe one more level of reference, representing the SDAT1 and SDAT2 that this last pair of pointers accesses, as in

```
LITERAL     SDAT1 (STPTR1 -> BS5),
            SDAT2 (STPTR1+1)->BS5);
```

You must ALLOCATE appropriate storage space to the pointers involved. In this example, appropriate ALLOCATE calls would be

```
ALLOCATE (PGEN,8);
ALLOCATE (PTX, 2);
ALLOCATE (STPTR1, 3);
ALLOCATE (STPTR2, 3);
```

Once these declarations and allocations are made, you can refer to items in the object without writing out pointer expressions. Simply refer to an item by its symbolic name, and let the compiler's literal facility do the rest. If, for example, you access BOOLX, the compiler processes BOOLX into PB −> BB, and PB in turn into PGEN+3, thus accessing BOOLX through the appropriate pointer expression, (PGEN+3)−>BB. If the precedence frightens you, recall that literals are always treated like entities in parentheses. The DG/L literal facility easily expands the other names to the pointer expressions describing their addresses and types, and generates as complex an expression as needed. Consider, for example, what happens on a reference to SDAT1. It is first resolved as

```
(STPTR1 −> BS5)
```

BS5 is a based string, so the right-hand side of this pointer expression is fully resolved. However, STPTR1 is itself a LITERAL, so another level of resolution takes place, leading to

```
((PTX −> BP) −> BS5)
```

which is used to evaluate SDAT1. This method eases your programming by letting the DG/L literal facility do the expression-substitution job it was designed to do. It frees you from clerical manipulation of pointer expressions, and lets you concentrate on the structure of your program.

End of Chapter

Chapter 11
Operating Procedures

This chapter shows you how to compile and then link an executable DG/L program.

Compilation

The compilation line varies with the operating system.

Format for AOS and AOS/VS

The format under AOS and AOS/VS for a DG/L compilation command line is

XEQ*//S/* DGL*/command-switch .../* sourcefilename*/arg-switch .../*

The /S switch allows you to store the program termination IPC message in STRING. You can use the value of STRING in a macro to initiate a link just in case the compiler sent no error messages. Your macro might look something like

```
XEQ/S DGL/L%/% %1%
[!EQUAL,([!STRING]),()]
     WRITE Successfully compiled %1%.
     XEQ LINK/NSLS %1%-% [DGLIB]
[!ELSE]
     WRITE Compilation failed. Sent error message:
     WRITE [!STRING]
[!END]
```

This macro attempts to link if STRING is empty. If STRING is not empty, it contains an error message from the compiler, and in that case, the macro displays the error message instead of attempting the link.

Global command switches follow the DG/L program name. These allow you to specify nondefault options for your compilation. The switches can be simple, of the form /S, or they can be keyword switches, of the form /S=string. In the latter case, depending on the keywords you select, the string can specify code options, storage files, conditional compilation options, revision numbers, or a directory for temporary files. You can find details in the section "Command Switches."

The name of the source file follows any command switches. The compiler first checks your input file for a dot, indicating an extension specification. If there is an extension specifier, the compiler uses it. If there is no extension, the compiler tries to read source text from filename.DG. If it doesn't find filename.DG, it searches for filename. If all of these attempts fail, the compiler sends an error message.

You may follow the source filename with local argument switches at your option. Under AOS and AOS/VS, these switches are the equivalents of certain command switches. You can find details of their use in the section "Argument Switches."

The relocatable binary object file for the source filename has a default name filename.OB under AOS and AOS/VS.

Format for RDOS

The format under RDOS for a compilation command line is

DGL*[command-switch ...]* sourcefilename *[arg-switch ...]*

Certain command switches are inoperative under RDOS, as the section "Command Switches" specifies. Moreover, you must use the argument switch alternative if you need a switch more than one symbol long. This includes all keyword switches.

The relocatable binary object file for the source filename has the default name filename.RB under RDOS.

Command Switches

/A

Proceed through all phases of the compiler that detect errors, regardless of syntax errors.

/B

Produce a brief output listing (source text and storage map only). You must use the global or local /L switch with this switch.

/C

Check syntax of source text, but do not generate output code, or check the semantics of the program.

/CODE=code-option

Generate code for the specified machine/operating system. You may generate AOS or RDOS code on AOS/VS, but not vice versa. You may also generate RDOS code on AOS but not vice versa.

Code=	Machine	Operating System
N	NOVA®	RDOS
NOVA	NOVA	RDOS
E	ECLIPSE®	RDOS
ECLIPSE	ECLIPSE	RDOS
RDOS	ECLIPSE	RDOS
A	ECLIPSE	AOS
AOS	ECLIPSE	AOS
16	ECLIPSE	AOS
X16	ECLIPSE	AOS
VS16	ECLIPSE	AOS
A	MV/8000-16	AOS/VS
AOS	MV/8000-16	AOS/VS
16	MV/8000-16	AOS/VS
X16	MV/8000-16	AOS/VS
VS16	MV/8000-16	AOS/VS
X	MV/8000-32	AOS/VS
X32	MV/8000-32	AOS/VS
32	MV/8000-32	AOS/VS
VS32	MV/8000-32	AOS/VS

If you omit this switch, DG/L generates code for the current environment. On RDOS systems, you must use the local /C argument switch instead of the /CODE= command switch.

/E=filename

List all errors in this file. Specified listing file will also list errors. On RDOS, use the /E argument switch instead. If you specify no list file or error file, error messages appear at the terminal.

/F

Direct the initializer not to initialize the floating-point unit, and flag as an error any code that would require use of the floating-point unit.

/G

Emit local symbols in the object file for the location of the start of each line (for AOS and AOS/VS only), and emit all procedure names as declared. Since procedure names are not scoped (Chapters 6 and 7) here, you will get multiple definition errors if you have two identical names (up to eight characters) in your program if you use /G. Use the /LOCAL link switch on a filename when you later invoke the Link utility to create an executable program. The /G switch then causes source line numbers to appear in the symbol table for those procedures you so designate. Line numbers appear in the symbol table as either LINE123 for .MAIN, or MYOWN123 for PROCEDURE MYOWN. These source line numbers match those in the generated code section of the listing file.

