decsystem10/20

ALGOL PROGRAMMER'S GUIDE

AA-0196C-TK

digital equipment corporation · maynard. massachusetts

First Printing: September 1971
Revised: September 1971
December 1971
May 1972
December 1972
July 1973
July 1974
May 1975
September 1975
Second Printing: March 1976
Revised: April 1977

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECSYSTEM-20	TYPESET-11

CONTENTS

			Page
CHAPTER	1	INTRODUCTION	
	1.1	GENERAL	1-1
	1.2	DECSYSTEM-10/20 ALGOL	1-1
	1.3	THE ALGOL COMPILER	1-1
	1.3.1	Compiler Extensions	1-2
	1.3.2	Compiler Restrictions	1-2
	1.4	THE ALGOL OPERATING ENVIRONMENT	1-3
	1.5	TERMINOLOGY	1-3
CHAPTER	2	PROGRAM STRUCTURE	
	2.1	BASIC SYMBOLS	2-1
	2.2	COMPOUND SYMBOLS	2-2
	2.3	DELIMITER WORDS	2-2
	2.4	USE OF SPACING AND COMMENTARY	2-4
CHAPTER	3	IDENTIFIERS AND DECLARATIONS	
	3.1	IDENTIFIERS	3-1
	3.2	SCALAR DECLARATIONS	3-2
CHAPTER	4	CONSTANTS	
	4.1	NUMERIC CONSTANTS	4-1
	4.1.1	Integer Constants	4-1
	4.1.2	Real Constants	4-1
	4.1.3	Long Real Constants	4-2
	4.1.3.1	Automatic Conversion of Real Constants to Long Real Constants	4-2
	4.2	OCTAL AND BOOLEAN CONSTANTS	4-3
	4.3	ASCII CONSTANTS	4-3
	4.4	STRING CONSTANTS	4-3
CHAPTER	5	EXPRESSIONS	
	5.1	ARITHMETIC EXPRESSIONS	5-1
	5.1.1	Identifiers and Constants	5-2
	5.1.2	Special Functions	5-2
	5.2	BOOLEAN EXPRESSIONS	5-3
	5.2.1	Boolean Operators	5-3
	5.2.2	Evaluation of Boolean Variables	5-4
	5.2.3	Arithmetic Conditions	5-4
	5.3	INTEGER AND BOOLEAN CONVERSIONS	5-6
CHAPTER	6	STATEMENTS AND ASSIGNMENTS	
	6.1	STATEMENTS	6-1
	6.2	ASSIGNMENTS	6-1
	6.3	MULTIPLE ASSIGNMENTS	6-2
	6.4	EVALUATION OF EXPRESSIONS	6-2

	6.5	COMPOUND STATEMENTS	6-3
CHAPTER	7	CONTROL TRANSFERS, LABELS, AND CONDITIONAL STATEMENTS	
	7.1	LABELS	7-1
	7.2	UNCONDITIONAL CONTROL TRANSFERS	7-1
	7.3	CONDITIONAL STATEMENTS	7-2
CHAPTER	8	FOR AND WHILE STATEMENTS	
	8.1	FOR STATEMENTS	8-1
	8.1.1	STEP-UNTIL Element	8-2
	8.1.2	WHILE Element	8-3
	8.2	WHILE STATEMENT	8-3
	8.3	GENERAL NOTES	8-3
CHAPTER	9	ARRAYS	
	9.1	GENERAL	9-1
	9.2	ARRAY DECLARATIONS	9-1
	9.3	ARRAY ELEMENTS	9-2
CHAPTER	10	BLOCK STRUCTURE	
	10.1	GENERAL	10-1
	10.2	ARRAYS WITH DYNAMIC BOUNDS	10-3
CHAPTER	11	PROCEDURES	
	11.1	PARAMETERS CALLED BY "VALUE"	11-1
	11.2	PARAMETERS CALLED BY "NAME"	11-1
	11.3	PROCEDURE HEADINGS	11-2
	11.4	PROCEDURE BODIES	11-3
	11.5	PROCEDURE CALLS	11-5
	11.6	ADVANCED USE OF PROCEDURES	11-6
	11.6.1	Jensen's Device	11-6
	11.6.2	Recursion	11-6
	11.7	LAYOUT OF DECLARATIONS WITHIN BLOCKS	11-7
	11.8	FORWARD REFERENCES	11-8
	11.9	EXTERNAL PROCEDURES	11-9
	11.10	ADDITIONAL METHODS OF COMMENTARY	11-10
	11.10.1	Comment After END	11-10
	11.10.2	Comments Within Procedure Headings	11-10
CHAPTER	12	SWITCHES	
	12.1	GENERAL	12-1
	12.2	SWITCH DECLARATIONS	12-1
	12.3	USE OF SWITCHES	12-1
CHAPTER	13	STRINGS	
	13.1	GENERAL	13-1
	13.2	STRING EXPRESSIONS AND ASSIGNMENTS	13-1
	13.3	BYTE STRINGS	13-1
	13.4	BYTE SUBSCRIPING	13-2
	13.5	NULL STRINGS	13-2
	13.6	STRING COMPARISONS	13-3
	13.7	LIBRARY PROCEDURES	13-3
	13.7.1	Concatenation	13-3
	13.7.2	Length and Size	13-3
	13.7.3	Copying	13-4
	13.7.4	Newstring	13-4

	13.7.5	Delete	13-4
CHAPTER	14	CONDITIONAL EXPRESSIONS AND STATEMENTS	
	14.1	GENERAL	14-1
	14.2	CONDITIONAL OPERANDS	14-1
	14.3	CONDITIONAL STATEMENTS	14-2
	14.4	DESIGNATIONAL EXPRESSIONS	14-4
CHAPTER	15	OWN VARIABLES	
	15.1	GENERAL	15-1
	15.2	OWN ARRAYS	15-1
CHAPTER	16	DATA TRANSMISSION	
	16.1	GENERAL	16-1
	16.2	ALLOCATION OF PERIPHERAL DEVICES	16-1
	16.2.1	Device Modes	16-2
	16.2.2	Buffering	16-3
	16.2.3	Error Returns	16-3
	16.3	SELECTING INPUT/OUTPUT CHANNELS	16-4
	16.4	FILE DEVICES	16-4
	16.4.1	Error Returns	16-5
	16.5	RELEASING DEVICES	16-5
	16.6	BASIC INPUT/OUTPUT PROCEDURES	16-6
	16.6.1	Byte Processing Procedures	16-6
	16.6.2	String Output	16-6
	16.6.3	Miscellaneous Symbol Procedures	16-8
	16.6.4	Numeric and String Procedures	16-8
	16.6.4.1	Numeric Input Data	16-8
	16.6.4.2	Numeric Output Data	16-9
	16.6.4.3	Octal Input/Output	16-10
	16.7	DEFAULT INPUT/OUTPUT	16-10
	16.8	LOGICAL INPUT/OUTPUT	16-11
	16.9	SPECIAL OPERATIONS	16-11
	16.10	I/O CHANNEL STATUS	16-12
	16.11	TRANSFERRING FILES	16-13
	16.12	CURRENTLY SELECTED CHANNEL NUMBERS	16-13
CHAPTER	17	THE DECSYSTEM-10/20 OPERATING ENVIRONMENT	
	17.1	MATHEMATICAL PROCEDURES	17-1
	17.2	STRING PROCEDURE	17-2
	17.3	UTILITY PROCEDURES	17-2
	17.3.1	Array Dimension Procedures	17-2
	17.3.2	Minima and Maxima Procedures	17-3
	17.3.3	Field Manipulations	17-3
	17.4	DATA TRANSMISSION PROCEDURES	17-4
	17.5	FORTRAN INTERFACE PROCEDURES	17-4
	17.6	GENERAL INFORMATION ROUTINE	17-5
	17.7	DATE AND TIME IN ASCII FORMAT	17-5
	17.8	RANDOM NUMBER ROUTINES	17-6
	17.9	ONTRACE AND OFFTRACE	17-6
	17.10	PAUSE	17-6
	17.11	DUMP	17-6
CHAPTER	18	RUNNING AND DEBUGGING PROGRAMS	
	18.1	COMPILATION OF ALGOL PROGRAMS	18-1
	18.1.1	Compilation of Free-Standing Procedures	18-4
	18.2	LOADING ALGOL PROGRAMS	18-4
	18.3	RUNNING ALGOL PROGRAMS	18-5
	18.4	CONCISE COMMAND LANGUAGE	18-5

	18.5	RUN-TIME DIAGNOSTICS AND DEBUGGING	18-5
	18.5.1	Facilities to Aid in Program Debugging	18-7
	18.5.1.1	Checking	18-7
	18.5.1.2	Controlling Listing of the Source Program	18-7
	18.5.1.3	Setting Line Numbers in Listings	18-8
	18.6	CROSS REFERENCE LISTING	18-8
	18.7	STACK ANALYSIS	18-8
	18.8	TRACE	18-9
	18.8.1	Dynamic Trace	18-9
	18.8.2	Post-Mortem Trace	18-11
	18.9	PERFORMANCE ANALYSIS	18-11
	18.9.1	Heap Space	18-11
	18.9.2	Code Utilization	18-12
CHAPTER	19	TECHNICAL NOTES	
CHAPTER	20	THE ALGOL DYNAMIC DEBUGGING SYSTEM	
	20.1	SUMMARY OF FEATURES	20-1
	20.2	GENERAL REMARKS	20-1
	20.3	TYPEOUT COMMANDS	20-2
	20.4	CHANGING ALGOL VARIABLES	20-6
	20.5	PAUSES	20-7
	20.6	EXECUTE COMMANDS	20-12
	20.7	DUMP	20-13
	20.8	MISCELLANEOUS COMMANDS	20-14
	20.9	SUMMARY OF COMMANDS	20-17
CHAPTER	21	THE DECSYSTEM-10 MACRO SUBROUTINES	
	21.1	GENERAL	21-1
	21.2	PROCEDURE HEADINGS	21-1
	21.3	ACCESSING FORMAL PARAMETERS	21-3
	21.4	RETURN OF RESULTS FROM TYPED PROCEDURES	21-5
	21.5	PROCEDURE EXITS	21-6
	21.6	FORMATS OF VARIABLES	21-6
	21.7	PROCEDURES WITH A VARIABLE NUMBER OF PARAMETERS	21-7
	21.8	INCLUDING PROCEDURE IN THE LIBRARY	21-8
	21.9	UTILITY ROUTINES	21-8
	21.9.1	Getting Core	21-8
	21.9.2	Input/Output	21-8
	21.9.2.1	Device Open	21-8
	21.9.2.2	File Open	21-9
	21.9.2.3	File Close	21-9
	21.9.2.4	Release Channel	21-9
	21.9.2.5	Select Channel	21-9
	21.9.2.6	Read Byte	21-9
	21.9.2.7	Write Byte	21-9
	21.9.2.8	Break Output	21-9
	21.9.2.9	Read Number	21-10
	21.9.2.10	Print Number	21-10
	21.9.2.11	String Output to Terminal	21-10
	21.10	GENERAL NOTES	21-10

TABLES

TABLE	2-1	DECsystem-10/20 ALGOL Symbols	2-1
	2-2	Compound Symbols	2-2
	2-3	Delimiter Words Used in DECsystem-10/20 ALGOL	2-3
	5-1	Operator Precedence	5-1

5-2	Function of Boolean Operators	5-4
5-3	Boolean Expressions	5-5
11-1	Parameters in a Procedure Call	11-1
16-1	Standard Device Names	16-2
17-1	FORTRAN Interface Procedures	17-4

CHAPTER 1

INTRODUCTION

1.1 GENERAL

DECsystem-10/20 ALGOL is an implementation of ALGOL-60; ALGOL is an abbreviation of ALGOrithmic Language, and 1960 is the year it was defined. The authoritative definition of ALGOL-60 is contained in the "Revised Report on the Algorithmic Language ALGOL-60",(1) hereafter referred to as the "Revised Report". This report leaves a number of ALGOL-60 features undefined, notably input/output, and permits the implementer of the language some latitude in interpreting other features. Many of these features have been discussed extensively since the publication of the Revised Report; some have been given rigorous interpretations in various versions of ALGOL, particularly the ALGOL-68 Language.(2)

Where there is need for interpretation in the Revised Report, such interpretations as seem reasonable have been made in light of current ALGOL opinion. Where no guidelines exist, ALGOL-68 is used as a basis. These points are discussed in Chapter 19.

1.2 DECSYSTEM-10/20 ALGOL

The purpose of this manual is to teach the use of DECsystem-10/20 ALGOL. The manual is written both for the user who is familiar with ALGOL implementations and for the user who has no knowledge of ALGOL but is reasonably fluent in a high-level scientific programming language such as FORTRAN IV. This manual is not a primer in high-level languages.(3)

Readers not thoroughly familiar with ALGOL should read the entire manual. Readers already familiar with ALGOL-60 should read all chapters except Chapters 5, 6, 7, 8, 9, 10, 11, 12, and 14, which need be referred to only briefly.

1.3 THE ALGOL COMPILER

The DECsystem-10/20 ALGOL Compiler is that part of the DECsystem-10/20 ALGOL System that reads programs written in DECsystem-10/20 ALGOL and converts them into a form (relocatable binary) that is acceptable to the DECsystem-10 or DECsytem-20 Linking Loader. The compiler is also responsible for finding errors in the user's source program and reporting them to the user.

Slight constraints are imposed on the way the user writes his program. These constraints, made to gain the most desirable feature of a single-pass compiler, concern the order in which the user declares the

identifiers in the program and the use of forward declarations under certain special circumstances.

Such a compiler can process ALGOL programs rapidly and does not require the use of any backing store. The minor restrictions imposed will not normally affect the user.

1.3.1 Compiler Extensions

The following ALGOL-60 extensions are allowed by the compiler:

1. A LONG REAL type, equivalent to FORTRAN's double precision, is added that gives the user power to handle double-precision real numbers.

2. An EXTERNAL procedure facility allows the user to compile procedures separately from the main program.

3. A WHILE statement, and an abbreviated form of the FOR statement, allow the user greater flexibility of iteration.

4. A new type STRING allows the user to manipulate strings of various size bytes. In addition, the user can individually manipulate the bytes within a string by means of a byte subscripting facility.

5. An integer remainder function REM, is provided.

6. Assignments are permitted within expressions.

7. Delimiter words may be represented in either reserved word format or as non-reserved words enclosed in single quotes (primes).

8. Constants of type REAL may be expressed as an integer part and a decimal part only as in FORTRAN.

The compiler accepts reserved word delimiters in normal mode, but can also accept programs using non-reserved delimiter words enclosed in primes. Refer to Chapter 18.

1.3.2 Compiler Restrictions

The compiler imposes the following restrictions on ALGOL-60:

1. Numeric labels are not permitted.

2. All formal parameters must be specified.

3. Identifiers are restricted to 64 characters in length.

4. Arrays and scalars must be declared before switches and procedures.

5. Forward references for procedures and labels must be given under certain circumstances.

For definitions of the terms used, refer to section 1.5 and to the Revised Report.

1.4 THE ALGOL OPERATING ENVIRONMENT

Programs compiled by the ALGOL compiler are run in a special operating environment that provides special services, including input/output facilities for the object program.

The ALGOL operating environment consists of:

1. The ALGOL Library, known as ALGLIB - a set of routines, some of which are incorporated into the user's program by the linking loader.

2. The ALGOL Object Time System, known as ALGOTS - responsible for organizing the smooth running of the program and providing services such as core management, peripheral device allocation, and fault monitoring in case the program encounters an error condition at run time.

Refer to Chapters 17 and 18 for a descripion of ALGLIB and ALGOTS.

1.5 TERMINOLOGY

Some of the following words, used in this manual, may be new to the reader. Many have a FORTRAN equivalent, and where such an equivalent exists, this is enclosed in parentheses.

Delimiter Word - a single, English language word that is an inherent part of the structure of the ALGOL language. Such words cannot normally be used for other purposes. Examples: BEGIN IF ARRAY.

Identifier - a name, established by user declaration, that represents some quantity within a program.

Label (Statement Number) - an identifier used to mark a certain statement in a program. Control of program execution can be transferred to the statement following the label. A numeric label which is similar to a FORTRAN statement number, is not available in DECsystem-10/20 ALGOL.

Procedure (Subroutine, Function) - part of a program, which may be invoked by "calling". In general, parameters are supplied as arguments and a result may be returned.

Parameter (Formal Parameter - Dummy Variable, Actual Parameter - Argument) See Procedure. - Formal Parameter is an identifier used within the procedure that represents the argument supplied when the procedure is called.

CHAPTER 2

PROGRAM STRUCTURE

2.1 BASIC SYMBOLS

DECsystem-10/20 ALGOL programs consist of a sequence of symbols from the DECsystem-10/20 ASCII character set. The meaning of individual characters given in Table 2-1, is much the same as in other high-level languages.

Table 2-1
DECsystem-10/20 ALGOL Symbols

Symbol	Meaning or Use
A-Z	Used to construct identifiers and delimiter words.
a-z	Lower case letters; are treated as upper case letters except when they appear in string constants and ASCII constants.
0-9	Decimal digits; used to construct numeric constants and identifiers.
+	Arithmetic addition operator.
-	Arithmetic subtraction operator.
*	Arithmetic multiplication operator.
/	Arithmetic division operator.
^	Arithmetic exponentiation operator.
( )	Parentheses; used in arithmetic expressions and to enclose parameters in procedure specifications and calls.
[ ]	Square brackets; used to enclose subscript bounds in array declarations, and array subscript lists.
,	Comma; general separator, placed between array subscripts, procedure parameters, items in switch lists, etc.
.	Decimal point; used in numeric constants and byte subscripting. Also, used as a readability symbol in identifiers.

Table 2-1 (Cont.)
DECsystem-10/20 ALGOL Symbols

Symbol	Meaning or Use
;	Semicolon; used to terminate statements.
:	Colon; used to indicate labels, and separate lower and upper bounds in array declarations.
=	Equality; used in arithmetic and string comparisons.
#	Nonequality.
< >	Less than, greater than.
& @	Introduces exponent in floating-point numbers.
'	Prime, or single quote; used to enclose delimiter words when the non-reserved word implementation is used.
"	Opening and closing string quotes.
!	Comment.
%	Introduces an octal constant.
$	Introduces an ASCII constant.
←	Alternative to := (refer to Table 2-2).

2.2 COMPOUND SYMBOLS

Compound symbols consist of two adjacent basic symbols. Any intervening spaces or tabs do not affect their use. The compound symbols are shown in Table 2-2.

Table 2-2
Compound Symbols

Symbol	Usage
:=	Assignment
<=	Less than or equal to
>=	Greater than or equal to

2.3 DELIMITER WORDS

Certain letter combinations are reserved as part of the structure of the language and may not be used as identifiers unless the compiler option to accept delimiter words in single quotes is in use. Such an option is selected by using a special switch option (refer to Chapter

18). The standard method of delimiter word representation, that is, reserved words, is assumed throughout this manual. For example, the delimiter word

```
BEGIN
```

will always appear in the text of this manual as shown above and cannot be used as an identifier in a program. If the alternative method of representation is used, it would appear as

```
'BEGIN'
```

and

```
BEGIN
```

could be used as an identifier. Table 2-3 contains a list of all the delimiter words used in the language.

Table 2-3
Delimiter Words Used in DECsystem-10/20 ALGOL

Reserved Word	Chapter Reference
AND	5.2.1
ARRAY	9
BEGIN	10
BOOLEAN	5.2
CHECKOFF	18
CHECKON	18
COMMENT	2.4
DIV	5.1
DO	8
ELSE	7.3
END	10
EQV	5.2.1
EXTERNAL	11.9
FALSE	4.2
FOR	8
FORWARD	11.8
GO	7.2
GOTO	7.2
IF	7.3
IMP	5.2.1
INTEGER	3.2
LABEL	11
LINE	18
LISTOFF	18
LISTON	18
LONG	3.2
NOT	5.2.1
OR	5.2.1
OWN	15
PROCEDURE	11
REAL	3.2
REM	5.1
STEP	8
STRING	13
SWITCH	12
THEN	7.3

Table 2-3 (Cont.)
Delimiter Words Used in DECsystem-10/20 ALGOL

Reserved Word	Chapter Reference
TRUE	4.2
UNTIL	8
VALUE	11
WHILE	8

2.4 USE OF SPACING AND COMMENTARY

The readability of ALGOL programs can be enhanced by the judicious use of spacing, tab formatting, and commentary. Spaces, tabs, and form feeds (page throws) may be used freely in a source program subject to the following constraints:

1. Spaces, tabs, line feed, or form feed characters may not appear within delimiter words.

2. Where two delimiter words are adjacent, or where an identifier follows a delimiter word, these must be separated by one or more spaces and/or tabs.

3. Spaces, tabs etc., are significant within string constants.

Comments are introduced by either the word COMMENT or the symbol ! (available in DECsystem-10/20 ALGOL, but not necessarily in other implementations of ALGOL). Such a comment may appear anywhere in a program; the comment text is terminated by a semicolon. Refer to Section 11.10 for additional means to add comments to a program.