/H

Use the hardware multiply/divide unit instead of the software simulation (for NOVA code only).

/I

Suppress listing and cross-references of include files.

/INNER

Compiled code is linked for an inner (4-6) ring, and can be called through a gate array from an outer ring. This switch is valid only for 32-bit AOS/VS.

/L

Produce a listing. When compiling filename.DG, /L directs the listing to @LIST under AOS and AOS/VS, and to filename.LS under RDOS.

/L=filename

Put listing in filename. On RDOS, use the /L argument switch instead.

/M

If generating code for RDOS ECLIPSE, use LEF (Load Effective Address) instructions if it is feasible to do so. This switch is assumed when generating code for AOS and AOS/VS systems.

/N

Do not generate object code, but proceed otherwise with all compiler phases, and provide an object code listing.

/NOLEF

Do not use LEF instructions in code (opposite of /M). This switch applies only to AOS and AOS/VS environments.

/O=filename

Place object file in filename. If you do not specify the filename with this switch, filename.OB and filename.DG have the same name except for the extensions. On RDOS, use the /B argument switch instead.

/OPT=string

Compile conditionally lines surrounded by /**xxx*/, where xxx are any letters contained in this string. If there is a minus sign (-) in xxx, compile only if none of the characters in xxx are in the string. On RDOS, use the /O argument switch instead.

/P

Assume the correct number of arguments will always be passed to procedures called externally. If you are sure that this will be the case, /P compilation eliminates the need for many unnecessary runtime checks.

/Q

Allow question marks (?) in identifier names.

/R

Do all integer division in subexpressions with floating-point arithmetic. This increases accuracy by reducing rounding and truncation errors.

/REV=rev-no

where for	**rev-no is**
RDOS, AOS, 16-bit AOS/VS	num1*/.[num2[.]]]*
32-bit AOS/VS	num1*/.[num2[.[num3[.[num4[.]]]]]]*

and	**is a**
num1	major revision number
num2	minor revision number
num3	update number
num4	pass number

and where	**for**
num < 100	/CODE=N (RDOS code generation) /CODE=E
num < 256	/CODE=A (16-bit AOS and AOS/VS code generation) /CODE=X (32-bit AOS/VS code generation)

Enter a revision number for an object .OB or .RB file. The default revision number is the current DG/L compiler revision number if you set no REV switch. On RDOS, use the /R argument switch instead.

/S

Generate code for full subscript checking. Use this switch normally only when debugging. An object program without subscript checking runs faster and requires less memory space.

/T

Report string overflows. In a program compiled under /T, an attempt to store before the first or after the last character of a string generates a nonfatal error.

/TEMP=directory

Put temporary files in this directory. If directory is on a fixed-head disk, compilation speed may be faster. This switch is valid only for AOS and AOS/VS environments.

/V

Add block level of the line to the listing, whether the line is from an INCLUDE file, and whether it compiled conditionally. You must use this switch in conjunction with the /L command switch or argument switch. The format is eight characters in four fields.

1. Columns 1-4 contain the line number of the source line.
2. Column 5 contains I if the line is in an include file, a space if it is not.
3. Column 6 contains one of the single characters

   C if the line was conditionally compiled
   * if the line is in a /*...*/, COMMENT ...; or END ..; comment
   " if the line is part of a string that extends over a line
   ' if it extends over a line and was conditionally compiled
   a space if the line fits none of the categories above

4. Column 7 indicates block level. If the block level is between 0 and 9, the appropriate digit appears here. If the block level is between 10 and 35, the appropriate uppercase letter from A to Z appears here. If the block level is between 36 and 61, the appropriate lowercase letter from a to b appears here. If the block level is above 61, an asterisk (*) appears here.
5. Column 8 contains a blank to separate the source line from the preceding seven columns.

/W

Produce warning messages. (This may produce a rather large output file.) One class of such messages warns you when the compiler makes a necessary modification to your program; for example, processing your REAL (15) number as a REAL (4). The other class informs you of potentially dangerous constructs in your program, such as passing constant parameter to a procedure by reference, risking destruction of the constant.

/WSAVS

Generate WSAVSs instead of WSAVRs. This checks for arithmetic overflows and division by 0. This switch is valid for the 32-bit AOS/VS environment only.

/X

Generate a full cross-reference table, including constants as well as variables.

/Y

Put constants in shared code partition, instead of shared data partition for AOS and AOS/VS-16 programs with overlays.

/Z

If generating code for AOS, 16-bit AOS/VS, RDOS ECLIPSE computers, or RDOS NOVA computers, assume EXTERNAL integers, pointers, and Booleans are located in the ZREL partition.

Argument Switches

Argument switches are equivalent, for the specified file, to command switches. In the RDOS environment, you must use argument switches if the corresponding command switch is anything other than a single letter.

filename/B

Same as /O=filename

code-option/C

Same as CODE=code-option

filename/E

Same as /E=filename

filename/L

Same as /L=filename

string/O

Same as /OPT=string

rev-no/R

Same as /REV=rev-no

Conditional Compilation

If you use the /O local or global OPT=string switch in the DG/L compiler command, you can direct that the compiler conditionally compile sections of your program. (Under RDOS, only the local /O switch is valid.) The conditional compilation mechanism closely resembles one variety of comment.

To direct that the compiler conditionally compile a program section, use the form

/**letters-digits spacings cond-text*/

- The letters-digits field contains one or more letters or digits. It may not contain any other characters.
- One or more spacing characters follows this field.
- The cond-text is the DG/L source text for the compiler to compile conditionally. The source text is terminated by a */.

The mechanism operates as follows: if you do not give an option list in the command line, the cond-text does not compile. If you give an option list in the command line, the compiler compares those characters with the characters in the letters-digits field. If it finds any of the characters in the option list in the letters-digits field, the cond-text is compiled into the program. Otherwise, it is treated as a comment.

For example,

```
/**AB3 CONCOM: I := I + 1;*/
```

compiles the statement labelled CONCOM into the surrounding DG/L program only if one or more of the characters A, B, or 3 occurs in the command line's option list.