CHAPTER 3

IDENTIFIERS AND DECLARATIONS

3.1 IDENTIFIERS

An identifier must begin with an upper-case letter and optionally be followed by one or more upper-case letters and/or decimal digits. An identifier may not contain more than 64 characters.

NOTE

1. Unlike FORTRAN, there is no implied type attached to an identifier.

2. All identifiers in a program (except labels) have to be "declared", that is, the use to which identifiers are to be put must be specified, prior to the actual usage.

Examples:

The following are identifiers:

I

ALPHA

P43

J4K5

HOUSEHOLDERTRIDIAGONALIZATION

The following are not identifiers:

4P	does not begin with letter
BOOLEAN	unless the non-reserved word delimiter representation is used
ONCE AGAIN	space not allowed

DECsystem-10/20 ALGOL also permits the use of a decimal point as a "readability symbol" in the alphabetic portion of identifiers. These readability symbols can appear between two alphabetic characters of an identifier and are ignored by the compiler. Thus:

ONCE.AGAIN

and

```
PI.BY.TWO
```

have exactly the same effect as

```
ONCEAGAIN
```

and

```
PIBYTWO
```

respectively.

Note that

```
ALPHA3.5
```

and

```
BETA.22
```

are not identifiers, since the decimal point does not appear between two alphabetic characters.

3.2 SCALAR DECLARATIONS

A declaration reserves an identifier to represent a particular quantity used in a program. Such declarations are mandatory in ALGOL. At any particular point during program execution, the form of the variable or quantity associated with the identifier depends on the type of variable. The type of variable is determined by the type of identifier which represents it.

There are five types of scalar variables, that is, variables which contain a single value:

```
Integer
Real
Long Real
Boolean
String
```

Integer, real, and long real variables are capable of holding numerical values of the appropriate type (and only of that type). The range of values is as follows: integer: -34,359,738,368 through 34,359,738,367; real and long real: approximately -1.7&38 through 1.7&38; values less than approximately 1.4&-39 in magnitude are represented by zero.

Boolean variables (similar to FORTRAN's Logical variables) can hold a Boolean quantity, which is usually one of the states TRUE or FALSE but, in general, can be any pattern of 36 bits.

String variables are somewhat more complicated. The user is referred to Chapter 13 for a full description of the subject.

All of the above variables can be declared for use by preceding a list of the identifiers to be used by the appropriate delimiter word for their type. Throughout this manual, a "list of items" consists of those items arranged sequentially and separated by commas.

Examples:

```
INTEGER I,J,K;

LONG REAL DOUBLE,P,Q,ELEPHANT;

BOOLEAN ISITREALLYTRUE;

STRING S,T;
```

CHAPTER 4

CONSTANTS

4.1 NUMERIC CONSTANTS

There are three forms of numeric constants:

1. Integer constants
2. Real constants
3. Long Real constants

4.1.1 Integer Constants

Integer constants consist of a number of adjacent decimal digits, subject to the constraint that the number represented must be in the range 0 through 34,359,738,367.

NOTE

Any preceding sign that appears in the program is not considered part of the constant.

Examples:

3

24

9276541

See also Section 4.3

4.1.2 Real Constants

Real constants consist of a decimal number (containing either an integral part or a fractional part, or both) followed by an optional exponent. If the value of a decimal number is unity, then this may be omitted, and the real constant solely represented by the exponent (see the last example in this section). The exponent consists of either the & or @ symbol followed by an optionally signed integer. This has the effect of multiplying the decimal number by the power of ten specified in the exponent. If no decimal number appears, a value of unity is assumed.

The range of real constants is approximately 1.4&-39 to 1.7&38; numbers less than 1.4&-39 are represented by zero. Real numbers are stored to a significance of approximately eight and one-half decimal digits.

Examples:

Representation	Value
3.141592653589793	3.14159265
.0001	0.0001
4.37&5	437000.0
5&-3	0.005
&-6	0.000001

4.1.3 Long Real Constants

Long real constants are used to represent numeric quantities to approximately twice the precision available with real numbers: about seventeen decimal digits. Long real constants are formed by writing a real constant in floating-point form, but replacing the & or @ by && or @@. The range of long real constants is the same as that of real constants, except numbers below approximately 3.0&&-30 can only be represented to single precision due to hardware considerations.

Examples:

Representation	Value
3.14159265358979323846&&0	3.1415926535897932
12&&-3	0.012

4.1.3.1 Automatic Conversion of Constants to Long Real Constants - The compiler keeps all real constants internally to the precision available with long real variables and determines the precision required by the context in which the constant is used. This is to avoid loss of precision in arithmetic expressions involving long real variables and real constants, thus making easier the conversion of programs to use long real variables. As a result of this, a long real constant only needs to be specified explicitly if it is desired to force a calculation to be done to long real precision irrespectively.

For example, if LR is a long real variable and X is a real variable, the following two assignments would produce (slightly) different results:

1. LR:=0.1+X

   would perform the addition, then convert the result to long real and assign this to LR.

2. LR:=0.1@@0+X

   would take the value of X and convert to long real, then perform the addition to long real precision and assign the result to LR. If X was a long real variable, however, there would be no difference, as the addition would be performed to long real precision in both cases.

4.2 OCTAL AND BOOLEAN CONSTANTS

Octal constants consist of the symbol % followed by a number of octal digits. Up to twelve significant digits may appear (leading zeros are ignored); these digits are right justified.

Examples:

```
%777777777774

%0470
```

Octal constants may only be used in Boolean expressions.

Boolean constants consist of the words TRUE and FALSE. They are equivalent to the octal constants %777777777777 and %000000000000, respectively.

4.3 ASCII CONSTANTS

Up to five ASCII symbols can be packed right justified to give an integer-type constant. The format is a dollar sign ($), followed by up to five ASCII symbols enclosed within a delimiting symbol pair. The leading delimiter symbol immediately follows the $, and may be a readable character or an invisible one such as a space. Thus, the user can generate a single ASCII character constant by placing one space on each side of it, and preceding the triplet by a dollar sign.

Examples:

Text	Octal Value
$ A	000000 000101
$/01234/	160713 516674

4.4 STRING CONSTANTS

String constants allow the user to store any reasonable length string of ASCII characters within a program. The length of such a constant is restricted only by the amount of core storage available to the user for the execution of the program. String constants may be used, typically, to output a message during the execution of the program or as values assigned to string variables.

The string of symbols is enclosed within quotes ("). There are restrictions on the symbols that may appear within the string.

1. [ ] ; and " may not appear alone.

2. ; [and] may appear if they are properly paired.

3. Single occurrences of [ ] ; and " are represented by [[ ]] ;; and "" respectively.

4. Where a string has to be broken across two or more lines of source, the carriage return and line feed characters can be ignored by preceding them with a control-back arrow character.

NOTE

[[and]] are stored as such in the byte string generated by the compiler. ;; and "" are stored as a single ; or ", respectively. The restriction on the representation of ; is made to protect the user against quoting the whole program by missing out a ".

Square brackets are used to enclose symbols that have a specific effect when the string is output. These are discussed in Paragraph 16.6.2.

Examples:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

"REMEMBER THAT SPACES ETC.  ARE SIGNIFICANT"

"[P5C]INPUT DATA:[5C]"

''''''A[[I]] := 0.1;;''''''
```

CHAPTER 5

EXPRESSIONS

5.1 ARITHMETIC EXPRESSIONS

DECsystem-10/20 ALGOL arithmetic expressions are written in a form similar to that used in FORTRAN and many other high-level scientific computer languages. The usual algebraic rules concerning precedence of operators and brackets are followed (see Table 5-1).

Table 5-1
Operator Precedence

Operator	Priority (decreasing)
parentheses	1
exponentiation	2
multiplication and division	3
addition and subtraction	4

There are two additional operators, DIV and REM, that indicate integer division and remainder, respectively, and these have the same precedence as ordinary division. Within the precedence scheme, the order of evaluation is always from left to right. For example:

```
X ^ Y ^ Z means (X ^ Y) ^ Z
```

and

```
I DIV J REM K means (I DIV J) REM K
```

Unlike FORTRAN, when ordinary division of one integer by another is performed, the real result is not rounded to an integer value.

The difference between the various types of division is clarified by the following examples:

```
7/4 yields a result of 1.75, whereas

7 DIV 4 yields a result of 1, and

7 REM 4 yields a result of 3
```

The interpretation of integer division for negative integers follows:

Let M, N>0, then

```
-M DIV N = M DIV (-N) = -(M DIV N)

-M DIV (-N) = M DIV N
```

The integer remainder operator, REM, is defined so that for all integral M, N:

```
M REM N = M - N*(M DIV N)
```

5.1.1 Identifiers And Constants

Arithmetic expressions consist of operands, that is, identifiers and constants, of the three types, integer, real and long real, together with the arithmetic operands + - * / DIV REM and ↑ and parentheses where necessary.

Since automatic conversion takes place as necessary when an expression is evaluated, the user may freely mix the three different types of identifiers and constants, (refer to section 4.1.3.1 for the effect of mixing real or long real constants and variables).

Integer quantities may have more precision than can be represented in a real variable. The user must beware of possible loss of significance in integral quantities used in mixed type expressions.

5.1.2 Special Functions

Three special functions are provided for use in arithmetic expressions. The first is the transfer function, ENTIER, which converts a real or long real quantity into an integer quantity defined as the largest integer value not exceeding the argument.

Thus

```
ENTIER(3.5) = 3
```

and

```
ENTIER(-3.5) = -4
```

The special function ABS yields the absolute value (also known as the modulus) of its argument. The argument may be any integer, real, or long real quantity; the result is always of the same type as the argument.

Thus

```
ABS(-3.5) = 3.5
```

and

```
ABS(-3) = 3
```

The special function SIGN is the signum function whose argument can be integer, real, or long real. The result is always integral, being minus one or zero or plus one, depending on whether the argument is negative, zero, or greater than zero, respectively.

Thus

```
SIGN(-3.5) = -1

SIGN(0) = 0

SIGN(3.5) = 1
```

NOTE

ENTIER, ABS, and SIGN are not delimiter words. They may be used for other purposes in a program.

Examples of simple arithmetic expressions follow:

```
X

I + 3

X*Y/Z

P+Q/R

X2 + Y

XJ-4

J + ENTIER(K-2)

SIGN(ENTIER(J/K) + 1)

(X + Y) * (-I)
```

5.2 BOOLEAN EXPRESSIONS

Boolean expressions involve Boolean identifiers, Boolean and octal constants, arithmetic conditions, and Boolean operators interspersed in an order similar to that of arithmetic expressions.

5.2.1 Boolean Operators

There are five Boolean operators listed here in decreasing order of precedence.

1. NOT (unary operator)
2. AND
3. OR
4. IMP (implication)
5. EQV (equivalence)

NOT is a unary operator that complements a Boolean quantity in the same way that a unary minus sign negates an arithmetic quantity in an arithmetic expression. In this case, FALSE is changed to TRUE, or vice versa.

Table 5-2 gives the result of A OP B where OP stands for one of the Boolean operators AND, OR, IMP, or EQV, for all values of A and B.

Table 5-2
Function of Boolean Operators

A	FALSE		TRUE	
B	FALSE	TRUE	FALSE	TRUE
A AND B	FALSE	FALSE	FALSE	TRUE
A OR B	FALSE	TRUE	TRUE	TRUE
A IMP B	TRUE	TRUE	FALSE	TRUE
A EQV B	TRUE	FALSE	FALSE	TRUE

In addition, the following theorems hold true:

A IMP B is equivalent to NOT A OR B,

A EQV B is equivalent to A AND B OR NOT A AND NOT B.

5.2.2 Evaluation Of Boolean Variables

Actually, Boolean variables may have a value consisting of any pattern of bits, rather than be confined to the values TRUE and FALSE. The logical operations operate on a bit-by-bit basis according to the preceding rules.

The actual test employed to determine the truth of a Boolean expression such as

```
B AND C
```

is to evaluate it and regard it as true if the value is nonzero, that is, at least one bit is set, otherwise regard as false.

This is particularly important when octal constants are used in Boolean expressions. For example, if the user wishes to test a particular bit in a Boolean variable, an appropriate octal constant can be used, for example:

```
B AND %1
```

is a Boolean expression that is true if and only if the bottom (least significant) bit of B is a one.

5.2.3 Arithmetic Conditions

Arithmetic conditions are used as operands in Boolean expressions. They consist of two arithmetic expressions coupled with a comparator. The comparator, which decides the particular type of test to be performed on the two expressions, is one of the following:

< less than

<= less than or equal to

= equals

> greater than

>= greater than or equal to

not equal to

Such an arithmetic condition can be regarded as true or false according to whether the condition specified by the comparator is met when the arithmetic expressions on each side are evaluated. The resulting condition may form part of a Boolean expression.

The following examples of Boolean expressions, shown in Table 5-3, also involve arithmetic conditions.

Table 5-3
Boolean Expressions

Expression	Meaning
NOT B	NOT B
B AND NOT C	B AND (NOT C)
A OR B AND C	A OR (B AND C)
B EQV X<Y	B EQV (X<Y)
X+Y<Z AND B OR P=Q	(((X+Y)<Z) AND B) OR (P=Q)

5.3 INTEGER AND BOOLEAN CONVERSIONS

an integer quantity can be converted to a Boolean quantity by means of the dummy function BOOL. Similarly, the dummy function INT converts a Boolean quantity to an integer quantity.

The value passed by these functions is unchanged: the functions are included for semantic correctness. Thus:

```
BOOL(I)
```

may be regarded as a Boolean operand, and

```
INT(B)

INT(%400000000000)
```

as integer operands.

BOOL and INT are not reserved words and can be used for other purposes by "declaring" them as required. However, this practice should be avoided to prevent confusion.

CHAPTER 6

STATEMENTS AND ASSIGNMENTS

6.1 STATEMENTS

The statement is the basic operational unit in ALGOL-60 and describes an operation, such as an assignment, to be performed at run time.

6.2 ASSIGNMENTS

Assignments convey the value produced by the execution of an expression to a destination variable of the appropriate type. This is done by writing the destination identifier, followed first by the symbols : and = and then by the expression to be evaluated. Thus

```
X := Y + Z
```

causes the result of the addition of the values contained in the variables Y and Z to be placed in the variable X.

When an assignment is made to a variable type differing from that of the result of the expression, a type conversion is performed. Integer, real and long real expressions may be assigned to variables of any of these three types, but not to any other types. Boolean and string expressions can only be assigned to a variable of the same type.

If a real or long real value is assigned to an integer type variable, a rounding process occurs.

```
I := X
```

results in an integral value equal to

```
ENTIER(X + 0.5)
```

being assigned to I.

When an integer expression is assigned to a real or long real variable, a conversion to that type is performed. Real to long real conversion simply consists of zeroing the low-order precision word of the long real result after assignment of the real result to the high-order part of the long real variable. Long real to real assignments truncate the low-order part of the long real expression, after appropriate rounding.

6.3 MULTIPLE ASSIGNMENTS

A value may be assigned simultaneously to several variables of the same type by a multiple assignment. This takes a form such as

```
P := R := S := X + Y - Z ;
```

where the result of adding Y to X and subtracting Z is assigned to P, R, and S simultaneously.

All identifiers on the left-hand side of a multiple assignment must be of the same type. If the user wishes to assign a value to two or more different types of variables, the "assignment within expression" (embedded assignment) feature must be used, as below.

A parenthesized assignment may be substituted for any operand in an expression. For example,

```
X := (Y := P+Q)/Z;
```

This causes the embedded assignment to be made after the inner expression P+Q is evaluated. Where a type conversion is performed as part of an embedded assignment, the operand type is the same as that assigned to the variable in the embedded assignment. Thus

```
X := (I := 3.4) ;
```

sets I equal to 3 and X equal to 3.0.

6.4 EVALUATION OF EXPRESSIONS

All expressions in DECsystem-10/20 ALGOL are evaluated observing the normal algebraic rules of precedence, including bracketing.

Within the precedence structure, expressions are always evaluated from left to right. For example, if X is a scalar, and F a function procedure (see Chapter 11) that alters X,

```
X := X+F ;
```

may have a different effect than

```
X := F+X ;
```

This is known as a "side effect".

Consider also:

```
A[I] := (I := I+1) ;
```

The subscript I is always evaluated before I is incremented, as it is to the left of the embedded assignment, within the statement. Thus the above expression is equivalent to

J := I; I := I+1; A[J] := I ; The user can always predict the order of evaluation of an expression and can count on such things as

```
X := (P := P+Q)/(P+R) ;
```

being evaluated correctly, thus giving the same result as

```
P := P+Q ;

X := P/(P+R);
```

6.5 COMPOUND STATEMENTS

A compound statement consists of a number of statements, preceded by BEGIN, separated by semicolons, and terminated by END. ALGOL statements, unlike those in FORTRAN, are terminated by a semicolon not by the end of a line of text.

For example:

```
BEGIN

          I := 3;  J := 4;

          K := I + J;

          X := K

END
```

is a compound statement. Semicolons do not have to appear after the BEGIN or before the END; BEGIN and END act as a type of bracket.

The usefulness of compound statements will become apparent in later chapters.

CHAPTER 7

CONTROL TRANSFERS, LABELS, AND CONDITIONAL STATEMENTS

7.1 LABELS

A label is a method of marking a place in a program so that control can be transferred to that point from elsewhere in the program.

DECsystem-10/20 ALGOL uses identifiers as labels. These identifiers are placed before statements and are followed by a colon. Numeric labels are permitted in the Revised Report, but are not implemented in DECsystem-10/20 ALGOL. Most implementations of ALGOL-60 do not allow integer labels.

For example:

```
COMP:  X := X + Y ;
```

is a statement labeled by COMP.

More than one label can be attached to a statement if required; thus,

```
LAB1:  LAB2:  Y := 0 ;
```

7.2 UNCONDITIONAL CONTROL TRANSFERS

A transfer of control, or "jump", to a statement in a program is effected by a GOTO statement. This statement consists of the word GOTO followed by the name of the label attached to the relevant statement. The two words GO TO can be used instead of the word GOTO in any statement where GOTO can be used. Thus:

```
BEGIN      INTEGER I,J,K;

LAB:       I := J := 3;

           K := I + J;

           GOTO LAB

END
```

is an example of a somewhat tedious program. Clearly, to write any reasonable program, it is necessary to be able to jump conditionally.

7.3 CONDITIONAL STATEMENT

Conditional statements provide a method to make the execution of either a statement or a compound statement dependent on some condition in the program, such as the value of a variable. The simplest form of a conditional statement is

```
IF B THEN S
```

where B is some Boolean expression, and S is a statement. For example:

```
IF X < 0 THEN I := I + 1
```

Here, X < 0 is the Boolean expression and I := I + 1 is the statement which is obeyed if and only if the Boolean condition is true, that is, if X is negative.

A more general form of a conditional statement is

```
IF B THEN S1 ELSE S2
```

In this case, the statement S1 is obeyed if and only if the Boolean expression B is true, and S2 is obeyed if and only if it is false. In order to eliminate the "dangling ELSE ambiguity" (a construction in which an ELSE could be paired with either of two THENs), S1 must not be conditional, FOR, or WHILE statement which ends in an ELSE clause. (Refer to Chapter 14 for more complete information.)

A control transfer, a type of statement, can appear in a conditional statement. Thus:

```
BEGIN     INTEGER I;

          I := 0;

LAB:      I := I + 1;

          IF I < 100 THEN GOTO LAB

END
```

is a simple way of counting to one hundred. More sophisticated methods are shown in Chapter 14.

CHAPTER 8

FOR AND WHILE STATEMENTS

8.1 FOR STATEMENTS

The for statement enables the user to iterate a portion of the program in a fashion similar, to but more sophisticated than, FORTRAN's DO loop.

The general format is

```
FOR V := FORLIST DO S
```

where V is a variable and S is a statement (compound or otherwise).

FORLIST can consist of any number of FOR elements (separated by commas). A FOR element takes one of the following forms:

1. An expression:

   ```
   E
   ```

2. A STEP-UNTIL element taking the form:

   ```
   E1 STEP E2 UNTIL E3
   ```

3. A WHILE element taking the form:

   ```
   E WHILE B
   ```

   where B is some Boolean expression.

Any number of FOR elements may appear in a FOR statement and executed serially. Consider the following examples:

```
FOR I := 3,5,10 DO .....

FOR X := 2.5,5.0,10.0 DO .....