You may also conditionally compile a section of code based on the absence of a letter or digit in the option list specified with the local /O or global /OPT=string switch. To do this, simply enter a minus sign (-) in the sequence of letters and digits following the /** delimiter. For example,

```
/**A-B3 CONCOM: I := I + 1;*/
```

the compiler passes over the statement labelled CONCOM if any one of A, B or 3 occurs in the option list. If none occur there, the statement compiles.

Letters in the string should be uppercase.

You may nest sections of conditional code. In this case, all enclosing sections must prove true for an inner conditional section to be compiled. For example

```
BEGIN STRING ST;
        EXTERNAL STRING PROCEDURE GETCOUTPUT;
        OPEN (1, (GETCOUTPUT));
        WRITE (1 "UNCONDITIONAL WRITE<NL>");
/**A
        WRITE (1, "CONDITIONAL ON A ONLY<NL>");
/**B
        WRITE (1, "CONDITIONAL ON A AND B<NL>");
                                                */
                                                */
        END;
```

The statement that writes "CONDITIONAL ON A AND B" will be compiled only if you specify both A and B in the option list, as in the command line

```
) X DGL PROG AB/O↵
```

If you specify neither A nor B, only the "UNCONDITIONAL WRITE" will be compiled, as in the command line

```
) X DGL/X PROG GRD/O↵
```

The same is true if you specify B without A, as in

```
) X DGL PROG BCDE/O↵
```

If you specify A without B, the "UNCONDITIONAL WRITE" and the "CONDITIONAL ON A ONLY" will be compiled, as in

```
) X DGL/A PROG AZS/O↵
```

Examples

```
) X/S DGL/G/L=test.LS/E=test.E/V test.DG↵
```

In this AOS or AOS/VS command line, the /S switch stores the program termination IPC message in STRING. The /G switch produces line number local symbols and emits all procedure names as declared. The /L switch sends listings to TEST.LS. The /E switch puts error listings in TEST.E. The /V switch includes block levels in the listing, and tells whether the line is from an include file, or whether it compiled conditionally.

```
R
DGL/F/V MRM ABC/O $LPT/L↵
```

In this RDOS command line, you compile MRM.DG (or MRM, if MRM.DG does not exist). By using the /F switch, you generate code without floating-point operations. The listing and error messages will go to $LPT (the line printer). By using the local /L switch, you can also use the /V switch to get block levels and data on includes and conditional compilation. The string ABC is a list of options for conditional compilation.

Linking

After you have successfully compiled source files, you can attempt to construct an executable DG/L program. Under AOS and AOS/VS, you do this by invoking the Link utility. The basic linking command line has the form

XEQ*//S/* LINK/NSLS*//switch...]* filename*//LOCAL] [DGLIBi]*

The /NSLS switch, which prevents the system from doing a system library search, must always appear in the DG/L LINK command line. Descriptions of other, optional switches appear in the section "Link Switches."

If you append the /LOCAL switch to the files you wish to link, and used the /G command switch at compilation time, you get source listing numbers in the symbol table.

DGLIBi is the appropriate system library for your target system. You must enclose your choice of DGLIBi in brackets in the command line.

Use	**to link**
DGLIB	default code environment: current environment
DGLIB16	16-bit AOS/VS programs
DGLIB16S	16-bit AOS/VS programs using shared-page cache memory
DGLIB32	32-bit AOS/VS programs
DGLIB32S	32-bit AOS/VS programs using shared-page cache memory
DGLIBA	AOS programs
DGLIBAS	AOS programs using shared-page cache memory
DGLIBF	AOS programs using FPC hardware instructions (M600, C350, and S250 models only)
DGLIBE	ECLIPSE RDOS programs
DGLIBN	NOVA RDOS programs

Link Switches

/NSLS

is always in the DG/L LINK command line. Since the system library (URT.LB) already appears in its correct order in the DGLIB macros, this "No System Library Search" switch prevents an unnecessary system library search.

/TASKS=n

allocates the correct number of TCBs (task control blocks) in a multitasking program. For details of multitasking, consult the *DG/L Runtime Library (AOS and AOS/VS) User's Manual.*

/CHANNELS=n

allows you to increase the maximum number of channels open at any one time beyond the default value of 16. The available channel numbers are 0 through n-1 (default 0 through 15). You cannot decrease the number of channels with this switch. To do that, you must modify DGLPARAM.SR. Consult the *DG/L Runtime Library (AOS and AOS/VS) User's Manual.*

/SYS=env

determines the environment you are linking for. The default compilation systems, for the current system, are RDOS for RDOS, AOS for AOS, and 16-bit AOS/VS for AOS/VS. Otherwise,

use	for programs compiled with
/SYS=VS32	/CODE=32 switch. Use DGLIB32 or DGLIB32S.
/SYS=VS16	/CODE=16 switch (or no switch). Use DGLIB, DGLIB16, or DGLIB16S.
/SYS=AOS	/CODE=A switch. Use DGLIBA, DGLIBAS, or DGLIBF.
/SYS=RDOS	/CODE=E switch for ECLIPSE. Use DGLIBE. /CODE=N switch for NOVA. Use DGLIBN.
/SYS=RTOS	/CODE=E switch for ECLIPSE and /CODE=N for NOVA. You must spell out the libraries since there is no one SYS.LB that contains the entire RTOS system.

The user of 32-bit AOS/VS must have a /CODE=VS32 switch on DG/L in the compilation command line, and may have a /SYS=VS32 switch on LINK in the link command line.

The user of 16-bit AOS/VS may have a /CODE=VS16 switch on DG/L in the compilation command line, and must have a /SYS=VS16 switch on LINK in the link command line.

The RDOS DG/L libraries are supplied in .OB format for use on AOS/VS with LINK for cross-linking. You must convert the SYS.LB of the RDOS environment from .RB format to .OB format in order to link with AOS/VS. Follow these steps:

1. Under RDOS, use LFE to list the titles of the library routines in the selected library.
2. Still under RDOS, use LFE to extract each of the .RB ibrary routines.
3. Dump the .RB files to tape.
4. Load them onto your AOS or AOS/VS system using the program RDOS with the LOAD command.
5. Convert them to .OB format using CONVERT.
6. Use the AOS LFE to remerge the routines into the system library in the original order.

Using Clusters and External Procedures

You must specify in the LINK command line all EXTERNAL procedures that your main program calls. Each procedure must follow the main program name and precede the library macro.