FOR J := 1,2,5 STEP 5 UNTIL 20 DO .....
```

8.1.1 STEP-UNTIL Element

This particular form deserves closer inspection. Consider

```
FOR I := 1 STEP I UNTIL N DO S
```

The statement S is obeyed with I taking an initial value of 1, and being incremented by I until the final value N is achieved. The question is , "Is the I after the STEP recalculated during each turn

around the loop, or does it have a constant value equal to the initial value of I?"

The answer is slightly more complicated. Consider the general case

```
FOR V := E1 STEP E2 UNTIL E3 DO S
```

This is defined to have exactly the same effect as

```
          V := E1;
L1:       IF (V - E3)*SIGN(E2) > 0 THEN GOTO L2;
          S;
          V := V + E2;
          GOTO L1;
L2:
```

Clearly, the value of I following the STEP in the previous example is evaluated, if necessary, twice during each turn around the loop, once in the sign test at L1, and again to update V. ALGOL allows the user to modify V, E1, E2, and E3 freely throughout the loop, and takes account of all these changes in the evaluation of the loop.

NOTE

DECsystem-10/20 ALGOL allows the user the abbreviated form

```
FOR V := E1 UNTIL E3 DO S
```

instead of

```
FOR V := E1 STEP 1 UNTIL E3 DO S
```

8.1.2 WHILE Element

A FOR statement with a single WHILE element takes the form

```
FOR V := E WHILE B DO S
```

This is interpreted as follows:

```
L1:       V := E;
          IF NOT B THEN GOTO L2;
          S;
          GOTO L1;
L2:
```

Once again, the complexity of the loop may be affected by changing V and E within the loop.

8.2 WHILE STATEMENT

The WHILE statement is an enhancement of ALGOL-60 provided in DECsystem-10/20 ALGOL. The general form of the statement is

```
WHILE B DO S
```

and is interpreted as follows:

```
L1:     IF NOT B THEN GOTO L2:

        S;

        GOTO L1;

L2:
```

8.3 GENERAL NOTES

1. Within a FOR statement of any kind, the user can change the controlling variable or any other variable appearing within the action of the loop. Such changes predictably affect the execution of the loop by the rules given above.

2. On exit from a FOR statement either by jumping out of the loop or by exhausting the FOR elements, the controlling variable has a well-defined value equal to the last assigned value of the controlling variable. This may not be true of other ALGOL-60 implementations. Section 4.6.4 of the Revised Report should be studied carefully in this connection.

CHAPTER 9

ARRAYS

9.1 GENERAL

Arrays are essentially collections of variables of the same type, allowing the user to address each variable individually by means of a common name and a unique subscript or subscripts. In the simplest case, an array is a vector and is known as a one-dimensional array. A matrix is a two-dimensional array, etc.

There is no limit to the number of subscripts allowed, other than those imposed by the ability of the computer to store the array.

9.2 ARRAY DECLARATIONS

Arrays may be of type integer, real, long real, Boolean, or string and these are declared similarly to scalar variables, except the size of the array must be stated. For each subscript that the array possesses, a lower and an upper bound, called the "bound pair" for that subscript, must be given.

For example, to declare two one-dimensional integer arrays A and B with lower bound 1 and upper bound 5:

```
INTEGER ARRAY A,B[1:5]
```

NOTE

> The lower and upper bounds must be enclosed in square brackets and separated by a colon.

When there are two or more subscripts, the declaration is similar, and the bound pairs are separated by commas. Thus

```
LONG REAL ARRAY P,Q,R[-5:2,0:10]
```

declares three long real arrays, P, Q and R, with the first subscript bounded by -5 and 2 and the second subscript bounded by 0 and 10.

Arrays of the same type but of different sizes may be declared in the same statement.

```
REAL ARRAY A[1:10], B,C[1:10,1:12]
```

NOTE

In the case of real arrays, the REAL may be omitted in the declaration, and is assumed by default, thus:

```
ARRAY A[1:10], B,C[1:10,1:12]
```

The bounds in an array need not be static, as in the examples above, but may be any arithmetic expressions, which are evaluated to give an integral value for the individual bound pairs. The use of such dynamic array declarations will become apparent later. No bound may exceed 131,072 in magnitude.

9.3 ARRAY ELEMENTS

An individual element of an array can be referred to by following the name of the array by a list of subscripts in square brackets. The number of subscripts must be identical to the number in the array declaration. Thus, a typical element of A used in the last declaration might be

```
A[5] or A[9] or generally, A[I]
```

where I is some integer expression or, in general, any expression whatsoever, with the limitation that its value when used as a subscript and evaluated as an integer is in the range 1 through 10, the bounds of the array A.

As an example of the use of arrays, consider the declaration

```
REAL ARRAY D,E,F [1:10,1:10]
```

and suppose the operation required, was to set F equal to the matrix product of D and E:

```
FOR I := 1 UNTIL 10 DO

FOR J := 1 UNTIL 10 DO

BEGIN     X := 0;

          FOR K := 1 UNTIL 10 DO X := X + D[I,K]*E[K,J];

          F[I,J] := X

END
```

NOTE

1. In the above example X is used to accumulate the inner product of the multiplication for all values of I and J. The variable X was used instead of F to facilitate the computation.

2. An element of an array of a particular type may be used anywhere that a scalar variable of the same type may be used, even in such places as the controlling variable in a FOR statement.

CHAPTER 10

BLOCK STRUCTURE

10.1 GENERAL

ALGOL program structure is somewhat more complicated than other high-level languages, such as FORTRAN. An ALGOL program consists of a number of "blocks" arranged hierarchically. A block consists of the words BEGIN and END enclosing the declarations and (optionally) statements.

Thus:

```
BEGIN

    BEGIN

    END

    BEGIN

        BEGIN

        END

    END

END
```

is an ALGOL program, assuming appropriate declarations and statements in the blocks.

The block structure offers the user many interesting features not available in non-block structured languages. For instance, the user may declare an identifier that appears to conflict with another identifier in an enclosing block. Thus:

```
BEGIN   INTEGER I;

        BEGIN INTEGER I;

        END

END
```

In fact, there is no conflict as there are two different Is. The only I that statements in the outer block can "see", is the one in the outer block. Similarly, any statement in the inner block will always use the I in that block. Such a declaration in an inner block is known as a "local" variable and takes precedence over declarations occurring at an outer or more "global" level. In general, all variables can be "seen" from any point in a program that is either in

the same block as the declaration or in a block that is enclosed by the block in which the declaration of the variable occurred. Note that a more local variable is always taken in preference to a relatively global variable. Consider the following example:

```
BEGIN     INTEGER I,J;

          [1]

          BEGIN     INTEGER J,K

                    [2]

          END;

          BEGIN     INTEGER I,K

                    [3]

          END

END
```

Any statements occurring at point [1] can see the declarations of I and J, which are local, but cannot see the declarations of J and K in the first inner block, or the declarations of I and K in the second inner block. At [2], the local variables J and K can be seen, as can the global variable I in the outer block. The global variable J is not seen because the local variable J takes precedence over it; the variables I and K in the second inner block are not seen at all. A similar situation occurs at [3]; here both local variables I and K, as well as the global variable J, are seen.

Note that the "scope" of a variable is the set of all places in a program where it can be seen and therefore used. This term will be used frequently throughout this text.

In general, local variables are more efficient to use than global ones. This statement is also true of most ALGOL-60 implementations. Where a global variable is used frequently, a local variable should be assigned as having the same value and used instead. For example:

```
BEGIN     INTEGER I;

          .....

          I := .....

          BEGIN     INTEGER II;

                    II := I;

                    .....

                    ..... II .....

          END

          .....

END
```

Here, in the inner block, a local variable II is used, and assigned the value of the global variable I for use throughout the local block.

10.2 ARRAYS WITH DYNAMIC BOUNDS

The concept of the scope of a variable can be applied most usefully to arrays. In DECsystem-10/20 ALGOL, all arrays are constructed at execution time (that is, no fixed space is reserved during compile-time), irrespective of whether their bounds are static or dynamic. When a declaration of an array is encountered within a block, the space required to construct it is obtained and the array is laid out. When the end of the block enclosing the array is reached, that is, the array variable is no longer within scope, the space utilized by the array is recovered and can be used later for other arrays.

Consider the case of a problem in which the size of an array to be used in a calculation is dependent on the data to be processed. The programmer has the choice of making the array large enough to cope with the worst case, or constructing the array with dynamic bounds to suit the size required by the particular data. The first method has the disadvantage of wasting space on many occasions. The latter method only has the minor disadvantage of the overhead needed to construct the array. Such overhead is very small compared to the running time of most programs, therefore, the second method is more desirable.

Consider the following example:

```
BEGIN     INTEGER N;

L:        N := .....

          .....

          BEGIN ARRAY A[1:N,1:N];

          .....

          END;

          GOTO L

END
```

A value for N is calculated in this example, possibly dependent on some data read into the program, and used to declare the array A, which is used to process the data in the inner block. When the end of the inner block is reached, the space used by A is recovered and control passes to L, where another value for N is calculated, and the process repeated.

CHAPTER 11

PROCEDURES

Procedures are similar in concept to the FORTRAN subroutine, but with more sophisticated and general applications.

A "procedure" is a portion of an ALGOL program that is given a name for identification and can be "called" from any part of a program which is in the scope of the body of the procedure. A procedure can execute a number of statements, and in the case of function procedures can return a value to the procedure body. In addition, it may or may not have parameters.

In DECsystem-10/20 ALGOL, a procedure can be one of the following types: integer, real, long real, Boolean, string or typeless. The formal parameters of a procedure (know as "dummy variables" in FORTRAN), can be one of the following types: integer, real, long real, Boolean or string, as scalars, arrays or procedures, or label. There are eighteen different types of parameters. In addition, all of these parameters may appear in two different modes and strings, neither of which is the same as FORTRAN's method of handling parameters.

11.0.1 PARAMETERS CALLED BY "VALUE"

Calling parameters by "value" is the most common and, with the exception of arrays and strings, the most efficient way to pass a parameter to a procedure. The value of the expression presented in a procedure call, known as the actual parameter, is evaluated on entry to the procedure and assigned to a formal parameter within the procedure. This formal parameter acts as a local variable in the procedure which is initialized, the initial value being that of the actual parameter supplied in the call to the procedure.

Since, in the case of arrays or strings, a new copy of the array or string is made, this type of parameter-passing for arrays and strings should be avoided unless specifically required.

11.1 PARAMETERS CALLED BY "NAME"

Calling parameters by "name" is a useful method of passing a parameter to an ALGOL procedure. Whenever the formal parameter associated with the actual parameter in a procedure body appears in the body of the procedure, the actual parameter is re-evaluated as if it appeared in the procedure body at that point. For example, if the actual parameter were an array element such as

```
A[I]
```

the element would be re-evaluated using the value of I available each time the formal parameter is used, not the value of I at the time the procedure body is entered.

Table 11-1 shows the different types of formal parameters, with valid actual parameters that can be substituted in a procedure call.

Table 11-1
Parameter in a Procedure Call

Formal Parameter Type	Permissible Actual Parameter
Integer Real Long Real	Any arithmetic expression
Boolean	Any Boolean expression
String	Any string expression (refer to Chapter 13)
Label	A label or switch element (refer to Chapter 12 and Paragraph 14.4)
Switch	A switch
Integer Array	An array of type integer*
Real Array (or Array)	An array of type real*
Long Real Array	An array of type long real*
Boolean Array	An array of type Boolean
String Array	An array of type string
Procedure	A non-type procedure
Integer Procedure Real Procedure Long Real Procedure	A procedure of type integer, real, or long real
Boolean Procedure	A procedure of type Boolean
String Procedure	A procedure of type string

*In the case where the array parameter is called by value, any arithmetic type (integer, real or long real) array is allowed as an actual parameter. A type conversion takes place during the copying process.

11.2 PROCEDURE HEADINGS

Procedure headings identify the type of procedure, the number and the type of parameters.

A procedure heading consists of:

1. The type of procedure (omitted in the case of typeless procedures).
2. The word PROCEDURE followed by the name of the procedure.
3. A semicolon if the procedure has no parameters, otherwise
4. A list of the formal parameters, enclosed in parentheses, and followed by a semicolon.
5. Specifications of the formal parameters.

   Omitting formal parameter specifications, this looks like

```
LONG REAL PROCEDURE LR;

BOOLEAN PROCEDURE BOOLEAN (I,J,K);

PROCEDURE CALC(THETA,X);
```

The formal parameter specification that follows consists of a list of descriptions of the formal parameters, appearing in any order, and a value specification if any of the parameters are to be called by value. (If this is omitted, the parameters, by default, will be called by name.) For example, the specification of the formal parameters for the second example above might be:

```
VALUE I,J;  INTEGER I,J,K;
```

meaning that all three formal parameters are of type integer (scalars), and I and J are to be called by value, while K is to be called by name. A typical formal parameter specification for the third example might be:

```
REAL PROCEDURE THETA;  ARRAY X;
```

NOTE

Procedure headings must precede the body of the procedure.

11.3 PROCEDURE BODIES

The body of a procedure is that part which follows the procedure heading, and consists of a single statement, a compound statement, or a block. In the last-mentioned case, there may be declarations of local variables within the block, and also other blocks or procedures. Consider the following examples of realistic procedures:

1. A real procedure, SQUAREROOT, to calculate the square root of a real quantity. The first parameter is the quantity, the second is a label that is used as an escape if the quantity is found to be negative. The result of the procedure is the square root of the quantity. Note how the result of the calculation is assigned to the procedure by placing the name of the procedure on the left-hand side of an assignment.

```
REAL PROCEDURE SQUAREROOT(X,L);
        VALUE X;  REAL X;  LABEL L;
BEGIN   REAL Y,Z;
        IF X < 0 THEN GOTO L;
        Y := (1 + X)/2;
IT:     Z := (X/Y + Y)/2;
        IF ABS(Z - Y) < 1&-6 THEN GOTO OK;
        Y := Z;  GOTO IT;
OK:     SQUAREROOT := Z
END
```

The previous example uses the Newton-Rapheson method of finding the square root of a number by taking an initial approximation (1 + X)/2 and iterating until the difference between successive approximations is less then 1&-6. The procedure is again described below, with the aid of some commentary. The DECsystem-10/20 ALGOL alternative method of commentary (refer to Chapter 2) is used for brevity:

```
REAL PROCEDURE SQUAREROOT(X,L);
  VALUE X;  REAL X;  LABEL L;
BEGIN ! CALCULATES THE VALUE OF SQRT(X)
      USING THE NEWTON-RAPHESON METHOD.
      L IS USED FOR AN ESCAPE IF X < 0;
  REAL Y,Z;
  IF X < 0 THEN GOTO L; ! EXIT IF X < 0;
  Y := (1+X)/2;         ! FIRST APPROXIMATION;
IT:
  Z := (X/Y + Y)/2;     ! ITERATE;
  IF ABS(Z-Y) < 1&-6
      THEN GOTO OK;     ! TEST FOR CONVERGENCE;
  Y := Z;  GOTO IT;     ! OTHERWISE CONTINUE;
OK:
  SQUAREROOT := Z;      ! FINAL RESULT;
END
```

2. This function evaluates the sum of the values of any real procedure G over the integers 1 ..... N (that is, iterating N times) where N is also a parameter of the procedure.

```
REAL PROCEDURE SUM(G,N);
        VALUE N;  REAL PROCEDURE G;  INTEGER N;
BEGIN   INTEGER I;  REAL X;
        X := 0;
        FOR I := 1 UNTIL N DO X := X + G(N);
        SUM := X
END
```

NOTE

In this example, the formal parameter G is invoked so that the actual procedure substituted for G is called.

11.4 PROCEDURE CALLS

In the preceding example, the procedure G was "called". Since G is a function procedure, it is only necessary for its name to appear in an expression for the procedure to be entered with the actual parameters specified substituted for the formal parameters.

The procedure SQUAREROOT can be called in a similar way, for example:

```
P := SQUAREROOT(Z + 0.5,ERR)
```

causes the square root of Z + 0.5 to be calculated.

An example of the use of the procedure SUM can be used to calculate the sums of the square roots of the first J integers, with the result squared, as follows:

```
X := SUM(SQUAREROOT,J)^2;
```

Here is a further example of a procedure and the calls:

```
PROCEDURE MATRIXMULT(A,B,C,N);
VALUE N;  ARRAY A,B,C ;  INTEGER N;
BEGIN   INTEGER I,J,K;  REAL X;
        COMMENT THIS PROCEDURE PERFORMS THE MATRIX
        MULTIPLICATION OF B AND C AND STORES THE RESULT
        IN A.  THE ARRAYS ARE ASSUMED TO BE SQUARE
        AND OF BOUNDS 1:N,1:N;
        FOR I := 1 UNTIL N DO
        FOR J := 1 UNTIL N DO
        BEGIN X := 0;
```

```
            FOR K := 1 UNTIL N DO X := X +
                    B[I,K]*C[K,J];
                    A[I,J] := X
            END
    END
```

A typical call for this procedure might be

```
    MATRIXMULT(E,F,G,N);
```

or

```
    MATRIXMULT(E,F,F,N);
```

Since the arrays are called by name, a call such as MATRIXMULT(E,E,F,N); would give rather interesting results.

This call could be made to work by calling B and C of MATRIXMULT (A, B, C, N) by value. However, this would increase the overhead of the procedure considerably.

11.5 ADVANCED USE OF PROCEDURES

11.5.1 Jensen's Device

This method of using a procedure exploits the power and flexibility of the call-by-name concept. Consider the following example:

```
    REAL PROCEDURE SUM(I,N,X); VALUE N; INTEGER I,N; REAL X;
    BEGIN   REAL Y;
            Y := 0
            FOR I := 1 UNTIL N DO Y := Y + X;
            SUM := Y
    END
```

On the surface, the procedure appears to calculate the value of N*X. However, consider the call

```
    Z := SUM(J,10,A[J]);
```

and remember that J and A[J] are parameters called by name. Since I and consequently J take new values, each X in the loop is evaluated as a particular value of A[J], using the value of J just assigned. Hence the above call calculates

```
    A[1] + A[2] + ..... + A[10].
```

Similarly, the call

```
    Z := SUM(K,M,A[I,K]*B[K,J]);
```

calculates the (I,J)th inner product of A and B.

11.5.2 Recursion

ALGOL procedures are recursive, that is, they may call themselves, directly or indirectly, to any reasonable depth. (The only restriction is the amount of core storage available to the object program.) An often-quoted and very inefficient method of calculating the factorial function of a small positive integer N is:

```
INTEGER PROCEDURE FACTORIAL(N);  VALUE N;  INTEGER N;

IF N = 1 THEN FACTORIAL := 1

ELSE FACTORIAL := N*FACTORIAL(N-1);
```

This procedure has only a single statement, but no local variables, and can therefore be written in a compact form. A call such as

```
J := FACTORIAL(6);
```

causes the procedure to be entered with N equal to 6. The call to FACTORIAL inside FACTORIAL enters the procedure a second time with N equal to 5, but this N is different from the one to the previous N, which retains its value of 6, and is stored in a different space. In this particular case, FACTORIAL is entered six times, the last time with N equal to 1.

11.6 LAYOUT OF DECLARATIONS WITHIN BLOCKS

Declarations must always be made at the head of a block, before any assignments, procedure calls, etc., in the following order: 1) scalars and arrays and 2) procedures and switches (see Chapter 12).

Procedure bodies that occur in a block should follow the declarations at the head of the block, although this is only enforced when necessary. Consider the following example:

```
BEGIN

PROCEDURE P(X);  VALUE X;  REAL X;

BEGIN   INTEGER J;

        .....

        J := I;

        .....

END;

INTEGER I;
```

The assignment of I to J within the body of P utilizes the I that is declared following the body of P, rather than some global I. However, the compiler has not yet "seen" this I and, therefore, cannot take any rational action. In a case such as this, the user must declare I before the body of P:

```
BEGIN   INTEGER I;

PROCEDURE P(X);  VALUE X;  REAL X;

BEGIN   INTEGER J;

        .....

        J := I;

        .....

END
```

If the user neglects to declare I before P, the compiler can easily detect the condition, because either I is unknown at the time of the assignment to J, or else there is a more global I available, whereupon an error message will occur when the declaration of I is found following the body of P.

11.7 FORWARD REFERENCES

Although most ALGOL-60 compilers operate in two or more passes, the DECsystem-10/20 ALGOL compiler operates in one pass. Consequently, some minor restrictions have to be made to ALGOL-60 in order not to restrict the user in other ways.

A forward reference for a procedure has to be given when a procedure is called (either directly, or indirectly, by passing the procedure name as an actual parameter in a procedure call) before its body is encountered by the compiler. In most cases the user can avoid this situation by a minor re-ordering of the program. However, in rare cases like the following, where procedure P calls procedure Q, and vice versa, a forward reference, as shown, must be given.

```
BEGIN

FORWARD REAL PROCEDURE Q;

PROCEDURE P(X);  VALUE X;  REAL X;

BEGIN REAL Y;

          .....

          Y := Q(X);

          .....

END;

REAL PROCEDURE Q(Z);  VALUE Z;  REAL Z;

BEGIN REAL F;

          .....

          F := P(Z);

          .....

END;
```

In general, a forward reference consists of the word FORWARD, followed by the type of the procedure (omitted if the procedure is typeless), the word PROCEDURE, and the name of the procedure. For example:

```
FORWARD LONG REAL PROCEDURE INTEGRATE
```

or

```
FORWARD PROCEDURE PROBLEM
```

NOTE

The forward reference must occur in the same block as the procedure body.

A forward reference has to be given for a label in either of the following rare cases:

1. The label is used as an actual parameter in a procedure call, and has not yet appeared in the program.

2. A variable of identical name has appeared in the program and is in the scope of the procedure call.

For example:

```
BEGIN REAL L;
        .....
BEGIN FORWARD L;
        .....
        P(L);
        .....
L;  .....
END;
        .....
```

In this case, a forward reference for L must be given.

11.8 EXTERNAL PROCEDURES

If a procedure is to be compiled independently of a program (see Paragraph 18.1.1), an EXTERNAL declaration must be made in the program instead of the procedure. The form of this is the same as that of a FORWARD declaration, but with the word FORWARD replaced by EXTERNAL. For example:

```
EXTERNAL INTEGER PROCEDURE CALC
```

Such an EXTERNAL declaration can be made in any block within the program, and has the same scope as if the procedure appeared at that point.

At present all EXTERNAL procedure names referenced in a program or scanned in a library must differ in their first six characters, as only the first six characters are available to LINK.

11.9 ADDITIONAL METHODS OF COMMENTARY

Two further ways of writing commentary are available to the user in addition to COMMENT and ! described in Section 2.4.

11.9.1 Comment After END

Following the delimiter word END, the user may add commentary, terminated by a semicolon, with the following restrictions:

1. The commentary may only contain letters and digits.
2. If the reserved delimiter word mode of compilation is employed, any words appearing in the comment may not be delimiter words.

For example:

```
END OF PROC INVERT;
```

11.9.2 Comments Within Procedure Headings

This method of commentary allows the user to comment formal parameters in a procedure heading. This is done by enclosing the commentary, which may consist of letters only, between the symbols) and :(and omitting the comma on the left of the formal parameter. This cannot apply to the first formal parameter.

The example in Section 11.6.1 which is:

```
REAL PROCEDURE SUM (1, N, X);
```

can thus be rewritten as

```
REAL PROCEDURE SUM(I) COUNT:(N) INCREMENT:(X);
```

In a similar fashion, a call to such a procedure can be commented. The following example uses the call to SUM in Section 11.6.1:

```
Z := SUM (K, M, A[I,K]*B[K,J]);
```

to be commented as:

```
Z:=SUM(K) COUNTER:(M) CROSS PRODUCT:  (A[I,K]*B[K,J]);
```

CHAPTER 12

SWITCHES

12.1 GENERAL

Switches enable the user to jump to one of a number of labels, depending on the value of an arithmetic expression, and in addition, provide automatic detection when such an expression is out of range for the switch.

12.2 SWITCH DECLARATIONS

A switch declaration takes the form of the word SWITCH followed by the name of the switch, an assignment (:=), and a list of labels. These are called switch elements, and must be in the scope of the switch declaration. For example:

```
SWITCH SW := LAB,L1,L2,OK,STOP;
```

A switch name must follow the usual rules of scope and, therefore, must not conflict with any local variable of the same name.

In addition to the example above, a switch element may also be one of the labels in the switch declaration.

12.3 USE OF SWITCHES

A jump to a particular label in a switch declaration is made by following the word GOTO with the name of the switch and an arithmetic expression in square brackets. Thus:

```
GOTO SW[I]
```

This causes control to pass to the I'th label in the switch declaration, unless I is negative or zero, or is larger than the number of switches in the switch declaration. In either case, there is no transfer of control. If the expression in square brackets is not integral, it is evaluated and rounded as usual. Consider the following more complicated example:

```
SWITCH SW := LAB,L1,L2,OK,STOP;

SWITCH TW := L3,SW[J],L4;

.....

GOTO TW[I];
```

If I has the value 3, a jump to L4 occurs. If I has the value 2 and J has the value 1, a jump to LAB occurs, via SW.

More sophisticated switch elements are described in Chapter 14.

CHAPTER 13

STRINGS

13.1 GENERAL

DECsystem-10/20 ALGOL-60 includes a major extension to the string features defined in the Revised Report. Users wishing to run their programs on machines other than the DECsystem-10 should check whether the compiler they will use offers similar facilities. Scalar, array or procedure variables may be of type STRING, and are declared by the delimiter word STRING. The byte size, length and contents of string variables are defined via the various assignment statements described below.

Typical string declarations might be:

```
STRING S,T;  STRING ARRAY SA[1:10];

STRING PROCEDURE B(X);  VALUE X;  REAL X;
```

13.2 STRING EXPRESSIONS AND ASSIGNMENTS

String expressions are limited to a single variable, a string procedure call or a string constant. (For a full description of string constants see Section 4.4.) The only string operators are the comparison operators and the assignment operator. All other operations are achieved via the string library procedures described in Section 13.7.

String expressions can be assigned only to string variables. For example:

```
S:=T;

SA[I]:=SA[3];

SA[2]:=B(Z);

T:="ANY ""OLD"" IRON";
```

13.3 BYTE STRINGS

The value associated with a string variable is a byte string. A byte string is a sequence of bytes of a uniform size between one and thirty-six, which can be efficiently handled by the DECsystem-10/20 hardware. In some ways, byte strings can be thought of as arrays, but the most important difference is that the size of a byte string can vary continuously. Thus when a byte string is created by means of a

READ statement, the programmer need not know how long the string will be. The routine starts accepting characters after encountering the open quotation marks and continues until the close quotation marks have been read.

When one string is assigned to another, e.g.,

```
S:=T;
```

then a copy of T is made to which S will refer. Any previous value of S is destroyed (and the memory space occupied is released). Subsequent changes to the value of T will not affect S, or vice versa, unless a further assignment is made from one to the other.

NOTE

This is an important change from the implementation of strings prior to Version 5.

13.4 BYTE SUBSCRIPTING

Byte strings can be modified by means of the byte subscripting mechanism. Individual bytes in a string are referenced by following the string variable name by a decimal point and then the subscript number enclosed in square brackets. For example

```
S.[I]
```

refers to the I'th byte of string S. The subscript may be any arithmetic expression and is evaluated in exactly the same way as an array subscript.

Byte-subscripted string variables are regarded as being of type integer, having an integer value equivalent to the byte to which they refer. Therefore, to change the value of a particular byte in a string, a byte-subscript must appear on the left-hand side of an arithmetic statement with the appropriate new value on the right-hand side. If the new value is too large to be held in the byte, this is simply truncated. No warning is given.

13.5 NULL STRINGS

Until a value is assigned to a string by the program, the string takes null value. That is, it is assumed to contain no bytes. Any attempt to reference the string by a byte-subscript will result in a fatal run-time error, though it can be used on the right-hand side of a string assignment, in which case the variable to which it is assigned similarly becomes null.

13.6 STRING COMPARISONS

Two byte strings can be compared with each other using the usual comparison operators. For example

```
IF S < T THEN GO TO L;
```

where S and T are string variables, string constants or string procedures. The effect of the comparison is to compare the strings byte-by-byte, the "lesser" string being that with the first lower value byte, working from left to right. Thus 'ABCD" is less than "ABCE" or "ABCDE". Where the strings to be compared are of different byte sizes, then the smaller bytes are regarded as being extended on the left by null bits.

In the special case of ASCII strings (strings of byte size 7, like string constants), trailing nulls and trailing blanks, or any mixture thereof, are treated as equal. Similarly ASCII strings of different lengths will compare equally if the extra length comprises only spaces and nulls. In all other cases strings of unequal length can only be regarded as equal if the extra length consists entirely of null bytes.

13.7 LIBRARY PROCEDURES

Section 16.6 deals with the input and output procedures that are applicable to strings.

The procedures LINK, LINKR and TAIL that were included in the library until Version 3B have been dispensed with.

13.8 CONCATENATION

A string can be assigned the concatenated value of two strings with the procedure CONCAT. For example

```
S:=CONCAT(T,U);

S:=CONCAT(S,T);
```

If, in the first example, T had a different byte size from U, then the size of the first string encountered (T in this case), would be adopted by S. The bytes copied from U would be truncated or filled with null bits as appropriate.

13.8.1 Length And Size

The primary attributes of a string, that is, length in bytes and byte size in bits, are returned by the integer procedures LENGTH and SIZE, respectively.

Thus

```
I:=LENGTH(S);  J:=SIZE(S);
```

would return the number of bytes in string S in I, and the number of bits in each byte in J.

13.8.2 Copying

A new byte string can be generated from an existing one by means of the string procedure COPY. This procedure can have one, two or three parameters.

1. The effect of COPY with one parameter is precisely the same as a simple string assignment, but this feature has been retained for the sake of continuity.

2. Where there are two parameters, e.g.,

```
S:=COPY(T,M);
```

   where M is an arithmetic expression, then S is assigned the value of the first through M'th bytes of T.

3. If there are three parameters, e.g.,

```
S:=COPY(T,M,N);
```

   where both M and N are arithmetic expressions, then S is assigned the value of the M'th through N'th bytes of T.

13.8.3 **Newstring**

Although bytes in a null string may not be referred to (see Section 13.5 above), a string containing nulls of any appropriate byte sizes can be created by using this string procedure. NEWSTRING takes two parameters, the first being the number of null bytes to be assigned to the string, and the second their size.

For example

```
S:=NEWSTRING(100,7);
```

causes a null string of 100 ASCII nulls to be assigned to S. Although S is 100 bytes long at this point (and thus byte subscripts up to and including S.[100] are valid), any subsequent assignment of another string to S may vary both the length and byte-size of S.

13.8.4 **Delete**

In Section 13.5 it was explained that if a null string is assigned to another string, then that string also becomes null, and the value previously held is lost. Any space that the previous value occupied is returned to memory, as with any ordinary string assignment. The typeless procedure DELETE has the same effect on the string passed to it as a parameter, as the assignment of a null string would have. Deleting a null string has no effect, beyond using computer time.

CHAPTER 14

CONDITIONAL EXPRESSIONS AND STATEMENTS

14.1 GENERAL

ALGOL-60 allows great flexibility in the construction of expressions and conditions.

Consider, for example, a variable I which could be set equal to 0 or 1 according to the value of a Boolean variable B: this could be written as:

```
I := 0;

IF B THEN I := 1;
```

Also, consider the case where a user wants to perform some action, depending on the value of B:

```
IF B THEN X1 := Y;  IF NOT B THEN X2 := Y;
```

14.2 CONDITIONAL OPERANDS

ALGOL-60 allows the user to substitute a conditional operand for any operand in an expression by the use of a construction involving IF ..... THEN ..... ELSE.

For instance, the first example above can be rewritten

```
I := IF B THEN 0 ELSE 1;
```

Clearly, this is more compact and of great use in cases such as:

```
J := J + (IF K < 1 THEN 1 -K ELSE K-1);
```

Note that the conditional operand must be bracketed, and may only be unbracketed when it forms the complete expression itself.

In general, a conditional operand may replace an operand in any arithmetic or Boolean expression. In addition, a conditional operand may also replace a label and act as an element in a switch list, for example:

```
SWITCH SW := L1, IF B THEN L2 ELSE L3, L4;
```

It is also permitted, in an array subscript (and also in a byte subscript), for example:

```
X := A[I, IF L = 0 THEN J ELSE J+1];
```

Since a conditional operand may replace any operand in an expression, operands may also be replaced in conditional expressions. Consider the following example:

```
IF IF B THEN B1 ELSE B2 THEN I := I + 1;
```

This looks complicated but is really quite simple if brackets are inserted for clarity. Thus:

```
IF (IF B THEN B1 ELSE B2) THEN I := I + 1;
```

14.3 CONDITIONAL STATEMENTS

The reader was introduced to conditional statements of the form

```
IF B THEN S1 ELSE S2
```

in Chapter 7. The full power of this type of statement can now be demonstrated.

First, S1 and S2 can be compound statements or blocks. For example:

```
IF I < 0 THEN

BEGIN          I := -I;  B := FALSE

END ELSE

BEGIN          I := I + 1;  GOTO L2

END
```

Second, the whole structure of the IF ..... THEN ..... ELSE statement can be made more powerful by using conditional statements within themselves. For example:

```
IF X < 0 THEN X := 0 ELSE IF B THEN GOTO L
```

This is equivalent to the following sequence of statements:

```
     IF NOT X < 0 THEN GOTO L1;

     X := 0;  GOTO L2;

L1:  IF NOT B THEN GOTO L2;

     GOTO L;

L2:
```

Clearly the former method of expression is both briefer and more elegant. Conditional statements take the general form

```
IF B THEN S1 ELSE S2
```

where S1 and S2 may both be conditional statements. However, if there is any ambiguity, bracketing using BEGIN and END must be used to clarify this. Consider the following example:

```
IF B THEN IF X = 0 THEN Y := Z ELSE P := Q;
```

This could be interpreted as

```
IF B THEN
BEGIN
IF X = 0 THEN Y := Z
END
ELSE P := Q
```

or

```
IF B THEN
BEGIN
IF X = 0 THEN Y := Z ELSE P := Q
END
```

The first case is interpreted as:

```
          IF NOT B THEN GOTO L1;
          IF NOT X = 0 THEN GOTO L2;
          Y := X;  GOTO L2;
     L1:  P := Q;
     L2:
```

The second case is interpreted as:

```
          IF NOT B THEN GOTO L2;
          IF NOT X = 0 THEN GOTO L1;
          Y := Z;  GOTO L2;
     L1:  P := Q;
     L2:
```

ALGOL-60 forbids such ambiguities by forbidding the sequence THEN IF ..... THEN ..... ELSE.

14.4 DESIGNATIONAL EXPRESSIONS

A designational expression is something that acts as an argument in a GOTO statement, either directly, or indirectly via a formal procedure parameter of type label. This may simply be a label or a switch element. Thus the following are designational expressions:

```
L
IF B THEN L1 ELSE L2
IF X < 0 THEN SW[I] ELSE IF X+Y >= Z THEN TW[J] ELSE L
```

These designational expressions would be used in the following manner:

```
GOTO L;
GOTO IF B THEN L1 ELSE L2;
GOTO IF X < 0 THEN SW[I] ELSE IF X+Y >= Z THEN TW[J] ELSE L;
```

CHAPTER 15

OWN VARIABLES

15.1 GENERAL

OWN variables are a special kind of ALGOL variable, and may be of type integer, real, long real, Boolean or string, either scalar or array. The variables have the following properties:

1. Although following the normal scope rules, the variables are not recursive; the same copy of each variable being used in all occurrences of a procedure or block.

2. When control passes out of a block, the values are retained and are still available when the block is re-entered.

3. The initial value is set to zero before execution of the program. (FALSE in the case of Boolean OWN variables.) OWN STRINGS are initialized to possess no byte string.

OWN variables are declared by writing the usual declaration with the word OWN preceding it. For example:

```
OWN INTEGER I,J,K;

OWN REAL ARRAY THETA[1:M];
```

15.2 OWN ARRAYS

OWN arrays are implemented in a completely dynamic fashion in DECsystem-10/20 ALGOL. The declaration proceeds according to the following rules.

1. If this is the first time the array is declared, space is obtained and then the array laid out. If the array has been laid out before, proceed to Step 2.

2. The bounds are examined to ensure that these are identical to the ones of the previous construction of this array, and the array is left unaltered if found to be of the same dimension; otherwise, proceed to Step 3.

3. A new array is constructed and the common elements if any, are copied from the old array; the remaining elements are zeroed. The old array is then deleted and the allocated space is recovered for future use.

For example, if an OWN array A is declared as follows:

```
OWN REAL ARRAY A[1:M,M:N];
```

where M = 2 and N = 5 the first time, and M = 1 and N = 4 the second time, the elements [1,2], [1,3] and [1,4] are copied over, and the remaining elements of the new array are zeroed.

CHAPTER 16

DATA TRANSMISSION

16.1 GENERAL

Data transmission encompasses the input and output of data between the user's program and peripheral devices, such as disk, DECtape, magnetic tape, card reader, card punch, and line printer. The DECsystem-10/20 ALGOL object-time system, in conjunction with the ALGOL library, provides the user with a set of basic procedures for handling data from most DECsystem-10 or DECsystem-20 devices in a uniform fashion. The user may also perform input/output operations with virtual peripherals that appear as byte strings in the user's program.

All peripheral devices are under the user's control completely and can be allocated or released at any time throughout the execution of the program. The user can handle up to sixteen devices simultaneously (seventeen, if one of them is the terminal attached to the job), any number of which may be file devices (disk, DECtape) and have independent files open.

16.2 ALLOCATION OF PERIPHERAL DEVICES

Peripheral devices are allocated to the user's program by calls to the library procedures INPUT or OUTPUT. A call to one of these procedures usually has two parameters. The first is the channel number, an integer in the range 0 to 15, on which the device is to operate. Only one device at a time may be operated on a channel. A channel provides either input or output facilities, except in the case of a terminal, where the input and output functions are performed simultaneously on the same channel. The second parameter is either a string or a string constant. The text contained in the string is the logical name of the device to be allocated to this channel.

The DECsystem-10 or DECsystem-20 Users Handbook should be consulted, as appropriate, for an explanation of what constitutes a logical device name. In the simplest case, the name may be the actual name of the peripheral device. Device names shown in Table 16-1 are recognized as standard.

Table 16-1
Standard Device Names

Device Name	Peripheral
DSK	Disk
DTA	DECtape
MTA	Magnetic tape
CDR	Card reader
CDP	Card punch
LPT	Line printer
PTR	Paper-tape reader
PTP	Paper-tape punch
PLT	Plotter
TTY	Terminal

For example, to allocate the card reader for use as an input device on channel 5, the user would use the statement

```
INPUT(5,"CDR");
```

or, if S were a string possessing a byte string that had the characters CDR in it,

```
INPUT(5,S);
```

Similarly, if the disk were to be used an as output device on channel 9:

```
OUTPUT(9,"DSK");
```

NOTE

With the exception of terminals, all devices are allocated to operate in one direction only; thus, if the user wants input and output from the disk, two separate channels must be used.

Terminals are always allocated bi-directionally, irrespective of whether the user uses INPUT or OUTPUT. For example,

```
INPUT(0,"TTY");
```

allocates the user's terminal for input and output on channel 0.

16.2.1 Device Modes

Normally, a device is allocated in ASCII mode, that is, when the user reads a character from the device, the readable text, such as a stored source program or data is represented by a 7-bit byte. To allocate the device in a different mode, a third parameter is specified in the call to the INPUT or OUTPUT procedure. Thus, to allocate a disk to channel 9 in binary image mode (the mode used for the storage of binary data on a disk), the user can use

```
OUTPUT(9,"DSK",11);
```

The DECsystem-10 or DECsystem-20 Assembly Language Handbook should be consulted, as appropriate, for a full explanation of the different modes used with peripheral devices. The INPUT and OUTPUT procedures allow the user to allocate any standard peripheral device in any buffered mode.

16.2.2 Buffering

The INPUT and OUTPUT procedures normally allocate two buffers for each allocated device (terminals are allocated two buffers for input and two for output). The user may desire to use either one or more than one buffer for a device. For example, in a non-compute bound job that uses a lot of disk transfers at odd intervals, four or even eight buffers may be desirable to increase the speed of execution of the program.

The number of buffers to be used can be controlled by adding a fourth parameter to the procedure call. Thus, to allocate a disk on channel 14 in mode 0 with eight buffers, the call is

```
OUTPUT(14,"DSK",0,8);
```

NOTE

The mode must always be specified when allocating buffer space, otherwise there would be an ambiguity in the third parameter.

16.2.3 Error Returns

Normally, if the device allocation fails (for example if the device is in use by another job), a suitable message is typed and the program terminated. The user can prevent this by providing, as the fifth parameter to INPUT or OUTPUT, a label to which control is to be passed in the event of an error. For example:

```
OUTPUT(14,"MTA",0,0,ERROR.LABEL):
```

If the actual label parameter is a switch whose subscript is out of range, the procedures behave as though the label parameter were absent.

NOTE

The third and fourth parameters must be specified in this case to avoid ambiguity. However, if zeros are specified, then default values will be taken. The default being ASCII mode and 2 buffers for the third and fourth parameter respectively.

16.3 SELECTING INPUT/OUTPUT CHANNELS

Before a user uses a device to transfer data, assuming that the device has already been allocated to some channel, the appropriate input or output channel must be "selected" for use as the input or output channel. All data input and output always occurs on the currently selected input channel and output channel, respectively. The user may change the selection of channels at any time, switching from one channel to another without loss of data, irrespective of whether complete lines (or records) of data have been read or not. In fact, the DECsystem-10/20 input/output system does not assume any structure in the data: all input and output channels are regarded as pipelines through which the user pulls or pushes data.

To select an input channel, a call to the procedure SELECTINPUT must be made. This has one parameter, which is the channel number. Thus

```
SELECTINPUT(5);
```

causes input channel 5 to be selected.

Similarly, the procedure SELECTOUTPUT is used to select an output channel.

16.4 FILE DEVICES

Some peripheral devices, such as disk and DECtape, require the opening of a specifically named file before any input or output operations can be performed. This optionally may be performed on spooled devices (refer to the appropriate Operating System Commands manual for a description of spooling). The opening of this file is performed by means of the procedure OPENFILE, which is called after the device has been allocated to a channel. The procedure call has two parameters: the channel number on which the device has been allocated and a string variable possessing a byte string or a string constant, the text of which is the name of the file.

The user can also specify a protection and/or project-programmer number of a file by means of optional third and fourth Boolean or integer parameters. For example, to open a file with protection 177 on disk area [11,50] the user could write

```
OPENFILE (9,"TEST.DAT",%177,%000011000050);
```

When a user has finished with a file it should be closed. A file is closed by using the procedure CLOSEFILE, with a parameter that is the channel number on which the file is open. Thus,