You must also specify in the LINK command line any CLUSTER that your main program calls. Each cluster must follow the main program name and precede the library macro.

You may intersperse clusters and separate EXTERNAL procedures at will. A LINK command line containing EXTERNAL procedures, clusters, or both has the form (optional switches omitted)

XEQ LINK/NSLS main-prog [*ext-proc* / *cluster*] ... [DGLIBi]

Using Overlays

You wait until link time to decide which procedures to put in overlays. Desginating overlays at link time ("loading on call") makes creating overlays fast and simple, and eliminates the need to recompile programs if you decide to modify an initial overlay structure.

Specify overlays after the main program and any clusters or external procedures, and before the library macro.

The delimiters for overlays and overlay areas must be separated from other material with spaces. Each overlay area is delimited by !*□...□*! and adjacent overlays within an overlay area are separated by □!□. A link command line with two overlay areas, the first with two overlays and the second with three, would have the form (omitting any optional switches, external procedures, or clusters)

) XEQ LINK/NSLS main-prog !*□A B C□!□D□*!□E□!□F G□!□H□*! [DGLIBi]

For strategies in the use of overlays, consult the *DG/L Runtime Library (AOS and AOS/VS) User's Manual.*

Building an Executable Save File under RDOS

Under RDOS, use the Relocatable Binary Loader (RLDR) to load the output from the compiler. You must load the DG/L runtime library and any EXTERNAL procedures or clusters your main program calls. The file DGLIB is supplied with the compiler to load the runtime library in correct order. The /N switch is necessary to prevent a system library search. The optional /C switch builds a file for RTOS. The form of the command line in summary is

RLDR/N//*C*/main-prog [*ext-proc* / *cluster*] ... @DGLIB@

For a more detailed discussion of the RLDR command, including optional arguments and overlay specifications, refer to the *RDOS/DOS Command Line Interpreter User's Manual.*

Code Generation under RDOS

To create an executable save file under RDOS, you might type one of

RLDR/N filename @DGLIBE@
RLDR/N filename @DGLIBN@

The files DGLIBN and DGLIBE specify the libraries you will use when creating your executable program (save) files. The file DGLIB is identical to DGLIBE or DGLIBN, whichever is appropriate in your current environment.

End of Chapter

Appendix A
DG/L and ALGOL 60

The DG/L programming language is based on the ALGOL 60 programming language. It has the block structure and most of the syntax of ALGOL 60. It also gives you data types, operations, and statements that ALGOL 60 lacks. A large number of runtime routines complement the DG/L language.

Paragraph numbers in the following refer to Peter Naur et al., "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM*, 6 (Jan., 1963),1-17.

I. ALGOL 60 features not supported in the DG/L language include

1. Call by name (para. 4.7.3.2).
2. Nested quotation marks in literals (paras. 2.6.1 and 2.4.1).
3. No maximum length for identifiers. DG/L has a maximum significant length for identifiers of 32 characters.
4. Two division operators (see II.1. below). The DG/L language uses only / for both integer and real division, and does not use ÷ (para. 3.3.4.2).
5. Revised Report notation of 10 <integer> (para. 2.5.1 and II.2. below).
6. A program consisting simply of a <compound statement> (para. 4.1.1).
7. Boolean operators as the single characters (para. 3.4.6.1 and II.3.). $\wedge$, $\vee$, $\neg$, $\supset$, $\equiv$, $\leq$, $\geq$, $\neq$
8. non-DG/L code in a DG/L program (para. 4.7.8).

II. ALGOL 60 features functionally the same but syntactically different in the DG/L language are

1. The single DG/L division operator does all division as real except between integers, which it does as integer division. If you use the global /R switch at compilation time, however, it carries out all division as real.
2. You may use either E <integer> or D <integer> for scientific notation.
3. The DG/L Boolean operators are AND, EQV, IMP, NOT, OR, XOR, <= or =<, >= or =>, and <> or ><.

III. Extensions are

1. String operations
   a. SUBSTR, INDEX, LENGTH, SETCURRENT.
   b. Arithmetic operations with strings.
   c. concatenation operator (!!).
   d. Type conversions (Boolean, Integer, Real, Pointer, Bit).
   e. Octal numbers and some special symbols enclosed in brackets to represent ASCII characters.
2. Input and Output
   a. Fully formatted output, unformatted input, and output for all supported data types.
   b. Full interface to RDOS, AOS, and AOS/VS system calls.
   c. Cache memory management with optional virtual memory capability.
   d. INFOS® II and CLRE interfaces.
3. Data definition and access
   a. BASED variables of all data types.
   b. Literal capability for generalized expression definition: LITERAL name <any expression> interprets the name as the expression.
   c. Global versus local data access.
   d. 2- and 4-word real numbers.
   e. 1- and 2-word integers.
   f. BIT string data type.
   g. BYTE on left-hand side of Assignment expression.
   h. EXTERNAL procedures and data.
   i. GLOBAL declaration.
   j. CLUSTER declaration.

4. Statements

   a. Parametric procedures.

   b. Generalized DO:

$$\left[\begin{Bmatrix}WHILE\\UNTIL\end{Bmatrix} \quad <boolexp> \right] \text{ DO statement} \left[\begin{Bmatrix}WHILE\\UNTIL\end{Bmatrix} \quad <boolexp> \right] ;$$

   c. Expanded FOR (Chapter 4).

   d. INCLUDE that acts like CLI: given NAME, search for NAME.DG, then NAME. given NAME., search for NAME.

   e. Expanded IF

      IF boolexp THEN IF boolexp ...;

5. Commenting

   a. Conditional compilation of the form /**letter-digit-string statement*/. If any of the characters in the letter-digit-string appear in the command line with a local /O or global /OPT=string switch, the statements will compile; otherwise, they are treated as comments. A minus sign (-) in the letter-digit-string reverses these conditions.

   b. /* comments*/

   c. If a percent sign % appears on a line outside a quoted string the rest of the line is a comment.

6. Options

   a. Complete syntax check.

   b. Nonfloating-point output.

   c. Short LEF generation on mapped ECLIPSE systems.

   d. Integer division with reals.

   e. Subscript checking.

   f. Full cross-reference of constants.

   g. Reference of external integers on page zero.

   h. Nonprinting of INCLUDE files.

   i. Notation for block level, conditional execution, and INCLUDE file listing with global /V switch.

7. Operators
   a. !! string concatenator.
   b. XOR Boolean exclusive OR operator.
   c. -> connector between pointer and based item.
8. Available functions
   a. FORTRAN 5 math library.
   b. Multitasking support.
   c. Full interface to system functions.
   d. INFOS II interface.

End of Appendix

Appendix B
Compiler Error Messages

This appendix contains a list of DG/L compiler error codes and the corresponding messages. If the compiler discovers errors in your program, it displays the appropriate messages, and the line number on which the error occurred, on the terminal. It also writes the error messages to the list and error files you specify with the /L and /E compilation switches (Chapter 11).

The commercial at (@) symbol appears in many of the messages in this appendix. If any of these errors occur during compilation, the name of the item (overlay, function, variable, or the like) that caused the error replaces "@" in the error message.

Error Code	Message
1	ILLEGAL DIGIT FOR RADIX
2	FILENAME MUST END WITH ";"
3	INCLUDE FILE CAN'T BE OPENED
4	MISSING "*/" FOR A "/*"
5	MISSING END OF QUOTED STRING
6	MISSING "TO" AFTER "GO"
7	MISSING DIGITS IN A REAL CONSTANT
8	WARNING PRECISION OF A REAL CONSTANT NOT 2 OR 4
9	WARNING PRECISION OF AN INTEGER CONSTANT NOT 1 OR 2
10	SETTING OF PRECISION OF A CONSTANT MORE THAN ONCE
11	TOO MANY NESTED INCLUDE FILES
100	TOO MANY ITEMS IN A COMMA LIST
101	PROGRAM DOES NOT BEGIN PROPERLY
102	NOT ENOUGH "BEGIN"S OR PROGRAM DOESN'T END AT END STATEMENT
104	INSUFFICIENT NUMBER OF END STATEMENTS
105	STATEMENT BEGINS ILLEGALLY-- NEITHER IDENTIFIER NOR KEY WORD
106	MISSING OR ILLEGAL SUBSCRIPT FOR LABEL "@"
107	INCORRECT OR ILLEGAL LABEL BEFORE ":"
108	IDENTIFIER BEGINNING WITH "@" IS AN ILLEGAL LABEL
109	ILLEGAL EXPRESSION AFTER THE "IF"

114	MISSING "THEN" IN IF STATEMENT
115	IDENTIFIER DOES NOT FOLLOW "ARRAY" IN DECLARATION
118	EXPRESSION MAY NOT BE USED AS A STATEMENT
119	ILLEGAL SUBSCRIPT OR MISSING "]" FOR LABEL "@"
120	INCORRECT VARIABLE AFTER THE "FOR"
121	MISSING ":=" AFTER THE FOR VARIABLE
122	MISSING OR ILLEGAL EXPRESSION AFTER ":=" IN A FOR STATEMENT
124	MISSING OR ILLEGAL FOR LIST OR MISSING "DO"
125	MISSING ":" AFTER A STATEMENT NUMBER
126	ILLEGAL OR MISSING EXPRESSION AFTER THE "STEP"
127	MISSING "UNTIL" IN A "STEP" FOR LIST CLAUSE
128	ILLEGAL OR MISSING EXPRESSION AFTER THE "UNTIL"
129	ILLEGAL VARIABLE LIST FOR DECLARATION OR MISSING ";"
133	MISSING OR ILLEGAL FIRST IDENTIFIER IN DECLARATION
134	MISSING OR ILLEGAL FIRST IDENTIFIER FOR "EXTERNAL" DECLARATION
135	"OWN" OR "BASED" IS NOT FOLLOWED BY A DATA TYPE
136	ILLEGAL OR MISSING PROCEDURE NAME AFTER "PROCEDURE"
139	MISSING OR ILLEGAL BOUNDS LIST FOR ARRAY VARIABLE "@"
140	ILLEGAL DIMENSION LIST OR MISSING "]" FOR ARRAY VARIABLE "@"
141	MISSING OR ILLEGAL VALUE FOR HIGH BOUND
143	MISSING OR ILLEGAL EXPRESSION FOLLOWING THE OPERATOR "@"
144	ILLEGAL EXPRESSION AFTER THE "IF"
145	MISSING "THEN" IN THE IF EXPRESSION
146	ILLEGAL EXPRESSION AFTER THE "THEN"
147	AN IF EXPRESSION MUST HAVE AN "ELSE" CLAUSE
148	ILLEGAL EXPRESSION AFTER THE "ELSE"
149	MISSING ")" IN THE EXPRESSION
150	MISSING OR ILLEGAL FIRST SUBSCRIPT FOR VARIABLE "@"
151	ILLEGAL SUBSCRIPT LIST OR MISSING "]" FOR VARIABLE "@"
152	MISSING OR ILLEGAL FIRST ARGUMENT EXPRESSION FOR FUNCTION "@"
153	ILLEGAL ARGUMENT LIST OR MISSING ")" FOR FUNCTION "@"