```
CLOSEFILE(9);
```

closes the file that is open on channel 9.

The user may also rename or delete existing files: if a file is already open, use of OPENFILE causes the file to be renamed with the new name supplied. Thus the sequence

```
OPENFILE (5,"TEST1.DAT");

OPENFILE (5,"TEST2.DAT");
```

causes the file with name TEST1.DAT to be renamed TEST2.DAT. If the string containing the new name is null, the original file is deleted.

Thus,

```
OPENFILE (5,"TEST3.DAT");

OPENFILE (5,"");
```

causes the file TEST3.DAT to be deleted.

16.4.1 Error Returns

Normally, if the operation requested fails (for example an input file does not exist), a suitable message is typed and the program terminated. The user can prevent this by providing a label and an optional integer variable as the fifth and sixth parameters to OPENFILE. In the event of an error, control will be passed to the label, with an error-code set into the integer variable if present. The error-codes are those returned by the ENTER and LOOKUP UUO'S (refer to Appendix E of the DECsystem-10 Monitor Calls Manual or Appendix A of the DECsystem-20 Monitor Calls Manual, as appropriate).

NOTE

The third and fourth parameters must be present if the error return parameter is specified. Defaults will be taken if both parameters are specified as zero.

If the actual label parameter is a switch whose subscript is out of range, the procedure behaves as though the label parameter were absent. The integer error-code parameter is called by name.

16.5 RELEASING DEVICES

The procedure RELEASE is used to release a device from a channel. Thus,

```
RELEASE(5);
```

releases the device allocated to channel 5. If the device is a file device, and a file is still open on the device, this will be automatically closed. Releasing a device on a channel causes a channel to become free; if this channel is currently selected for input or output operations, it is deselected.

If an attempt is made to allocate a device to a channel that already has a device allocated, the allocated device is first released and, if a file is open on the device it is closed before the release.

If a user terminates his program without releasing devices on channels, these are automatically released.

16.6 BASIC INPUT/OUTPUT PROCEDURES

16.6.1 Byte Processing Procedures

The following procedures may be used with any device to handle bytes of any standard size (1 to 36 bits). However, because they are normally used with devices supplying or accepting ASCII bytes, they are "symbol" oriented.

1. INSYMBOL(S); - (where S is usually some integer variable) causes the next byte to be read from the currently selected input channel and stored in S.

2. OUTSYMBOL(J); - (where J is usually some integer expression) causes the value of J to be output as a byte to the currently selected output channel. If J is too large for the byte size of the device in use, it is truncated to size.

3. NEXTSYMBOL(S); - acts in exactly the same way as INSYMBOL except that the byte pointer for the input channel is not advanced to the next available byte. This gives the user a look-ahead facility of one byte.

4. SKIPSYMBOL; - causes the next byte from the selected input channel to be read and ignored.

5. BREAKOUTPUT; - causes all bytes in the buffer of an output device to be sent immediately to it. This procedure is normally used to conduct a question-and-answer dialogue on a terminal, with the question and answer on the same line. Normally, a block of data is sent to a device only when the buffer is full (the exception being the terminal, where a break is sent at the end of each line).

16.6.2 String Output

A byte string may have its contents transferred to the currently selected output channel by means of the procedure WRITE, whose single parameter is either a string constant or a string variable that possesses the string to be output. For example:

```
WRITE(S);
```

or

```
WRITE("THE MOON IS MADE OF GREEN CHEESE");
```

With exceptions explained in the following paragraphs, all of the bytes in the string are output literally, with the exception, of course, of the quotes in a string constant, which are not in fact stored in the bytes string at all.

NOTE

Unlike some other ALGOL implementations, spaces and other non-printing symbols in byte strings are meaningful in DECsystem-10/20 ALGOL.

Special editing characters are permitted within square brackets within the text of a byte string. These have a special function:

P	Page throw
C or N	New line (C stands for carriage return, line feed)
T	Tab
S	Space
B	Break output

Any combination of these characters, with optional preceding repetition counts, can appear within square brackets in a byte string and are output as their special interpretation demands. For example:

```
WRITE("ABCD[P2C5S]EFGH");
```

causes the following to be output:

1. the symbols ABCD followed by a page throw
2. two new lines and five spaces
3. the symbols EFGH.

To output the symbols

```
[ ] " or;
```

these must appear in the form

```
[[ ]] "" or ;;
```

respectively. Thus

```
WRITE(""A[[I]] := 3;;"");
```

causes the text

```
"A[I} := 3;"
```

to be output.

16.6.3 Miscellaneous Symbol Procedures

The procedures SPACE, TAB, PAGE, and NEWLINE cause the appropriate number of spaces, tabs, page throws, or new lines to be output. This number is specified by a single integer parameter. If the parameter is omitted a value of one is assumed. Thus

```
SPACE(5);
```

causes five spaces to be output, whereas

```
SPACE;
```

or

```
SPACE(1);
```

cause one space to be output.

16.6.4 Numeric and String Procedures

Numeric procedures are used to read and print numeric quantities. The procedures will normally be used with a device that is operating in ASCII mode, and are capable of processing integer, real, or long real quantities in fixed-point and floating-point representation.

16.6.4.1 **Numeric Input Data** - Numeric data for input can be represented in any format that would be acceptable as a numeric constant in a program, irrespective of the type of variable involved. When a number is read, an automatic type conversion is performed, giving a result of the same type as if an assignment of the data represented as a constant in the program had been executed.

There is a minor restriction in that no spaces, tabs, or other non-printing symbols may appear in such numeric data except between the exponent sign (& or @ for real, && or @@ for long real) and the exponent. Otherwise, any symbol that is not a part of a numeric quantity may act as a terminator for such a quantity. It is strongly recommended that spaces, tabs, or new lines be used as separators. For example:

```
3.4 -9.6 1.36 -52

0 14.9
```

NOTE

> In reading a numeric quantity, the terminating symbol, that is, the first symbol that is not part of the number, is lost.

DECsystem-10/20 ALGOL also allows the user to input floating-point data written in FORTRAN format, that is, using E for & or @, and D for && or @@. However, no other special effects inherent in FORTRAN formatting are introduced.

The procedure READ is used to input numeric data and also strings. This procedure may have any number of parameters (up to an installation-dependent maximum), of type integer, real, long real, Boolean, or string.

The effect is as follows:

1. For integer, real and long real variables, a number is read and converted to the type appropriate to the parameter and then assigned to the variable.

2. For Boolean, a number is read as if for an integer variable, and assigned to the variable.

3. For a string variable, the data text is scanned until a quote (") is found, and the text following this up to but not including the next free quote is read in and a byte string generated, which is then possessed by the string variable.

If the sequence "" is found, a single " is stored, and reading of the string continues.

16.6.4.2 **Numeric Output Data** - Numeric data is output by means of the procedure PRINT. This procedure may have one, two, or three parameters, the first of which is the variable to be printed. This variable may be an integer, real, or long real. The second and third parameters determine the format to be used and are integer expressions. If omitted, both parameters are assumed to be zero. The effect of the various combinations of the format integers, M and N, is as follows:

M>0, N>0: Fixed-point printing, M places before the decimal point, N places after. A sign, space if positive, - if negative appears before the number. Zeros before the decimal point are replaced by spaces and the sign moved up to the number.

This format always outputs M+N+2 symbols.

M>0, N=0: The same as the preceding except that (1) no fractional part appears, and (2) the decimal point is suppressed.

This format always outputs M+1 symbols.

M=0, N>0: Floating-point format, consisting of a sign, a decimal digit, a decimal point, N more decimal digits, and an exponent consisting of & for real, && for long real followed by the exponent sign and a two-digit exponent, zero suppressed from the left.

This format outputs N+7 symbols for real and N+8 symbols for long real quantities.

If only two parameters appear, format M,0 is assumed for integer variables, and format 0,N for real and long real quantities, where M and N take, respectively, the value of the second parameter.

If only one parameter appears, the format is interpreted as 0,0 which assumes standard printing modes of 11,0 for integer quantities, 0,9 for real quantities, and 0,17 for long real quantities.

If the user requests more digits to be printed than are significant in real or long real numbers, the appropriate number of zeros follow a properly rounded printing of the number to the maximum precision available.

16.6.4.3 **Octal Input/Output** - The procedures READOCTAL and PRINTOCTAL, respectively, allow the user to input and output quantities in octal format.

On input, for single precision variables, up to 12 octal digits are read, preceded by the symbol %, the terminator being any non-numeric symbol. For long real variables, two such octal numbers must be presented for input, each preceded by the symbol %.

On output, 12 octal digits, preceded by the symbol %, are printed for single precision variables. For long real variables, two quantities each with 12 octal digits are printed separated by a space.

The foregoing procedures have one scalar parameter which may be of type integer, real, long real or Boolean.

16.7 DEFAULT INPUT/OUTPUT

If the user does not select any input or output channels, input and output occur via an "invisible" channel from and to the user's terminal. Thus, for simple programs where the user wishes to input a few numbers and print a few results, he simply uses READ, types in the data on line through his terminal, and gets back the results from PRINT.

16.8 LOGICAL INPUT/OUTPUT

In addition to the 16 channels used to communicate with peripheral devices, an additional 16 channels, numbered from 16 to 31, are provided. These are input or output channels that use byte strings as a means of storage.

By means of the procedures INPUT or OUTPUT, the user can attach a channel to a byte string possessed by a string variable, and can read and write bytes from and to this byte string, either to and from a peripheral device, or to and from another byte string.

```
INPUT(20,S);
```

or

```
OUTPUT(20,S);
```

cause the byte string possessed by the string variable S to be used as logical channel 20; this channel may subsequently be selected for input or output, as appropriate.

The user is still free, of course, to manipulate the individual bytes within the byte string by means of the byte-subscripting facilities available. Such facilities enable the user to read a file from a peripheral device into a string, process it in any way whatsoever, and output it again.

16.9 SPECIAL OPERATIONS

These procedures are used on channels assigned to magnetic tapes, and perform operations of BACKSPACE, ENDFILE and REWIND. Each procedure takes one parameter, that is, the channel number on which the operation is to be performed.

Since there is no implicit structure on a magnetic tape, these procedures enable the user to build up formats in any way he chooses.

16.10 I/O CHANNEL STATUS

The status of any input or output channel can be determined at any time by means of the Boolean procedure IOCHAN, which takes an integer channel number as parameter. The status returned is bit coded as follows:

Bit	Value	Meaning if Set
18	%400000	Device is physical (i.e., not logical)
19	%200000	Directory device
20	%100000	Terminal device
21	%040000	ASCII mode
22	%020000	Magnetic tape
23	%010000	Plotter
24	%004000	Set for default TTY on channel -1
25	%002000	Device is spooled
26	%001000	Device can do input
27	%000400	Device is initialized for input
28	%000200	File is open for input
29	%000100	End of file encountered
30	%000040	Input status ok
31	%000020	Device can do output
32	%000010	Device is initialized for output
33	%000004	File is open for output
34	%000002	Device quota exceeded
35	%000001	Output status ok

Some of these bits are of little use to the user, but, for example, if a device is allocated, and the user does not know whether or not the device is file-structured, IOCHAN can be used to determine this. The bits of particular use to the user are the input and output end-of-file.

NOTE

An end-of-file on output is a logical status indicating that, for example, a disk quota is exceeded or a DECtape is full, or in the case of a logical device, the byte string is full.

When IOCHAN is used, the end-of-file flags are always cleared, if set, so that the user may proceed to read a magnetic tape after an end-of-file marker is found.

The following example shows how the user would handle an unknown device whose name is given to the program via the user's terminal:

```
BEGIN
    STRING DEVICE, FILE; INTEGER CHANNEL;
    WRITE ("CHANNEL NO: "); BREAK.OUTPUT;
    READ (CHANNEL);
    WRITE ("[C]DEVICE NAME: "); BREAK.OUTPUT;
    READ (DEVICE);
    OUTPUT (CHANNEL, DEVICE);
    IF IOCHAN (CHANNEL) AND %200000 THEN
    BEGIN
    WRITE ("[C]FILE NAME: "); BREAK.OUTPUT;
    READ (FILE);
    OPENFILE (CHANNEL, FILE)
    END;
    ..............
END
```

NOTE

When using Boolean expressions involving IOCHAN, the rules for evaluation in this implementation should be borne in mind. See Section 5.2.1.

16.11 TRANSFERRING FILES

Once devices have been allocated to an input and an output channel, a complete file of information may be transferred between them automatically by calling the parameter-less procedure TRANSFILE. This procedure copies bytes from one device to another from the currently selected input channel to the currently selected output channel, until an end-of-file status is raised on either the input or output channel.

16.12 CURRENTLY SELECTED CHANNEL NUMBERS

The number of the channel currently selected for input or output may be obtained by use of the integer procedures INCHAN or OUTCHAN.

CHAPTER 17

THE DECSYSTEM-10/20 OPERATING ENVIRONMENT

The operating environment of DECsystem-10/20 ALGOL programs consists of those procedures in the DECsystem-10/20 ALGOL Library required by the user's program, and the DECsystem-10/20 ALGOL Object Time System.

The former are those procedures detailed in Chapters 13 and 16, together with those described below. These procedures can be thought of as existing in a block surrounding the user's program, and, therefore, are available when called. The names of these procedures, however, are in no sense reserved as are words such as BEGIN.

Note that these procedures are only present in the user's program when required. They are loaded by the DECsystem-10 or DECsystem-20 Linking Loader when so directed by the DECsystem-10/20 ALGOL Compiler. The user is not required to take any action to include these procedures, other than make a call to them. A complete list of library procedures is given below.

17.1 MATHEMATICAL PROCEDURES

The following procedures expect one argument, of real type, and yield a real type result.

Procedure Name	Function
SIN	Sine
COS	Cosine
ARCTAN	Arctangent
SQRT	Square root
EXP	Exponential
LN	Logarithm (to base e)
TAN	Tangent
ARCSIN	Arcsine
ARCCOS	Arccosine
SINH	Sinh
COSH	Cosh
TANH	Tanh

The following procedures expect one argument, of long real type, and yield a long real type result. Note that they are formed by adding an L before the equivalent single precision procedure.

Procedure Name	Function
LSIN	Sine
LCOS	Cosine
LARCTAN	Arctangent
LSQRT	Square root
LEXP	Exponential
LLN	Logarithm (to base e)

The functions ENTIER, ABS and SIGN are also available, as described in Section 5.1.2.

NOTE

If arguments of type integer or long real are given in an ALGOL call to these procedures, the compiler plants the appropriate conversion code.

17.2 STRING PROCEDURES

For details of the procedures CONCAT, LENGTH, SIZE, COPY, NEWSTRING and DELETE, see Paragraph 13.7.

17.3 UTILITY PROCEDURES

17.3.1 Array Dimension Procedures

The integer procedure DIM, which takes as parameter the name of an array of any type, yields a result that is the number of dimensions of the array. This is most useful when the user passes an array as a parameter and wishes to check if it is, for example, a matrix.

The integer procedures LB and UB also take as first parameters the name of an array; the second parameter is the subscript number. The result is the lower or upper bound, respectively, of the subscript specified by the second parameter. The following procedure uses these to clear real matrices.

```
PROCEDURE ZERO(A); ARRAY A;
BEGIN
  INTEGER I,J;
  IF DIM(A) = 2 THEN
  BEGIN
    INTEGER L1,L2,U1,U2;
    L1 := LB(A,1); U1 := UB(A,1);
    L2 := LB(A,2); U2 := UB(A,2);
    FOR I := L1 UNTIL U1 DO
    FOR J := L2 UNTIL U2 DO A[I,J] := 0
  END
END
```

17.3.2 Minima and Maxima Procedures

The integer procedures IMIN and IMAX, the real procedures RMIN and RMAX, and the long real procedures LMIN and LMAX are used, respectively, to determine the minimum or maximum of a number of arguments of the appropriate type. These procedures normally accept up to ten parameters (this figure may be changed by re-assembling the ALGOL library with a different parameter).

For example:

```
I := IMIN(J,K);
X := RMAX(Y+Z,RMIN(Y-Z,Q));
```

17.3.3 Field Manipulations

The procedures GFIELD and SFIELD enable the user to manipulate a field within any integer, real, long real, Boolean or string variable. The integer parameters I and J specify a byte of length J bits whose leftmost bit is the I'th bit (counting from zero at the left-hand side). The byte specified may be from 1 to 36 in length and may be at any position in the variable.

For single word variables (integer, real, Boolean), I may range from 0 to 35, with the constraint I + J <=36. For double word variables (long real and string), I may range from 0 to 71, with the constraint I + J <=72.

The integer procedure GFIELD uses I and J as the second and third parameters; the first parameter is the variable. The result is the value of the byte (right justified) specified by I, J.

Thus

```
K := GFIELD(A,3,5);
```

gives the value of the byte consisting of bits 3 through 7 of A.

The procedure SFIELD sets a byte specified by the second and third parameters I, J to the value specified by the fourth parameter, of type integer. Thus

```
SFIELD(A,3,5,0);
```

zeros the byte specified in the first example.

17.4 DATA TRANSMISSION PROCEDURES

For details of these procedures refer to Chapter 16.

17.5 FORTRAN INTERFACE PROCEDURES

F-10 or F-40 FORTRAN subroutines may be incorporated in ALGOL object programs by loading these subroutines with the ALGOL main program (and any other separate ALGOL procedures).

Such FORTRAN subroutines should be specified by an EXTERNAL declaration in the ALGOL program and, depending on the FORTRAN compiler used, the appropriate procedures should be called.

Table 17-1
FORTRAN Interface Procedures

TYPE	NONTYPE	INTEGER	REAL	LONG REAL	BOOLEAN
FORTRAN				(DOUBLE PR.)	(LOGICAL)
F-10	F10CALL	F10ICALL	F10RCALL	F10DCALL	F10LCALL
F-40	CALL	ICALL	RCALL	DCALL	LCALL

The first parameter in these procedure calls must be the name of the FORTRAN subroutine, which must be declared as an external procedure of the appropriate type (or non-type). Subsequent parameters are taken as the arguments to the procedures.

CALL and F10CALL are used as single statements, for example:

```
CALL (FORT,X,Y)
```

is equivalent to

```
CALL FORT (X,Y)
```

in a FORTRAN program.

ICALL etc. must appear in the appropriate context in an expression, thus

```
P := Q + ICALL(Z)
```

NOTE

The parameter of CALL, ICALL, etc., maybe of any type, including arrays of any dimension, with the exception of string.

17.6 GENERAL INFORMATION ROUTINE

The integer procedure INFO depending on the value of the parameter specified, provides information about various aspects of the environment.

Parameter	Returns integer value of
0	core size in words
1	date (15-bit format)
2	time (ticks since midnight)
3	time (milliseconds since midnight)
4	runtime (milliseconds)
5	processor type (1=KA, 2=KI, 3=KL)
6	number of stack shifts so far
7	compiler version word

For example

```
PRINT(INFO(4));
```

might produce

```
1134600
```

- the job's runtime up to now in milliseconds.

17.7 DATE AND TIME IN ASCII FORMAT

Three routines are provided for returning the current time and date in string format suitable for printing without modification. The two date routines, FDATE and VDATE, give the option of a standard three-character abbreviation for the month (FDATE), or a variable-length string with the name of the month in full (VDATE). In both cases the year is given in full. String procedure TIME gives an eight-character string with the current time as HH:MM:SS.

For example

```
WRITE(VDATE);  NEWLINE;  WRITE(TIME);
```

might produce

```
27-JANUARY-1975
12:16:55
```

17.8 RANDOM NUMBER ROUTINE

Three routines have been included to provide a random number capability. The number generator is similar to that used in the FORTRAN library, which is documented in the Science Library manual. The ALGOL version is, however, initialized randomly. If a repeatable sequence of pseudo-random numbers is required then procedure SETRAN should be called before the first call to RAND, with the required initial value. For example, to generate the same sequence as a FORTRAN program using the default starting value currently used by FORTRAN, SETRAN(-1); should be included in the ALGOL program. The third procedure is SAVRAN which returns the value of the last random number without invoking the number generator.

RAND and SAVRAN are INTEGER procedures; SETRAN is non-type.

17.9 ONTRACE AND OFFTRACE

These two typeless parameterless procedures turn the dynamic tracing of procedure entry and labels on and off respectively. Neither the entry of trace items into the trace buffer (which is printed by use of the TRACE command to the REENTER dialogue) or the Dynamic Debugging System is affected by the procedures.

17.10 PAUSE

This typeless parameterless procedure merely exits to the Monitor, in such a way as to allow execution to be continued by typing the Monitor command CONTINUE. This is provided to allow, for example, a device to be assigned.

17.11 DUMP

This typeless procedure has one integer parameter: the number of block-levels containing the present one to be dumped. DUMP is identical to the Debugging System command DUMP (see section 20.7), with the following exceptions:

1. Output is directed to the currently selected output channel (ignores Debugging System REDIRECT commands)
2. Arrays are always dumped
3. A history trace (similar to the one produced when a run-time error is found) always precedes the dump.

The integer parameter performs the same function as the numeric argument to the DUMP command: a value of zero has the effect of "ALL".

CHAPTER 18

RUNNING AND DEBUGGING PROGRAMS

18.1 COMPILATION OF ALGOL PROGRAMS

DECsystem-10/20 ALGOL programs are compiled by the ALGOL compiler under the standard DECsystem-10 or DECsystem-20 timesharing monitor. The compiler is called by typing

```
R ALGOL
```

or (for the DECsystem-20)

```
ALGOL
```

at monitor command level.

The DECsystem-10/20 ALGOL Compiler responds by typing an asterisk on the user's terminal. The user then types a command string to the compiler, specifying the source file(s) from which the program is to be compiled, and the output files for listing and output of relocatable binary. The command string takes the form:

```
OUTPUT-FILE,LISTING-FILE=SOURCE-FILES
```

followed by a carriage-return (ALTMODE cannot be used to terminate a command string).

A file takes one of the forms

```
DEVICE:FILE-NAME.FILE-EXTENSION
```

or

```
DEVICE:FILE.NAME
```

for directory devices (disk and DECtape)

or

```
FILE-NAME.FILE-EXTENSION
```

or

```
FILE-NAME
```

where DSK is assumed to be a default device.

In the case of non-directory devices, the format is simply

```
DEVICE:
```

In cases where no FILE-EXTENSIONS are specified, the standard defaults REL for the relocatable binary output file, LST for the listing file, and ALG for the source file are assumed.

SOURCE-FILES

consist of one file or a list of files separated by commas. If a DEVICE is specified for the first file, and not for succeeding files, the second and following files are taken from the same device as the first.

Example:

EULER,TTY:=EULER

[read source from DSK:EULER.ALG, write relocatable binary on DSK:EULER.REL and listing on the user's terminal].

MTA0:,DSK:SIM26=SIM26,PARAM.TST

[read source from DSK:SIM26.ALG, DSK.PARAM.TST, write relocatable binary on device MTA0, and listing file on DSK:SIM26.LST].

Certain switches may be set by the user in the command string. These are:

BUFFERS:n	Set number of buffers in compiler's I/O buffer-ring to n.
CHECKON	Compile run-time array-bound checking (see 18.5.1.1).
CHECKOFF	Do not compile run-time array-bound checking, regardless of any CHECKON statements in the source.
HEAP:n	Set the initial size of the dynamic core area (which is used for I/O buffers, strings and OWN arrays) to n words. This area is dynamically expanded at run-time if necessary; its final size is typed out at the end of execution if the object program is loaded with DDT.
HELP	*Type helpful text.
KA10	Produce code to run on the KA10 processor.
KI10	Produce code to run on the KI10 processor.
KL10	Produce code to run on the KL10 processor.
LIST	*List the source program (default if a listing-device is specified in the command-string).
NOERRORS	Do not type error-messages on the terminal.
NOLIST	Do not list the source program.
NONUMBERS	The source program does not have line sequence numbers in columns 73 to 80 (default).
NOQUOTES	Delimiter words are not in quotes (default).

NOSYMBOL	Suppress output of expanded symbol table to the .REL file.
NUMBERS	The source program has line sequence numbers in columns 73 to 80.
PRODUCTION	Do not compile trace information or output expanded symbol-table to the .REL file.
QUOTED	*Delimiter words are in quotes.
SYMBOL	Include symbol table information in .REL file.
TEMPCODE:n	Set length of TEMPCODE area in compiler to n words: this is only necessary if the compiler produces a message to that effect, which may happen for some very complicated statement constructs.
TRACE	Control tracing.

NOTE

Since only one processor type, the KA20 is available for the DECsystem-20, no switch options are therefore available (unlike the DECsystem-10 which can run on one of the three processors, KA10, KI10 and KL10, and therefore has three options).

Switches may be shortened to a unique abbreviation: those marked with an asterisk (*) in the above list may also be given as a single character. Values (n) are in decimal.

Switches after a file-specification are set by a preceding /.

For example:

```
PROD,PROD=PROD1/L,PROD2/NOL/HEAP:2000
```

causes file PROD1 to be compiled with a listing, PROD2 to be compiled without listing, and the initial size of the run-time dynamic core area to be set to 2000 words (the default size is 521 words).

The ALGOL compiler reports all source program errors both on the user's terminal and in the listing device if it is other than the terminal. After compiling a program, the compiler returns with another asterisk, whereupon the user may compile another program, or type ^C to return to monitor level.

18.1.1 Compilation of Free-Standing Procedures

DECsystem-10/20 ALGOL allows the user to compile procedures independent of programs that call them. Such procedures may either follow the main program in the source file or may be in an independent source file either singly or together. The user uses exactly the same process to compile such files.

NOTE

Free-standing procedures must not appear before the main program. If the user requires to call those procedures from the main ALGOL program, the appropriate EXTERNAL declarations must be made (refer to Paragraph 11.9).

18.2 LOADING ALGOL PROGRAMS

ALGOL programs are loaded by means of the DECsystem-10/20 or DECsystem-20 Linking Loader in exactly the same way as programs generated by MACRO-10/20 and FORTRAN are loaded (for details, refer to the DECsystem-10/20 Assembly Language Handbook).

LINK-10 or LINK-20 automatically causes all procedures required from the ALGOL Library (ALGLIB) to be incorporated into the user's program.

For example, consider the source file MAIN.ALG which contains the ALGOL main program and the files SUB1.ALG and SUB2.ALG which contain free-standing procedures.

The user may compile these files to give one relocatable binary file by typing the following command string to the ALGOL compiler,

```
MAIN,MAIN=MAIN,SUB1,SUB2
```

and loading the resulting program by giving the command string

```
MAIN/GO
```

to LINK-10 or LINK-20. Alternatively, the three source files can be compiled independently by typing three command strings to the ALGOL compiler, for example:

```
MAIN,MAIN=MAIN

SUB1,SUB1=SUB1

SUB2,SUB2=SUB2
```

and giving LINK-10 or LINK-20 the command string.

```
MAIN,SUB1,SUB2/GO
```

After a program has been loaded, it may be executed.

18.3 RUNNING ALGOL PROGRAMS

ALGOL programs are executed by typing the console command

```
START
```

or any of its valid abbreviations. If the program executes successfully, a message will be printed on the user's terminal, and the program will return to monitor command level.

18.4 CONCISE COMMAND LANGUAGE

The Concise Command Language (CCL) features in the DECsystem-10 or DECsystem-20 monitor may be used to facilitate the compilation and execution of ALGOL programs. These features are used in exactly the same way as for programs written in DECsystem-10 or DECsystem-20 FORTRAN. For details, refer to the DECsystem-10 or DECsystem-20 Users Handbook.

Switches to the ALGOL compiler are enclosed in parentheses and separated by slashes. For example:

```
EXECUTE FOO(QUOTED/HEAP:2000/KI)
```

18.5 RUN-TIME DIAGNOSTICS AND DEBUGGING

If a run-time error occurs during the execution of an ALGOL object program, an error message is produced, detailing the type of error, and its address within the user's program. Such errors fall into two categories - fatal and non-fatal.

A mechanism has been provided by which DECsystem-10 users can trap non-fatal errors, and when they occur, transfer control to a label within the user's program. Each such error has a unique number, and a table of these appears below. The Library procedure TRAP, used to trap non-fatal errors has the following specification:

```
PROCEDURE TRAP (N,L);  VALUE N,L;

  INTEGER N;  LABEL L;
```

Where N is the number of the error to be trapped, and L is a label to which control is required to be passed when the error occurs.

Once such a trap is set up by a call to TRAP, it remains in force until another call to TRAP sets a trap to a different label, or until the trap is turned off by

```
TRAP (N)
```

that is, omitting the label parameter in a trap call.

NOTE

1. TRAP is available only on DECsystem-10.

2. The trap label is a formal parameter by value. The number of the trap that caused the jump to the label may be obtained by calling the integer procedure TRAPNO.

Table 18-1
Error Trap Numbers

TRAP NO.	ERROR
18	FLOATING POINT OVERFLOW
19	FIXED POINT OVERFLOW
32	INPUT OR OUTPUT DEVICE UNAVAILABLE
33	ILLEGAL MODE FOR INPUT OR OUTPUT DEVICE
34	INPUT OR OUTPUT ON UNDEFINED CHANNEL
35	ATTEMPT TO READ OR WRITE ON DIRECTORY DEVICE WITHOUT FILE OPEN
37	FILE NOT AVAILABLE OR RENAME FAILURE
38	ATTEMPT TO READ OR WRITE OVER END-OF-FILE
39	ERROR CONDITION ON INPUT OR OUTPUT
40	ILLEGAL CHARACTER IN NUMERIC DATA
41	OVERFLOW IN NUMERIC DATA
42	ERROR CONDITION ON CLOSING FILE
43	ILLEGAL INPUT/OUTPUT OPERATION
44	I/O CHANNEL NUMBER OUT OF RANGE
48	SQRT ARGUMENT NEGATIVE
49	LN ARGUMENT ZERO OR NEGATIVE
50	EXP ARGUMENT TOO LARGE
51	INVERSE MATHS FUNCTION ARGUMENT OUT OF RANGE
52	TAN ARGUMENT TOO LARGE

18.5.1 **Facilities to Aid in Program Debugging**

18.5.1.1 **Array Bound Checking** - The directive

```
CHECKON
```

when placed anywhere in a user's program causes all array subscripts from this point onward in the program to be checked at run-time for being in range. The directive

```
CHECKOFF
```

nullifies this action.

The compiler switches /CHECKON and /CHECKOFF may also be used: they override any CHECKON or CHECKOFF statements in the source program.

NOTE

1. The CHECKON, CHECKOFF facility causes the generated program to be slightly larger, and to run slower.

2. Most inexplicable errors arising during the execution of an ALGOL program are caused by an array subscript being out of range. Whenever such errors occur, the program should be recompiled with the array bound check feature on, and re-run.

18.5.1.2 **Controlling Listing of the Source Program** - Normally, a listing of the source program is output with the object program during compilation. The user can suppress this listing entirely by means of the /NOLIST compiler switch. However, if the user wishes to suppress only part of the listing and then continue listing, he can control the listing from within his program by means of the statements

```
LISTOFF
LISTON
```

The LISTOFF statement causes listing to be suppressed from the point in the program where LISTOFF was encountered to either the end of the program or until a LISTON statement is encountered. The LISTON statement causes listing to continue after it had been suppressed by a LISTOFF statement. The LISTON and LISTOFF statements have no effect if the /NOLIST switch is included in the compiler command string.

18.5.1.3 **Setting Line Numbers in Listings** - Ordinarily, the lines in the listing file are numbered sequentially starting at 1 and incrementing by 1. The user can, however, change the line numbers by placing sequence numbers in columns 73 through 80 of the source program and compiling with the /NUMBERS switch. Another way in which the user can change the line numbers is by means of the LINE statement. The statement

```
LINE n
```

causes the next line number to be set to n, which is a decimal integer. The line numbers that follow are incremented by 1 until either another LINE statement is encountered or the program terminates.

18.6 CROSS REFERENCE LISTING

The /CREF switch has been implemented, causing ALGOL to generate output for CREF. Output takes the following formats.

Variables and Labels: Each occurrence of a variable or label name is recorded, with a # whenever a defining reference is made. In the case of labels the line reference is to the line which causes code for the label to be generated. This may follow the line where the label appears. No distinction is made amongst different incarnations of a variable at various block levels.

Blocks: The messages "START OF BLOCK n" and "END OF BLOCK n" are suppressed in the CREF listing. However, there is a separate CREF of blocks on the first page following the program. As with labels, the line-numbers refer to those causing code to be generated, not necessarily those on which BEGIN or END appear in the source.

18.7 STACK ANALYSIS

The stack analysis takes two forms - if a program stops with an error, the stack is scanned to give the names of the active procedures. Thus a typical error message now reads:

```
?RUN-TIME ERROR AT ADDRESS 000162


In procedure OPENFILE
Called from procedure PQRSTUVWXYZ
Called from procedure THISPROCEDUREISNEXTTOINNERMOST
Called from procedure THISONEISNEARLYTHEOUTERMOST
Called from procedure THISISTHEOUTERMOSTPROCEDURE
Called from MAIN PROGRAM
File DSK:NOSUCH.FLE not available or rename failure on channel # 1
```

The above type of analysis is automatic whenever an error is detected. The second type is invoked by the Debugging System command PROFILE. This causes a list of all the procedures, labels and library procedures referenced in the program to be printed, together with a count of the actual number of times each was referenced (or passed through in the case of labels). As with the trace print, labels can be distinguished by a trailing colon (:) and library procedures by a trailing asterisk (*), but different procedures with the same name must be identified from the order in which they appear, which is the same as the loading order.

In the special case of overlaid programs only the root segment is scanned, for example:

```
>>PROF

PROFILE PRINT.


COUNT           NAME

 1              PQRSTUVWXYZ

 1              THISPROCEDUREISNEXTTOINNERMOST
```

```
1                THISONEISNEARLYTHEOUTERMOST

1                THISISTHEOUTERMOSTPROCEDURE

0                READ*

1                INPUT*

0                OUTPUT*

1                OPENFILE*
```

18.8 TRACE

18.8.1 Dynamic Trace

Two types of Trace are available: dynamic and post-mortem.

The user who is at a terminal and wishes to see a full dynamic trace of his program should type the following command sequence:

```
.LOAD TRTST
 ---- -----
ALGOL:  TRTST
LINK:   LOADING

EXIT

.REE
 ---

ALGOL DIAGNOSTIC SYSTEM

FACILITIES (H FOR HELP)?

>>ONTRACE


>>START
```

This produces (at least) one line for each trace reference, as follows:

```
******SECTION
****S6:
********PHASE
************WRITE* PHASE
************PRINT* 6
************WRITE* :    BEGIN
********PHASE
************WRITE* END

******SECTION
****S7:
********PHASE
************WRITE* PHASE
************PRINT*  7
************WRITE* :    BEGIN
********PARFOR
********PHASE
************WRITE* END
```

Dynamic trace output always clears the terminal output buffer and begins a new line. The current dynamic block level is represented by two asterisks for each level, printed along the line. Note that a notional block surrounds the main program and each procedure. A maximum of sixty asterisks are printed before the line is folded: folding is shown by a number in the first two columns representing multiples of thirty dynamic block levels.

The name of the procedure or label appears at the end of the line of asterisks. As with the profile, library procedures are distinguished by a trailing asterisk, and labels by a trailing colon.

Procedures are traced at entry but not exit; however, a procedure exit can be inferred from the block levels.

NOTE

Ordinary TTY output in Dynamic Trace mode may be interspersed amongst Trace output (as in the example above). Dynamic trace may also be turned on and off dynamically by use of the library procedures ONTRACE and OFFTRACE.

18.8.2 Post-Mortem Trace

A post-mortem trace will be printed automatically if a batch job gives a run-time error. Under timesharing, after typing out the location and type of error, and the stack analysis, the Debugging System offers the user various options. The TRACE command produces a trace similar to the one described under Dynamic Trace above, but with spaces instead of asterisks to represent block levels, and without, any other TTY output. The post-mortem trace entries are maintained in a circular buffer in the Heap. The default length of the buffer is 100 (decimal) but this can be altered by using the /TRACE switch to the compiler, with an appropriate value. Tracing is the default option: users can compile their programs without trace information by using the /PRODUCTION switch to the compiler. However, the Trace entry mechanism is quite efficient, and users are urged not to exclude it, as this also has the effect of making the stack analysis less useful. Library procedures are always traced.

18.9 PERFORMANCE ANALYSIS

Various features are provided which are designed to help users wishing to evaluate the performance of their programs.

18.9.1 Heap Space

DECsystem-10/20 ALGOL is very flexible in its use of memory: space for arrays, strings, I/O buffers and so on is allocated in an area called the Heap, which lies between the program code and the stack. The default initial Heap size is 521 (decimal) words. If a program needs more Heap than is currently available, the object-time system moves the stack up in memory to make more room, obtaining more core from the Monitor as necessary (subject to over-riding constraints applied by the system).

For most programs this provides a reasonable balance between core use and computation. However, the stack shifting mechanism can be very expensive if used often and for small expansions. The user can find out how the Heap is being used by typing the STATISTICS command (see Chapter 20, section 20.8.6) to the Debugging System after the program has executed (or simply typing CONTINUE immediately after the End of execution message). This produces output of the following format:

```
EXECUTION TIME:
ELAPSED TIME:
MAXIMUM HEAP SIZE:    ,# OF STACK SHIFTS:
MAXIMUM # OF USED WORDS IN THE HEAP TABLE:
```

(The last item is a measure of the fragmentation of the free space in the heap and is only produced if the optional Heap Integrity Checker is present in the object-time system, i.e., when assembly switch FTGETCHK is turned on.)

If the number of stack shifts is larger than, say, ten, an improvement in performance could be expected from re-compiling the program with the /HEAP switch value set to the maximum heap size given by the Statistics type-out.

18.9.2 **Code Utilization**

By judicious placement of labels and subsequent printing of the program's profile, the frequency with which a program executes particular sections of code can be monitored. The library procedure INFO can be used to obtain more detailed timing information.

CHAPTER 19

TECHNICAL NOTES

These notes concern the authors' particular interpretation of the "Revised Report on the Algorithmic Language ALGOL-60" and its implementation.

1. At all times, strict left-to-right evaluation of statements is employed. Section 3.4.6 of the Revised Report has been construed by some experts to mean that left-to-right evaluation of expressions is not required. However, there are undoubtedly many ALGOL-60 programs in existence that rely on this feature.

2. Section 4.3.5 of the Revised Report requires that a GOTO Statement with a designational expression which is a switch with a subscript out of range be regarded as a dummy statement. Neither DECsystem-10/20 ALGOL nor any other ALGOL-60 implementations, to the knowledge of the authors, follow this rule; there is a side-effect involved in the evaluation of the subscript.

CHAPTER 20

THE ALGOL DYNAMIC DEBUGGING SYSTEM

20.1 SUMMARY OF FEATURES

In ALGDDT, the ALGOL programmer has facilities to

1. Interrupt program execution
2. Set pauses and clear them
3. Examine and alter ALGOL variables in scope
4. Examine various system parameters
5. Automatically type ALGOL variables after a pause
6. Continue program execution from where it halted
7. Continue program execution from an appropriate label

20.2 GENERAL REMARKS

20.2.1 Symbol (.SYM) File

ALGDDT requires access at runtime to a file called Programname.SYM that is produced by LINK. If it cannot access the file (perhaps because it has another name, or resides under another PPN) an opportunity is given to supply its name, etc. ALGDDT also needs a free I/O channel: if there is none, it will ask the user to nominate one for it to release and use.

20.2.2 Entering ALGDDT

To enter ALGDDT from a terminal running a ALGOL program, type ^C. When this is detected, the ALGOL Dynamic Debugger will attempt to complete executing the current ALGOL statement before passing control to ALGDDT. Program execution will be halted and, unless ALGDDT was unable to find a .SYM file, the following message will be typed:

"Stopped at line nnn [,statement n]"

After a run-time error, ALGDDT will be entered (except in a batch job).

However, there are cases where the current statement never completes.

For example, if ^C is typed during the execution of a statement which reads data from the terminal, no data will be forthcoming as the user will be waiting at the other end for the system to enter ALGDDT mode. To overcome this, a second ^C is required.

ALGDDT may also be entered before execution of a program by use of the REEnter facility. The program can then be started with the START command. In this case, the effective "current statement", for determining scope of identifiers, is immediately before the first BEGIN of the main program.

20.2.3 ALGDDT Command Format

The syntax of the ALGDDT commands has been designed to resemble ALGOL commands; all of which are terminated by a semicolon (or carriage-return), and the lists associated with AUTO commands are bracketed by a BEGIN-END pair. However the resemblance is superficial, and for the most part only simple commands can be given in a single "statement". When the debugger is ready to accept a command, ">>" is prompted. All commands and options may be shortened to a unique abbreviation (except where noted). Blanks and tabs are ignored between elements of a command, as are "readability symbols" (periods) in ALGOL identifiers. A command may be continued on another line by typing a control-backarrow or control-underline. Comments may be introduced by a preceding ! and the rest of the line will be ignored.

20.2.4 Line Numbers

Several ALGDDT commands require line numbers. The compiler only puts line numbers into the symbol table accessed by ALGDDT for those lines where code is generated (including BEGIN and END statements of a block, but not for blank lines, declarations, etc.). If the user gives an "unknown" line number, ALGDDT will scan forward to the next known line number and use that; this usually has the desired effect.

In a program LINKed from more than one .REL file ("module"), the user can qualify a line number with a module-name preceded by "IN":

```
>> PAUSE 27 IN FOO
```

The module-name is the name of the external procedure (truncated to six characters), or the main program-name (.REL-file name) for the main program. If the module-name is not specified, the default is the current module (that in which execution was stopped by PAUSE, ^C or an error, or the main program before execution is started). Although, a program can have more than one module with the same name, ALGDDT will use the one that is also the main program, if any; otherwise an error message will be typed.

Additionally, in the OBJECT command, an octal address may be specified by a preceding number-sign (#).

20.3 TYPEOUT COMMANDS

The command TYPE is used to display the value of an ALGOL variable or array element on the terminal. The keyword is followed by a list of one or more variable names, separated by commas. The whole list is terminated by a semicolon. If more than one name is given, then all the names are repeated by the debugger.

Examples:

```
>> TYPE REAL.VARIABLE;       1.2345
>> TYPE REAL.VARIABLE, INTEGER.VARIABLE;
REAL.VARIABLE = 1.2345
INTEGER.VARIABLE = 12345
>>
```

20.3.1 String Typeout

When string typeout is requested, the length and decimal byte size of the string are output first, in parentheses, then

1. In the case of ASCII or SIXBIT strings, the character representation of the string is typed in quotes,

   example:

   ```
   >> TYPE STRING.VAR;    (10,7) "A string<CR><LF>"
   >> TYPE SIXBIT.VAR;    (15,6) "A SIXBIT STRING"
   ```

2. For strings with a byte size other than six or seven, an octal representation of the byte values, each separated by a comma, is typed.

   example:

   ```
   >> TYPE NONASCIISTRING;  (8,5)
   10,37,17,17,22,1,10,0
   ```

Extra bits at the ends of words are ignored.

20.3.2 Array Typeout

If the TYPE command is used with an array name, then the entire contents of an array is typed. The format of the typeout will correspond to the structure of the array:

```
>> TYPE ARRAYNAME;

[0,0]   1.7   2.3   5.4   1.32
[1,0]   2.3   1.7   5.4   8.1
[2,0]   5.4   ....  ETC.
```

The contents of a single location may be typed by specifying the location of the element:

```
>> TYPE ELEMENT[1,3,5,200];  0.0
```

NOTE

ALGOL variables may not be used as subscripts. All subscripts must be simple constants. Array bound checking is imposed on ALGDDT commands where appropriate.

Rectangular portions of arrays may be selectively typed by specifying the relevant ranges:

```
>> TYPE ARRAYNAME [0:1,2:3];

[0,2]   5.4   1.32
[1,2]   6.8   8.1
```

The whole of a particular dimension can be represented by an asterisk, e.g., A[1:20,*,1:5].

NOTE

There must be a specification for every dimension of a multi-dimensional array.

20.3.3 Displaying Current Array Dimensions

The current dimensions of an array can be displayed by using the DIMENSION command:

```
>>DIM A;  [0:17,1:5]
```

20.3.4 Typeout of Boolean Variables

Boolean variables (or arrays) can be typed using the above commands. Since FALSE is represented by zero, zero elements or variables will be displayed as 'FALSE'; non-zero variables having values of -1 (the value used in directly generated assignments), are either typed as 'TRUE' or octal for other values.

For example:

```
>> TYPE B1,B2;
B1= False
B2= 012345671234(True)
```

20.3.5 General Points On Typeout

All ALGOL variables in scope will be available for typeout, including formals, but the possibility of side-effects should be borne in mind. The only exception is a variable that is declared but never referenced in the program. For internal reasons, the debugger cannot access such variables.

Long, unwanted type-outs (of arrays, etc.) may be aborted by typing two ^Cs. Occasionally this may return control to the monitor, in which case CONTINUE or REENTER may be typed to return to ALGDDT.

20.3.6 Typeout of Object Code

The user may request a typeout of the Macro code for the current ALGOL statement, using the command

```
>> OBJECT
```

After a PAUSE this types out the code for the statement that will be executed next. After a run-time error, the code generated for the current statement is typed out instead and the instruction where the error occurred is marked with an asterisk. The user can examine other statements by using the OBJECT command with the relevant line number (and statement number if necessary). For example:

```
>> OBJECT 22;
```

or an octal address:

```
>> OBJECT #275;
```

In this case if no "IN module" is specified, the address is absolute; thus the following would dump the accumulators:

```
>> OBJECT (16) #0;
```

the decimal number of ALGOL statements (words if an octal address rather than a line number) to be output may be specified by typing a number in parentheses:

```
>> OBJECT (17) 22;
```

or

```
>> OBJECT (13);
```

the typeout is normally in octal, and symbolic instructions. Other formats may be specified by using the MODES option:

```
>> OBJECT line-number MODES a,a,a...;
```

where the a's are chosen from:

7 or A	ASCII
6	SIXBIT
S	Symbolic Instructions
I or D	Decimal Integer
O	Octal
F or R	Real Number
L	Long Real (each word and the next).

Line-number may be qualified by IN module-name if the program has external procedures. A complete example is therefore:

```
>> OBJECT (18) 27,3 IN EXTPRG MODES A,6,S;
```

This will dump 18 ALGOL statements starting at the fourth statement on line 27 in the external procedure EXTPRG. The dump will be in ASCII and SIXBIT characters, and symbolic instructions.

20.3.7 System Parameters

ALGDDT maintains a table of certain symbolic variables used in the system. The contents of these system variables distinguished by a preceding % can be displayed in the usual way. Most of the system variables, as defined in ALGPRM, will be available. A few are listed below as examples.

%DB	current data-base register
%SP	stack pointer
%CONDL	context "DL"
%DL	pointer to current display
%VERSHN	version # word of compiler used

ALGDDT does not permit alteration of system parameters.

20.4 CHANGING ALGOL VARIABLES

The values of an ALGOL variable may be altered by using a simple assignment statement:

```
>> INTVARIABLE:= 1235;
```

Type conversion will take place, but only a single constant is permitted on the right-hand side of the assignment:

```
>> BOOL :=TRUE;
>> BOOL := 003007;
>> INT:=1;
>> INT :=1.0;
>> REAL:=1.2345&7;
>> A[1,3]:=18.4;
>> S[1,7].[3] := 64;
```

Note, however, that the following are not legal:

```
>> A:=B;
>> C[I,J]:= 20
>> D[1,6].[I] := 66;
```

Since strings are essentially dynamic in nature, some extra rules apply. Unless a new byte size and/or length are specified in the type-in, then the current values are used. Type-in consists of octal bytes separated by commas. If the numbers typed are too large for the byte-size in force, then they are truncated, and a warning message is issued. A string enclosed by single or double quotes is also allowed if the byte-size is six or seven respectively. A quote may be entered by typing two quotes. All characters, including carriage returns, ALT-modes, semicolons, !s, and so on, are entered exactly as typed. The only exceptions are control-backarrow, which acts as usual; and a quote, which terminates the string. If more bytes than the specified length are typed, then the length is extended. If fewer, then the extra bytes are zeroed. The string may be continued on another line by typing control-underline.

NOTE

Square brackets have a special meaning in strings that are to be output (see Section 16.6.2).

20.5 PAUSES

By setting PAUSES, the user may cause program execution to be interrupted automatically when specific points are reached.

20.5.1 Setting PAUSEs

Pauses may be applied before any executable ALGOL statement by the ALGDDT command:

```
>> PAUSE line no [,statement no] [IN module-name];
```

or

```
>> PAUSE label:  [IN module-name];
```

or

```
>> PAUSE PROCEDURE name [IN module-name];
```

When this point is reached the message

"Pause at line nnnn [,statement n] in module name"

is typed.

The pause remains in effect, so that subsequent activations of this piece of code will also cause this message to be printed. Note that the portions of the command in brackets are optional. If the statement number is absent, then the first (or only) statement beginning on that line is assumed. If module name is absent, the current module is assumed; the main program is the current module before program execution starts. If the statement specified does not exist, or has no generated code, the PAUSE is placed on the next suitable statement, if any. Labels and procedures must, like all other identifiers, be in scope.

There is no restriction on the number of pauses a user can establish.

If a number (n) is typed in parentheses after a PAUSE instruction, then the resultant pause will be bypassed n-1 times before program execution is halted.

An upper limit may also be specified for a PAUSE in the form of

```
>> PAUSE (n:m);
```

This will cause the break to be taken from the n'th time up to and including the m'th time that the program reaches the breakpoint location. On completion of the break action after the m'th pass, PAUSE will be killed automatically.

PAUSE; with no line number or label specified, sets a PAUSE at the current statement (after a ^C has been typed).

20.5.2 Resetting Of PAUSE Proceed Count

When a pause is established using the ALGDDT command:

```
>> PAUSE (n);
```

or

```
>> PAUSE (n:nl);
```

This pause will occur on the n'th time after the program reaches the breakpoint location. Once the break has occurred, it will repeat each time the program reaches this point. To stop subsequent unnecessary pauses, the count can be reset by specifying the command

```
>> CONTINUE m;
```

This will result in the pause to be taken only on the m'th subsequent pass and the m'th subsequent pass after that and so on until the PAUSE is KILLed.

NOTE

> This command may be given as a direct ALGDDT command, or in an AUTO-list.

20.5.3 Clearing PAUSEs

A KILL command is provided for clearing pauses. Three possible varieties are available. After a Pause,

```
>> KILL;
```

kills the current pause.

```
>> KILL line-number [,statement-number] [IN module];
```

or

```
>> KILL label:;
```

or

```
>> KILL PROCEDURE procname;
   (at least PR must be typed)
```

kills the specific pause referred to, or results in the message "Pause specified does not exist."

```
>> KILL ALL;  ! ALL must be typed in full;
```

kills all currently set pauses.

NOTE

> If there are PAUSEs set on two or more labels or procedures with the same name in the same module, a block-number (the one given in listings in the START OF BLOCK n messages, or in the block CREF) must be given, thus:

```
>> KILL LABEL:  3 [IN module];  !  3 is block-number;
```

thereby resolving the ambiguity.

```
>> KILL A;  !kills AUTO-list A;
```

20.5.4 Listing PAUSEs

The currently set PAUSEs, and their AUTO-lists, may be listed by use of the LIST command, which has the following formats:

```
>> LIST;
```

lists all PAUSEs and the names of all DEFINEd AUTO-lists.

```
>> LIST line-number[,statement-number];
```

or

```
>> LIST label:;  !  etc.;
```

lists the PAUSE specified together with its AUTO-list, if any.

```
>> LIST A;
```

lists AUTO-list A.

```
>> LIST ALL;!  ALL must be typed in full;
```

lists all PAUSEs and all DEFINEd AUTO-lists.

For example:

```
>> LIST
Proceed-count    Autolist    Where
19               Private     27,3 in FOO
0                A           LABEL1: in MUMBLE
0                            293 in FOO
88                           PROC1 in FOO

Defined Autolists: A,C-E,J,L,S-Q

>> LIST 27,3;
Proceed count = 19, private Autolist:
DIM A: !Type the dimensions of array A;
TYPE I,J,K;
CONTINUE;
>>
```

See note in Section 20.5.2 about ambiguous labels and procedure-names.

20.5.5 Automatic Execution of Commands After PAUSE

Any pause can have an associated AUTO-list, that is a list of commands that are executed whenever the pause is reached. AUTO-lists can contain almost any ALGDDT commands. These may either be typed in immediately after declaring a PAUSE, or they may be identified by a single letter and referred to indirectly. The following two examples would have the same effect:

```
(direct)
     >> PAUSE 17 BEGIN;
     >>      TYPE I, J, K;
     >>      END;
(indirect)
     >> DEFINE B;
     >>      TYPE I, J, K;
     >>      END;
     >> PAUSE 17 AUTO B;
```

NOTE

The prompt is followed by a Tab when reading an AUTO-list in either mode.

The advantage of the indirect method is that the same list can be referred to from different pauses. If the direct method is used, the AUTO-list is destroyed when the controlling pause is KILLed. ALGDDT does not check AUTO-lists for consistency, thus

```
>> PAUSE 17 BEGIN ;  TYPE A;  KILL;  END
```

would only be obeyed once, since the KILL instruction would kill both pause and list. AUTO-lists are, however, checked for syntax (legality of commands, etc.), but not for semantics (scope of identifiers, etc.) as this depends on the context of the PAUSE from which they are invoked. AUTO-lists are terminated by an END and elements in the list are separated by semicolons or carriage returns.

The action on detecting an error during execution of an AUTO-list can be controlled by means of a switch on the AUTO or BEGIN keyword. However, this switch only controls the action for invocation of this AUTO-list from the PAUSE.

IGNORE	No error message, continue to next element of AUTO-list.
CONTINUE	Type error message, continue to next element of AUTO-list.
KILL	Type error message, KILL reference to this AUTO-list from this PAUSE. If this is a "private" AUTO-list (defined by BEGIN) it itself is KILLed.
STOP	Type error message, go to debugger command level (default).

It is sometimes useful to suppress the "Pause at..." message when a PAUSE is reached (especially if the associatied AUTO-list ends with a CONTINUE command); this may be done by using the /SILENT switch on the AUTO or BEGIN keyword. Note, however, that error messages will still be typed unless /IGNORE is also used. For example, suppose that

a program was failing because a variable (I) was not initialized at the start of a loop (on line 25); the following would be a temporary (and inefficient!) cure:

```
>> PAUSE 25 BEGIN/SILENT;
>>       I:=0;
>>       CONTINUE;
>>       END;
>> START;
```

20.5.6 DEFINE Command

This command is provided to enable the user to define an AUTO-list for use in a subsequent PAUSE or AUTO command. The format is:

```
>> DEFINE A !Or any valid AUTO-list name;
>>       (ALGDDT commands)
>>       END;
>>
```

20.5.7 EXTEND Command

EXTEND enables additional commands to be appended to a previously DEFINEd AUTO-list. The format is:

```
>> EXTEND A;
>>         (ALGDDT commands)
>>         END;
>>
```

This command can also be used to define an AUTO-list, and can therefore be used instead of the DEFINE command.

NOTE

If an AUTO-list ending with a CONTINUE command is EXTENDed, the new commands will not be executed. No error will be printed.

20.5.8 AUTO Command

A command is provided to invoke a DEFINEd AUTO-list directly. The format is, for example:

```
>> AUTO A;
```

NOTE

AUTO may appear within another AUTO-list. Up to 26 (decimal) levels of nesting of this kind are permitted.

20.6 EXECUTE COMMANDS

20.6.1 CONTINUE

The simplest execute command is CONTINUE. This continues normal execution of the program until it terminates, meets a pause, or the user types ^C.

20.6.2 GOTO

GOTO (or GO TO) can take two forms,

```
>> GO TO LABEL:;
```

or

```
>> GO TO line number [,statement number];
```

In either case, the command is only valid if the destination is within scope, otherwise the message

"Identifier does not exist, or is out of scope"

is printed. No ALGOL code is executed, and the status of the program is unchanged. Unless a break-point has been set to the destination, program execution will proceed as if CONTINUE had been specified.

Formals may be specified as labels.

20.6.3 START

If the debugging system was entered by way of the initial REEnter sequence, program execution may be started by using the START command.

20.6.4 RETRY Command

This command enables an ALGOL program to be entered and executed from the beginning of the statement in which an error has occurred.

NOTE

RETRY differs from CONTINUE as the latter continues program execution from the point within the statement where the error has occurred.

20.6.5 NEXT

The NEXT command allows the user to step through a program statement by statement. Typing NEXT has the same effect as setting a PAUSE on the "next" statement and then typing CONTINUE (except that the PAUSE kills itself when reached). Even in cases where the current statement transfers control to any point other than the next sequential statement (either explicitly by a GOTO or implicitly as part of a conditional IF or iterative FOR/WHILE statement), NEXT remains in operation until the KILL command is issued.

NOTE

If an ALGOL statement includes embedded assignments, extra line numbers may be generated. This will cause NEXT to stop after completing the assignment rather than at the end of the statement. A second NEXT command will rectify this and enable pauses to be taken between all subsequent statements.

20.7 DUMP

The DUMP command may be used to output the values of all currently active variables. The output of arrays may be suppressed by using the SCALARS keyword. This command has an optional parameter. If this is absent, then only the variables declared in the current block will be dumped. If a numeric parameter, n, is specified, then all variables declared in the n enclosing blocks will also be dumped. If "DUMP ALL" is specified, then the variables declared in all the currently active blocks will be dumped.

NOTE

DUMP is static in nature, that is, any variables enclosed in blocks in recursively-activated procedures will only have the value of the "latest" occurrence dumped.

20.7.1 REDIRECT Device:filename.ext [proj,prog]

This command causes all ALGDDT output from DUMP commands to be directed to the device designated. If the device is a disk and the filename is specified, then the resultant output is appended to that file.

The command REDIRECT with no arguments causes DUMP output to be directed to the TTY.

NOTE

The FINISH command should be used instead of ^C to exit from ALGDDT when a REDIRECT is in force, else the output file will not be closed.

20.8 MISCELLANEOUS COMMANDS

20.8.1 Accessing "hidden" Variables - UNWIND and BACK

In many cases, there are variables that cannot be accessed because they are "hidden" in the current context, either because there is a variable of the same name in an inner block or in cases of recursion. The UNWIND command is provided to allow access (for display or change) to these variables. It refers to the dynamic block levels that are typed in the history trace (produced on error or ^C, or by the WHERE command). It has three formats:

```
>> UNWIND n;
```

moves ALGDDT's context for variable accessing to level n;

```
>> UNWIND -n;
```

moves the context n levels "out" from the true context (that at which program execution was stopped);

```
>> UNWIND 0;
```

moves the context out to the outermost block.

Also, the command

```
>> UNWIND;
```

or

```
>> BACK;
```

returns the context to the true context.

20.8.2 EXPERT and NOVICE

All error messages have a full and an abbreviated form. Normally the full form is typed, but the EXPERT command causes the short form to be used. The NOVICE command causes ALGDDT to revert to the full form. In addition, the user may type a ? after a short message, and the full message will be typed.

The EXPERT mode also prevents ALGDDT from typing the procedure history (after ^C or error) which can be obtained by using the WHERE command.

If a line

```
ALGDDT/EXPERT
```

appears in the file SWITCH.INI the user's area, ALGDDT will be entered with EXPERT mode in force (until changed by a NOVICE command).

20.8.3 WHERE

The WHERE command causes the system to retype the stack trace that was produced when the error (if any) occurred. This is intended to be of particular use for users of visual display terminals:

```
>> Where
On line 5 in module 8
In procedure PROC1 (level 2)
```

```
Called from line 12 in procedure PROX (level 1)
Called from line 16 in main program
>>
```

(The "levels" are for use in the UNWIND command, see section 20.8.1.)

20.8.4 TRACE

The TRACE command causes ALGDDT to type the contents of the trace buffer. This consists of the names of the most recent labels and procedures encountered, with the most recent first. (The typeout may as usual be aborted by typing two ^Cs, or a ^O.) The number of entries in the buffer is 100 (decimal) by default. This value may be changed by giving the /TRACE switch to the compiler. Labels are distinguished by a : and library procedures by a *. The indentation of the names gives the dynamic block level (two spaces per level and each procedure is enclosed by an extra notational level):

```
>> TRACE

!ALGOL postmortem trace (latest first)

     PRINT*
   LABEL1:
   LABEL2:
   LABEL1:
       FOO
           OPENFILE*
   LABEL1:
 MAIN.PROGRAM

>>
```

20.8.5 PROFILE

This command types the history of the program in terms of the number of times each label and procedure has been encountered; typeout is in the same order as the occurrence of the objects in the program. As for TRACE, labels are marked by a : and library procedures by a *.

For example:

```
>> PROFILE

Profile print

Count   name


3       LABEL1:
1       LABEL2:
0       LABEL3:
1       FOO
0       BAR
1       PRINT*
1       OPENFILE*
0       SELECTOUTPUT*

>>
```

20.8.6 STATISTICS

This command types the execution time, elapsed time, and core size of the program.

20.8.7 FINISH

This command causes the system to exit to the Monitor, first closing all files and releasing all devices.

NOTE

This must be used, rather than ^C, if a REDIRECT is in force to avoid losing the output from DUMP commands.

20.8.8 ONTRACE and OFFTRACE

These two commands are provided to control the dynamic trace facility (see Chapter 18). Note in this connection that tracing is suppressed while executing ALGDDT commands (this is significant as accessing a formal by name may cause execution of pieces of code that would otherwise be traced).

20.8.9 HELP

A HELP command is provided which types the contents of file SYS:ALGDDT.HLP or HLP:ALGDDT.HLP.

20.8.10 SOURCE

This command types any ASCII file. The format is:

```
>> SOURCE device:filename.ext[proj,prog];
```

The defaults are : DSK:ALGDDT.ALG; PROJ, PROG, or both may be omitted when those of the user will be used. S.F.D.s are not accepted.

20.8.11 Indirect Command Files

In response to a prompt from ALGDDT, the user may use an indirect command file by specifying the filename with a preceding @. This will enable commands to be read from that file. If no filename is specified then DSK:ALGDDT.CMD will be taken as default. This feature is particularly useful for inputting frequently used AUTO-lists.

20.9 SUMMARY OF ALGDDT COMMANDS

AUTO	*
BACK	
BREAK	*
CONTINUE	*
DEFINE	*
DIMENSION	
DUMP	
END	*
EXPERT	
FINISH	*
GOTO	*
(OR GO TO)	
HELP	*
KILL	*
LIST	*
NEXT	*
NOVICE	
OBJECT	
OFFTRACE	
ONTRACE	
PAUSE	*
PROFILE	
REDIRECT	*
SOURCE	
START	*
STATISTICS	
TRACE	
TYPE	*
UNWIND	*
WHERE	*

The commands marked with an asterisk (*) in the above list may also be abbreviated to a single character, thus S means START, not STATISTICS.

CHAPTER 21

MACRO SUBROUTINES

21.1 GENERAL

The subroutines which may be called from an ALGOL program must have one of the following attributes:

1. The routine must obey the FORTRAN calling interface specification, and be called by way of the FORTRAN interface procedures, for example, F10CALL, F10ICALL and so on. Further information can be found in Chapter 17, section 17.5.

2. True ALGOL-like routines must begin with instructions: to call the OTS routine PARAM; to set up the environment and obtain any parameters; to provide an exit for PARAM to return any results (in the case of a TYPE procedure).

This chapter deals with specific details of the ALGOL implementation. Reference should be made to files ALGSYS and ALGPRM for definitions etc., and to ALGLIB for examples of correctly written MACRO subroutines.

21.2 THE PROCEDURE HEADING