154 MISSING OR ILLEGAL STATEMENT LABEL AFTER "THEN"

155 CONDITIONAL GO TO MUST HAVE AN ELSE CLAUSE

156 MISSING OR ILLEGAL STATEMENT LABEL AFTER THE "ELSE"

157 MISSING OR ILLEGAL EXPRESSION AFTER THE "("

158 MISSING OR ILLEGAL EXPRESSION AFTER THE "UNTIL" OR "WHILE"

163 MISSING OR ILLEGAL ARGUMENT EXPRESSION AFTER A "("

164 MISSING OR ILLEGAL IDENTIFIER AFTER "LITERAL"

165 MISSING LABEL OR MISSING IF GOTO EXPRESSION

168 MISSING "(" AFTER LITERAL IDENTIFIER "@"

170 MISSING ")" ENDING LITERAL IDENTIFIER "@"

172 PARAMETER OF PROCEDURE IS NOT AN IDENTIFIER

173 ILLEGAL PARAMETER LIST OR MISSING ")" LIST TERMINATOR

175 MISSING OR INCORRECT IDENTIFIER AFTER "VALUE"

176 MISSING ")" ENDING PRECISION VALUE

177 MISSING OR ILLEGAL VALUE FOR PRECISION IN THE DECLARATION

178 INVALID PARAMETER LIST FOR PROCEDURE "@" OR MISSING ";"

179 ILLEGAL VALUE ASSIGNED TO LITERAL "@"

181 MISSING DO IN AN "UNTIL" OR "WHILE" STATEMENT

190 ILLEGAL STATEMENT OR MISSING ";", "END", OR "ELSE"

194 PROCEDURE TYPE WITHOUT "PROCEDURE"

196 MISSING "(" IN I/O STATEMENT

197 MISSING OR ILLEGAL CHANNEL NUMBER IN I/O STATEMENT

198 MISSING "," AFTER THE CHANNEL NUMBER IN AN I/O STATEMENT

199 ILLEGAL FORMAT STRING IN I/O STATEMENT

200 ILLEGAL OR MISSING EXPRESSION AT START OF I/O LIST

201 MISSING "," OR ")" IN I/O STATEMENT

202 MISSING "(" IN SUBSTR CALL

203 MISSING OR INVALID STRING EXPRESSION IN SUBSTR CALL

204 MISSING "," AFTER THE STRING EXPRESSION IN A SUBSTR CALL

205 MISSING OR INVALID BOUND TO SUBSTR CALL

206 MISSING ")" IN SUBSTR CALL

207 MISSING OR ILLEGAL IDENTIFIER AFTER "SWITCH"

208 MISSING ":=" IN SWITCH DECLARATION

209	MISSING OR ILLEGAL LABEL OR IF GOTO AFTER ":="
211	"ELSE" OUTSIDE OF AN "IF" STATEMENT
212	ILLEGAL I/O LIST OR MISSING ")"
250	ILLEGAL NUMBER OF ARGUMENTS IN PROCEDURE CALL
251	ILLEGAL CLASS TYPE FOR VARIABLE "@"
252	NO REAL CONSTANTS WITH NO-FLOATING-POINT GLOBAL SWITCH
253	INVALID REAL NUMBER FORMAT "@"
255	MULTIPLE DEFINITION OF EXTERNAL VARIABLE "@"
300	ILLEGAL DECLARATION, "@" IS NOT A FORMAL PARAMETER
301	ILLEGAL TYPE CONVERSION FOR "@"
302	ILLEGAL TYPE CONVERSION
303	TOO MANY SUBSCRIPTS FOR ARRAY "@"
304	NOT ENOUGH SUBSCRIPTS FOR ARRAY "@"
305	ILLEGAL SUBSCRIPTED VARIABLE "@"
306	WRONG NUMBER OF ARGUMENTS FOR FUNCTION "@"
307	VARIABLE "@" HAS BEEN DECLARED AN ARRAY MORE THAN ONCE
308	VARIABLE IS UNDEFINED
309	ILLEGAL ASSIGNMENT TO DIMENSIONED VARIABLE "@"
310	ILLEGAL ASSIGNMENT TO "@"
311	ILLEGAL ASSIGNMENT TO AN EXPRESSION OR A PROCEDURE VARIABLE
312	ILLEGAL USE OF DIMENSIONED VARIABLE "@"
313	DECLARED SUBSCRIPTED LABEL MUST HAVE INTEGER SUBSCRIPT
314	VARIABLE "@" HAS BEEN DEFINED MORE THAN ONCE IN SAME BLOCK
315	VARIABLE "@" HAS BEEN DECLARED A VALUE WITHOUT A TYPE
317	ILLEGAL USE OF PROCEDURE VARIABLE "@" IN AN EXPRESSION
318	ILLEGAL ASSIGNMENT TO PROCEDURE NAME "@"
319	SUBSCRIPTED VARIABLE DECLARED OTHER THAN LABEL OR ARRAY
320	MISSING OR ILLEGAL DIMENSION FOR AN ARRAY
321	ILLEGAL INDEXING VARIABLE IN A FOR LOOP
322	ILLEGAL LABEL "@". IT HAS BEEN DEFINED PREVIOUSLY
323	DUPLICATE LABEL DEFINITION OF LABEL "@"
324	ALL FORMAL PARAMETERS OF PROCEDURE "@" ARE UNDEFINED

325	FORMAL PARAMETER "@" IS UNDEFINED
326	VARIABLE "@" HAS NOT BEEN DECLARED
327	ILLEGAL ASSIGNMENT TO A PROCEDURE VARIABLE "@"
328	VARIABLE "@" IS DEFINED AS OTHER THAN A FUNCTION CALL
329	ATTEMPT TO TRANSFER TO AN ILLEGAL LABEL
330	ILLEGAL USE OF A LABEL
331	CANNOT READ INTO A CONSTANT OR STRING LITERAL OR EXPRESSION
332	VARIABLE IN SWITCH LIST IS NOT A LABEL
333	VARIABLE "@" HAS NOT BEEN DECLARED AN ARRAY
334	RECURSION ERROR FOR LITERAL "@" DECLARATION
335	LABEL "@" NOT ASSIGNED POSITION IN BLOCK
336	PROGRAM CAN NOT BEGIN AS A COMPOUND STATEMENT
337	ILLEGAL SHAPE OF ARGUMENT
338	ILLEGAL TYPE CONVERSION OF ARGUMENT
339	VARIABLE IS NOT DECLARED BASED
340	REAL VARIABLE "@" WITH NO-FLOATING-POINT GLOBAL SWITCH
341	WARNING PRECISION OF REAL VARIABLE NOT 2 OR 4
342	WARNING PRECISION OF INTEGER VARIABLE NOT 1 OR 2
343	WARNING AUTOMATIC TYPE CONVERSION HAS BEEN PERFORMED
344	WARNING GENERIC SELECTION ON FUNCTION "@" WAS PERFORMED
345	WARNING PASSING MORE VARIABLES THAN EXPECTED BY PROCEDURE
346	WARNING PASSING FEWER VARIABLES THAN EXPECTED BY PROCEDURE
347	ILLEGAL USE OF A BASED VARIABLE
348	WARNING ARGUMENT TO PROCEDURE IS A CONSTANT
349	WARNING EXPRESSION OR VARIABLE IS A GLOBAL REFERENCE
351	ILLEGAL PARAMETER LIST FOR PROCEDURE
352	VARIABLE "@" HAS NOT BEEN INITIALIZED
353	ILLEGAL USE OF AN OVERLAY NAME "@"
354	ILLEGAL ASSIGNMENT TO OVERLAY NAME "@"
355	ILLEGAL OPERATION FOR "@"
356	ILLEGAL USE OF BUILT-IN FUNCTION AS PROCEDURE

357	ERROR: TOO MANY EXTERNALS
358	NO LOCAL VARIABLES IN A CLUSTER
370	MORE THAN 8191 LINES IN A PROGRAM USING /DEBUG
371	MORE THAN 254 INCLUDE FILES IN A PROGRAM USING /DEBUG
372	MORE THAN 8 EXECUTABLE STATEMENTS PER LINE IN A PROGRAM USING /DEBUG
373	TOO MANY STATEMENTS AND/OR FILES IN A PROGRAM USING /DEBUG
374	LABEL AND ITS STATEMENT ARE IN DIFFERENT FILES IN A PROGRAM USING /DEBUG
401	WARNING EXPRESSION OR VARIABLE IS A GLOBAL REFERENCE
402	STATEMENT TOO COMPLEX, CAUSING COMPILER STACK OVERFLOW
403	ILLEGAL OR MISSING CLUSTER NAME AFTER "CLUSTER"
404	INVALID PARAMETER LIST FOR CLUSTER "@" OR MISSING ";"
405	TOO COMPLEX: COMPILER STACK OVERFLOW IN PHASE 2
406	TOO COMPLEX: COMPILER STACK OVERFLOW IN PHASE 3
407	TOO COMPLEX: COMPILER STACK OVERFLOW IN PHASE 4

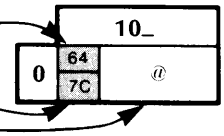
End of Appendix

Appendix C
ASCII Character Sets

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column.

LEGEND:

Character code in decimal
EBCDIC equivalent hexadecimal code
Character



OCTAL	00_		01_		02_		03_		04_		05_		06_		07_	
0	0 00	NUL	8 16	BS (BACK-SPACE)	16 10	DLE ↑P	24 18	CAN ↑X	32 40	SPACE	40 4D	(	48 F0	0	56 F8	8
1	1 01	SOH ↑A	9 05	HT (TAB)	17 11	DC1 ↑Q	25 19	EM ↑Y	33 5A	!	41 5D	)	49 F1	1	57 F9	9
2	2 02	STX ↑B	10 15	NL (NEW LINE)	18 12	DC2 ↑R	26 3F	SUB ↑Z	34 7F	" (QUOTE)	42 5C	*	50 F2	2	58 7A	:
3	3 03	ETX ↑C	11 0B	VT (VERT TAB)	19 13	DC3 ↑S	27 27	ESC (ESCAPE)	35 7B	#	43 4E	+	51 F3	3	59 5E	;
4	4 37	EOT ↑D	12 06	FF (FORM FEED)	20 3C	DC4 ↑T	28 1C	FS ↑\	36 5B	$	44 6B	, (COMMA)	52 F4	4	60 4C	<
5	5 2D	ENQ ↑E	13 0D	RT (RETURN)	21 3D	NAK ↑U	29 1D	GS ↑]	37 6C	%	45 60	-	53 F5	5	61 7E	=
6	6 2E	ACK ↑F	14 0E	SO ↑N	22 32	SYN ↑V	30 1E	RS ↑↑	38 50	&	46 4B	. (PERIOD)	54 F6	6	62 6E	>
7	7 2F	BEL ↑G	15 0F	SI ↑O	23 26	ETB ↑W	31 1F	US ↑←	39 7D	' (APOS)	47 61	/	55 F7	7	63 6F	?

OCTAL	10_		11_		12_		13_		14_		15_		16_		17_	
0	64 7C	@	72 C8	H	80 D7	P	88 E7	X	96 79	` (GRAVE)	104 88	h	112 97	p	120 A7	x
1	65 C1	A	73 C9	I	81 D8	Q	89 E8	Y	97 81	a	105 89	i	113 98	q	121 A8	y
2	66 C2	B	74 D1	J	82 D9	R	90 E9	Z	98 82	b	106 91	j	114 99	r	122 A9	z
3	67 C3	C	75 D2	K	83 E2	S	91 8D	[	99 83	c	107 92	k	115 A2	s	123 C0	{
4	68 C4	D	76 D3	L	84 E3	T	92 E0	\	100 84	d	108 93	l	116 A3	t	124 4F	|
5	69 65	E	77 D4	M	85 E4	U	93 9D	]	101 85	e	109 94	m	117 A4	u	125 D0	}
6	70 C6	F	78 D5	N	86 E5	V	94 5F	↑ or ^	102 86	f	110 95	n	118 A5	v	126 A1	~ (TILDE)
7	71 C7	G	79 D6	O	87 E6	W	95 6D	← or _	103 87	g	111 96	o	119 A6	w	127 07	DEL (RUBOUT)

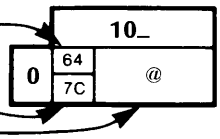
SD-00217 Character code in octal at top and left of charts.

↑ means CONTROL

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column.

LEGEND:

Character code in decimal
EBCDIC equivalent hexadecimal code
Character



OCTAL	00_		01_		02_		03_		04_		05_		06_		07_	
0	0 00	NUL	8 16	BS (BACK-SPACE)	16 10	↑P	24 18	↑X	32 40	SPACE	40 4D	(	48 F0	0	56 F8	8
1	1 01	↑A	9 05	HT (TAB)	17 11	↑Q	25 19	↑Y	33 5A	!	41 5D	)	49 F1	1	57 F9	9
2	2 02	↑B	10 15	LINE FEED	18 12	↑R	26 3F	↑Z	34 7F	" (QUOTE)	42 5C	*	50 F2	2	58 7A	:
3	3 03	↑C	11 0B	VT (VERT. TAB)	19 13	↑S	27 27	ESC (ESCAPE)	35 7B	#	43 4E	+	51 F3	3	59 5E	;
4	4 37	↑D	12 06	FF (FORM FEED)	20 3C	↑T	28 1C	↑\	36 5B	$	44 6B	, (COMMA)	52 F4	4	60 4C	<
5	5 2D	↑E	13 0D	CR (RETURN)	21 3D	↑U	29 1D	↑]	37 6C	%	45 60	-	53 F5	5	61 7E	=
6	6 2E	↑F	14 0E	↑N	22 32	↑V	30 1E	↑↑	38 50	&	46 4B	. (PERIOD)	54 F6	6	62 6E	>
7	7 2F	BELL ↑G	15 0F	↑O	23 26	↑W	31 1F	↑←	39 7D	' (APOS)	47 61	/	55 F7	7	63 6F	?

OCTAL	10_		11_		12_		13_		14_		15_		16_		17_	
0	64 7C	@	72 C8	H	80 D7	P	88 E7	X	96 79	` (GRAVE)	104 88	h	112 97	p	120 A7	x
1	65 C1	A	73 C9	I	81 D8	Q	89 E8	Y	97 81	a	105 89	i	113 98	q	121 A8	y
2	66 C2	B	74 D1	J	82 D9	R	90 E9	Z	98 82	b	106 91	j	114 99	r	122 A9	z
3	67 C3	C	75 D2	K	83 E2	S	91 8D	[	99 83	c	107 92	k	115 A2	s	123 C0	{
4	68 C4	D	76 D3	L	84 E3	T	92 E0	\	100 84	d	108 93	l	116 A3	t	124 4F	|
5	69 65	E	77 D4	M	85 E4	U	93 9D	]	101 85	e	109 94	m	117 A4	u	125 D0	}
6	70 C6	F	78 D5	N	86 E5	V	94 5F	↑ or ^	102 86	f	110 95	n	118 A5	v	126 A1	~ (TILDE)
7	71 C7	G	79 D6	O	87 E6	W	95 6D	← or _	103 87	g	111 96	o	119 A6	w	127 07	DEL (RUBOUT)

SD-00476 Character code in octal at top and left of charts.

↑ means CONTROL

End of Appendix

Appendix D
Octal and Hexadecimal Conversion

	8^1	8^2	8^3	8^4	8^5	8^0
0	0	0	0	0	0	0
1	32,768	4,096	512	64	8	8
2	65,536	8,192	1,024	128	16	16
3	98,304	12,228	1,536	192	24	24
4	131,072	16,384	2,048	256	32	32
5	163,840	20,480	2,560	320	40	40
6	196,608	24,576	3,072	384	48	48
7	229,376	28,672	3,584	448	56	56

SD-00973

	16^5	16^4	16^3	16^2	16^1	16^0
0	0	0	0	0	0	0
1	1,048,576	65,536	4,096	256	16	1
2	2,097,152	131,072	8,192	512	32	2
3	3,145,728	196,608	12,288	768	48	3
4	4,194,304	327,680	16,384	1,024	64	4
5	5,242,880	393,216	20,480	1,280	80	5
6	6,291,456	458,752	24,576	1,536	96	6
7	7,340,032	524,288	28,672	1,792	112	7
8	8,388,608	589,824	32,768	2,048	128	8
9	9,437,184	655,360	36,864	2,304	144	9
A	10,485,760	720,896	40,960	2,560	160	10
B	11,534,336	786,432	45,056	2,816	176	11
C	12,582,912	851,968	49,152	3,072	192	12
D	13,631,488	917,504	53,248	3,328	208	13
E	14,680,064	983,040	57,344	3,584	224	14
F	15,728,640		61,440	3,840	240	15

SD-00974

End of Appendix

Index

Nonalphabetic entries precede all others and follow the ASCII sequence of their symbols. Page numbers in italics indicate major entries. The letter "f" following a page entry means "and the following page;" "ff" means "and the following pages."

A

B

C

G

H

I

K

L

M

N

S

T

U

CUT ALONG DOTTED LINE

Data General

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government
- ☐ Educational

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other _____		
(Specify) _____		
_____		

3. Software

- ☐ AOS
- ☐ DOS
- ☐ MP/OS
- ☐ RDOS
- ☐ Other

Specify _____

4. Languages

- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communications

- ☐ HASP
- ☐ RJE80
- ☐ RCX 70
- ☐ CAM
- ☐ XODIAC
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

Data General

Data General Corporation, Westboro, Massachusetts 01580. (617) 366-8911

FOLD

TAPE

FOLD

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

Data General

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government
- ☐ Educational

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other (Specify) _____		
_____		
_____		

3. Software

- ☐ AOS
- ☐ DOS
- ☐ MP/OS
- ☐ RDOS
- ☐ Other.

Specify _____

4. Languages

- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communications

- ☐ HASP
- ☐ RJE80
- ☐ RCX 70
- ☐ CAM
- ☐ XODIAC
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

Data General

Data General Corporation, Westboro, Massachusetts 01580. (617) 366-8911

CUT ALONG DOTTED LINE

FOLD

TAPE

FOLD

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

ISD User Documentation Remarks Form

Your Name _____

Your Title _____

Company _____

Street _____

City _____ State _____ Zip _____

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title *DG/L™ Language Reference Manual* Manual No. *093-000229-01*

Who are you?

- ☐ EDP Manager
- ☐ Senior Systems Analyst
- ☐ Other _____
- ☐ Analyst/Programmer
- ☐ Operator

What programming language(s) do you use? _____

How do you use this manual? *(List in order: 1 = Primary Use)*

- ___ Introduction to the product
- ___ Reference
- ___ Other _____
- ___ Tutorial Text
- ___ Operating Guide

About the manual:	Yes	Somewhat	No
Is it easy to read?	☐	☐	☐
Is it easy to understand?	☐	☐	☐
Are the topics logically organized?	☐	☐	☐
Is the technical information accurate?	☐	☐	☐
Can you easily find what you want?	☐	☐	☐
Does it tell you everything you need to know?	☐	☐	☐
Do the illustrations help you?	☐	☐	☐

If you have any comments on the software itself, please contact your Data General Systems Engineer. If you wish to order manuals, see your Data General Sales Representative.

Remarks:

Date _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral

ISD User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

Data General Corporation, Westboro, MA 01580