```
SEARCH ALGPRM,ALGSYS
```

must be included at the beginning of a MACRO procedure if the rest of the material uses symbolic names.

The first executable instruction in the procedure should be a call to the OTS routine PARAM. This is followed by a set of descriptor words which describe the type of the procedure and the type of each of the formal parameters.

Example:

```
INTEGER PROCEDURE P(A,B,C);  VALUE A;  INTEGER A;
REAL B;  STRING C;
```

would have a header similar to the following:

```
.EXIT==1
.A==3
.B==4
.C==7
JSP         AX,PARAM
EXP         PMB
XWD         0,11
XWD         $PRO!$I!$SIM 4
```

```
XWD         $VAR!$I!$FOV,.A
XWD         $VAR!$R!$FON,.B
XWD         $VAR!$S!$FON,.C
```

An explanation follows:

1. The first word must normally be JSP AX,PARAM (AX is accumulator 16; PARAM is a macro defined in ALGSYS, by the ALGDIR macro, and which expands to @%ALGDR+1, i.e. @400011 which contains the address of the OTS routine PARAM.)

2. The second word is the address of the post-mortem block (or zero if none). The post mortem block is used by the TRACE features, and is laid out as follows, in the low segment:

```
PMB:    0                       ; THE PROFILE WORD
        XWD     WORDS,CHARS     ; IN THE NAME
        SIXBIT/NAME/            ; PRINTED BY TRACE ETC.
```

NOTE

The OTS expects the name to be terminated by a zero byte ("SIXBITZ"). If the name is a multiple of six characters in length, an extra word of zeroes may have to be supplied.

3. The third word is the length of the fixed stack required. There must be enough space for the exit formal and the parameters, see below. More space can be requested, and may be used for any purpose as local storage by the procedure.

   Length of fixed stack needed:

For the exit instruction (always needed)	1 word
For a typed procedure with 1-word result (Integer, Real, Boolean)	1 word
For a typed procedure with 2-word result (Long Real, String)	2 words
For each parameter called by name (Any type)	3 words
For each parameter called by value:	
Integer,Real,Boolean,Label,Procedure	1 word
Array,String,Long Real	2 words

4. The fourth word describes the procedure in the left half, and the number of parameters +1 in the right half. The left half must always be

   "$PRO!$SIM!TYPE"

   where "type" is one of:

```
$N      NON-TYPE
$I      INTEGER
$R      REAL
$LR     LONG REAL
$S      STRING
$B      BOOLEAN
```

5. The remaining words describe the formal parameters, and tell PARAM where to put them. The left half is a bit pattern, giving the kind, type and status of the parameter:

   Kind is one of:

   $VAR Variable
   $ARR Array
   $PRO Procedure

   Type is one of $I, $R, $LR, $S or $B as described above, or one of:

   $L Label
   $WV Any type
   $AB Arithmetic or Boolean
   $IB Integer Or Boolean
   $WF Real or Long Real
   $WA Arithmetic (Integer, Real or Long Real)

   Status must be one of:

   $FON Formal by "name"
   $FOV formal by "value"

   NOTE

   All three fields must be included.

   The right half of each descriptor word is the offset in the fixed stack where PARAM is to store the elaborated parameter. Conventionally, these are in ascending order of parameter position, but this is not necessary, and gaps may be left if desired. PARAM obeys these offsets implicitly (to check them would be inefficient). If insufficient words are allocated, or insufficient fixed stack is requested for each parameter, parameters will overwrite each other and be lost, or worse. The space required for each parameter is as described above.

   It is conventional to use symbolic names for the offsets, as in the example above.

   NOTE

   Word 1 is always the exit instruction, and word 2 (1-word result) or words 2 and 3 (2-word result) must be reserved for the result of a typed procedure.

21.3 ACCESSING FORMAL PARAMETERS

Parameters may be called by "name" or by "value". In the by "value" case, PARAM places the actual value on the fixed stack in the location specified by the right half of the descriptor words: thus to load parameter A in the example into A3,

```
MOVE     A3,.A(DL)
```

will suffice. Obviously, two word parameters occupy two words of stack. DL (accumulator 15) is always the base of the fixed stack.

Accessing formal parameters by "name" is more complicated. Essentially, PARAM stores instructions on the fixed stack, which the procedure XCT's. The three locations are known as F[0], F[1] and F[2].

F[0] contains an instruction which, when executed, will fetch the parameter into accumulator 0 (and 1 for a two word value). So, to load parameter B's value into accumulator 0, in the example, write:

```
XCT     .B(DL)
```

NOTE

> The instruction in F[0] (.B(DL) in this case) may be anything from a "MOVEI A0,value" for a constant actual parameter, to a PUSHJ to a complicated OTS routine which may in turn call other parts of the user program ("thunks"). All accumulators except DL, DB and SP may be destroyed, and the stack may be shifted.

The various name parameters must be evaluated exactly once each, and in order left to right, to obey the Algol rules. Therefore storing into a formal by name is a two-step process. At the correct point in the evaluation of parameters in the left to right sequence, an XCTA (XCT 1) of F[0] is written. This elaborates the address of the parameter into A2, which must now be saved. When it is required to store a value (from A0 and perhaps A1) into the parameter, XCT F[1] is written, with A2 set up as it was after the F[0]. (In very simple cases of actual parameters, A2 has no meaning and F[1] contains an instruction to store the value directly, e.g. a MOVEM.) Thus, to store the contents of A3 into parameter B in the example, write:

```
PUSH    SP,A3       ; THE XCTA MAY CLOBBER ANY AC!!
XCTA    .B(DL)      ; SET UP A2
POP     SP,A0
XCT     .B+1(DL)    ; STORE A0 INTO B.
```

NOTE

> The XCTA must be in the proper place in the left to right sequence but the XCT F[1] need not be.

F[2] is never referenced directly by the procedure as it is used by PARAM to store information needed to evaluate the parameters. A simple case would be: suppose the example procedure P were called by

```
I:= P(I,123456789,"ABC");
```

Then the three words for B on the fixed stack would be:

```
.B(DL)      F[0]    MOVE        A0,.B+2(DL)
.B+1(DL)    F[1]    SYSER1      11,
.B+2(DL)    F[2]    ^D123456789
```

(The SYSERl UUO produces an error-message, because a formal cannot be used as a store when the actual is a constant.)

In more complicated cases, F[2] is used to store the context (in the left half, = delocated DL), and address (right half) of a thunk.

Special considerations apply for formal arrays, labels, procedures and strings.

The code to go to a formal label is:

```
XCT             .L(DL)                  ;  F[0]
JUMPN           A2,(A2)
```

A2 will be zero if the actual is a switch whose subscript is out of bounds.

For formal arrays, the header word pair (see section 21.6) is stored in F[0] and F[1] (as arrays are always static): XCT's should not be used for arrays.

For strings, XCT F[0] will return with A0 and A1 containing the string header and A2 containing the address: XCTA and XCT F[1] are used to store a new string header, but not to store a byte into an existing string. In some circumstances A2 will point to A0, that is, the string header will be in A0 and A1, and nowhere else. This only happens when the actual parameter is a dynamic expression, i.e. a call of a string procedure.

To obey a formal procedure, an XCT of F[0] is coded, followed by actual parameter descriptor words, just as though a normal procedure were being called, with the PUSHJ replaced by the XCT F[0]. The first desriptor is XWD type of procedure wanted, number of actuals+1. The remaining words, one for each actual, are coded with bits 1 to 11 as described above for formals. Bit 0 is set if dynamic, that is, a formal actual, or a thunk. Bits 18-35 are the value for an immediate constant (simple static expression), the address for a non-immediate constant (regular static expression) or a static (e.g. own) variable, or the Q address for dynamic variables. The Q address is the offset in the fixed stack of the actual variable, and bits 12-17, the P address, are the appropriate procedure level, that is, the offset in the current display of the context DL of the variable. For a thunk (a dynamic expression), the right half is the address of the thunk. Before using dynamic variables and expressions as actual parameters to formal procedures, the reader is advised to inspect some generated code (with ALGDDT) similar to the case in question, as there are complications.

21.4 RETURNING RESULTS FROM TYPED PROCEDURES

The procedure stores the value into the second word (and third word in the case of a two-word result) of the fixed stack, thus:

```
MOVEM           A7,.EXIT+1(DL)              ;where .EXIT = 1
```

in the example.

21.5 PROCEDURE EXITS

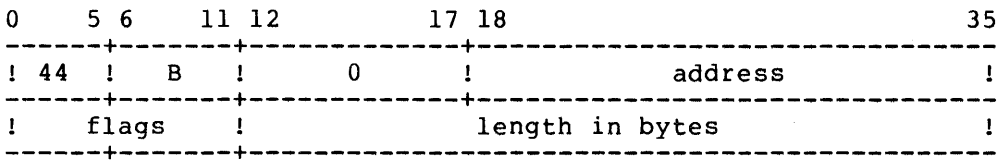
The first word in the fixed stack contains an appropriate jump to return to the call-site (via PARAM: the stack must be "unwound", and the result may need its type converted), so the procedure exits by use of:

```
JRST .EXIT(DL) ;Where .EXIT = 1
```

21.6 FORMATS OF VARIABLES

Integer and Real variables are obvious. Long real variables are in the format appropriate to the CPU type in use (KA or KI/KL). Boolean variables are zero for "false", and non-zero (including negative values) for "true".

String variables are passed as a two word header. The format is:



b is the byte-size.

The first word is a byte pointer to the string (such that an ILDB will get the first byte). This word is zero for a null string.

The second word is mostly taken up by the length in bytes (odd bytes in the last word may be rubbish). The flags are:

```
bit 1 = dynamic (not a constant)
bit 2 = result of a string-type procedure
```

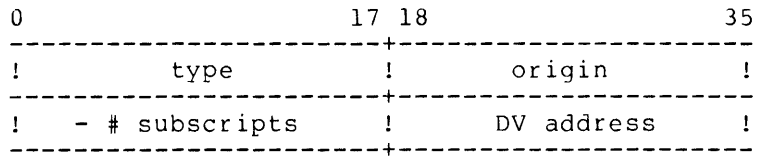
When a new value is assigned to a string, any old string must first be deleted. The safest way to do this is to call the OTS routine STRASS (string assign), which takes the necessary care not to delete constants and so on, thus:

```
PUSH      SP,word-0-of-new-string's-header
PUSH      SP,word-1-of-new-string's-header
PUSH      SP,word-0-of-old-string's-header
PUSH      SP,word-1-of-old-string's-header
PUSHJ     SP,STRASS
; Result in A0,A1
```

NOTE

STRASS will copy the string unless bit 2 is set in the new string's header.

Arrays are also passed as a two word header, thus:



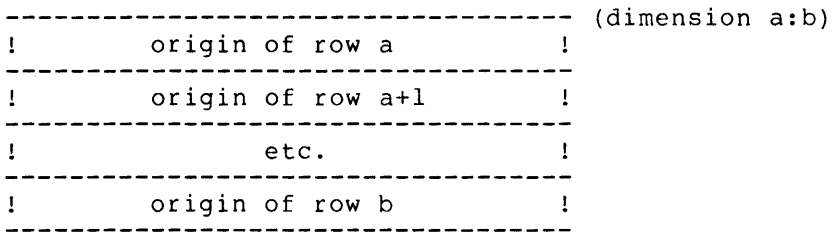
where:

1. "type" is integer etc. (coded as in descriptor words, see above).

2. "origin" is the address of the (possibly imaginary) zero'th element in the array (if one-dimensional), or in the N-1'th Iliffe vector (if N-dimensional); see below.

3. "DV address" is the address of the Dope Vector, which is used for subscript checking (and by the Debugging System): two words per dimension, containing the low and high bound of each.

This applies to a vector. However, for a matrix (two dimensional), the right half of word 0 is the address of the zero'th element of the Iliffe vector, which is as follows:



where the origins are addresses of the (possibly imaginary) zero'th elements. For more than two dimensions, a hierarchy of Iliffe vectors exists. (The purpose of this is to allow addresses of elements to be calculated without multiplications). Arrays and strings (except for constant strings) are allocated in the Heap, and occurences local to a block are deleted at block exit. Strings are initialised to "null" (the first word of the header is zero) at block entry.

21.7 PROCEDURES WITH A VARIABLE NUMBER OF PARAMETERS

This facility is not provided by the Algol-60 language but is available in the DECsystem-10/20 ALGOL run-time system. It is needed for some library procedures (IMAX etc., READ, PRINT, etc.)

The procedure must have a special heading, viz:

```
          XWD     DL,offset
PROC:     JSP     AX,PAR0
          etc.
```

where "offset" is a location on the fixed stack where PAR0 (an offshoot of PARAM) stores the number of actuals. The maximum number of actuals allowed is determined by the number of formal descriptors. If wild types were used in the formal descriptors, the actual types

can only be obtained by picking up the actuals' descriptor words (PRGLNK(DL) contains the call-site address, advanced over the actuals.)

21.8 INCLUDING THE PROCEDURE IN THE LIBRARY

The procedure may simply be added to the library (ALGLIB.REL) with FUDGE2 or MAKLIB.

If the procedure has a different version for KA and KI/KL processors, the LIBENT macro should be used (refer to file ALGSYS.MAC). An entry must also be made in a table in ALGSTB (in the compiler), to associate the name with the alias: this is done by using the LIB macro, which is described in comments in ALGSTB.MAC.

21.9 UTILITY ROUTINES AVAILABLE

A number of the routines in the OTS are available for use by Macro procedures.

21.9.1 Getting Core

Core may be obtained in the Heap by calling GETOWN; any amount may be had, and GETOWN will expand the program, shift the stack etc. as necessary. Calls are:

To get core:

```
MOVEI     A0,amount wanted
PUSHJ     SP,GETOWN ;  or GETCLR if wanted zero'd
;  on return, A1 = address of core.
```

To return core:

```
MOVEI     A0,0
MOVE      A1,address of piece
PUSHJ     SP,GETOWN ;  not GETCLR!
```

21.9.2 Input/Output

Buffered mode input/output may not be done directly, because the monitor will allocate the buffers above the stack, which may later have to expand or be shifted to allow the heap to expand. Direct access to the OTS routines is however allowed, as follows.

21.9.3 Device Open

```
MOVE    A0,[SIXBIT/device/]
HRLI    A1,#-buffers-required   ; 0 will give default
HRRI    A1,channel-number
MOVEI   A2,mode                 ; 0 is ASCII
PUSHJ   SP,INPT                 ; or OUTPT
```

On return, A1 is zero if successful. If unsuccessful, A1 contains an instruction which if XCT'd will give the standard failure message, etc.

To obtain a free channel number, scan the I/O directory (16 words starting at %IODR(DB)) for a zero word.

21.9.2.2 File Open

```
MOVEI   A1,channel number
MOVE    A2,[SIXBIT/Filename/]
MOVE    A3,[SIXBIT/Extension/]
MOVE    A4,[<protection>B9]
MOVE    A5,[project,,programmer]
PUSHJ   SP,OPFILE
```

On return, A0 is zero if successful. If not, it contains the monitor error code (as returned from LOOKUP or ENTER) plus 100 octal. The channel number is still in A1, and the standard error message and action may be obtained by obeying:

```
IOERR   5,(A1)
```

21.9.2.3 File Close

```
MOVEI   A1,channel number
PUSHJ   SP,CLFILE       ; May give error messages.
```

21.9.2.4 Channel Release

```
MOVEI   A1,channel number
PUSHJ   SP,RELESE
;non-skip if error (channel not in use)
;skip return if OK
```

21.9.2.5 Channel Select

For Input:

```
MOVEl   A?,channel-#
HRLM    A?,%CHAN(DB)
```

For Output:

```
MOVEl   A?,channel-#
HRRM    A?,%CHAN(DB)
```

21.9.2.6 Read Byte

```
PUSHJ   SP,INBYTE       ; USES A10-A13
;non-skip if end of file
;here with byte in A13
```

21.9.2.7 Write Byte

```
MOVE    A13,byte
PUSHJ   SP,OUBYTE       ; USES A10 - A13
;non-skip if end of file
;skip if OK.
```

21.9.2.8 Break Output

```
PUSHJ   SP,BRKBYT       ; USES A10 - A13
;non-skip if end of file
;skip if OK
```

21.9.2.9 Read Number

```
MOVEI   A2,type ; 0 for integer
                ; 1 for real
                ; 2 for long real
                ; 4 for any type
PUSHJ SP,READ.  ; note the period !
                ; uses almost all AC's.
; here, number is in A0 (and A1 if Long Real)
; type is in A2 if "any" was used.
```

21.9.2.10 Print Number

```
MOVEI   A2,type ; as above (0,1, or 2)
; number in A0 (and A1 if Long Real)
; A3 = # digits before decimal point
; A4 = # digits after decimal point
; A3 = A4 = 0 for "standard mode"
PUSHJ   SP,PRINT.       ; note the period.
                        ; uses most AC's
; on return, A3 = # characters output.
```

To print an integer in standard mode,

```
; integer in A0
PUSHJ   SP,IPRINT
```

21.9.2.11 String Terminal Output

(regardless of current channel settings)

```
MOVEI   A1,address of string (0 byte ends)
PUSHJ   SP,MONIT        ; if no break required, or
           MONIT0       ; if break required, or
           MONSIX       ; if string is sixbit (no break)
```

or `PUSHJ SP,CRLF ; to type a newline`

NOTE

Due to the buffering action of the OTS using OUTSTR (TTCALL 3,) UUO's may cause output to appear in the wrong order.

21.10 GENERAL NOTES

1. any register may be destroyed by a procedure, except for:

   DB=14 Pointer to database
   DL=15 Pointer to current display
   SP=17 Stack pointer

   NOTE

   These three registers must always be correct, since many OTS routines (parameter fetching, stack overflow handler, error handler, etc.) depend on them.

2. No register is safe over XCT's to access formals by "name".

3. The stack may be shifted by any OTS routine, including XCT's to access formals by "name". Therefore any stack addresses must be "delocated" (use SUBI An,(DB)) before, and "relocated" (use ADDI An,(DB)) after every such call. This also applies to string header addresses (returned in A2 by XCT F[0] - see above.)

4. Any amount of stack and heap may be used by a procedure (subject to external constraints). The OTS will extend these areas as needed.

5. The OTS will trap all arithmetic errors including overflows.

6. The OTS will intercept ^C (this causes entry to the Dynamic Debugging System (ALGDDT) in version 10).

7. All formals by "name" must always be accessed, and in strict left to right order, and once each only. "Access" here means XCT or XCTA on F[0]. Even formals by name which are not wanted in certain cases must still be accessed, if the Algol side-effects rules are to be obeyed. (For more detail on these requirements consult the revised report).

INDEX

ABS, absolute value, 5-2, 5-3, 17-2
Actual parameter, 11-1, 11-2, 11-9
Addition, 2-1
Addition, operator precedence, 5-1
ALGDDT, 20-1
ALGDIR, 21-2
ALGLIB, 1-3, 17-1, 17-3, 18-4, 21-1, 21-8
ALGOL-60, 1-1, 6-1, 7-1
ALGOL-68, 1-1
ALGOL symbols, 2-1
ALGOTS, 1-3, 16-1, 17-1, 18-10, 21-1, 21-8, 21-11
ALGPRM, 20-6, 21-1
ALGSTB, 21-8
ALGSYS, 21-1, 21-2, 21-8
ALL, 17-6
ALT-modes, 20-6
AND, Boolean operator, 5-3, 5-4
AND, delimiter word, 2-3
ARCCOS, 17-1
ARCSIN, 17-1
ARCTAN, 17-1
Arithmetic conditions, 5-4, 5-5
Arithmetic expressions, 5-3
Array bound checking, 18-6, 18-7, 20-4
Array declarations, 2-1, 2-2, 9-1
ARRAY, delimiter word, 2-3
Array elements, 9-2
Array subscript, 14-1
Array typeout, 20-3
Arrays, 9-1, 18-10
Arrays, compiler restrictions, 1-2
Arrays, OWN, 15-1
ASCII, 2-1, 16-2, 16-3, 16-8, 16-11, 17-5, 20-5, 20-16
ASCII constants, 2-2, 4-3
ASCII strings, 13-3, 20-3
Assembly switch FTGETCHK, 18-11
Assignment statement, 20-6
Assignments, 1-2, 4-2, 6-1
AUTO, commands, 20-2, 20-11, 20-17
AUTO-lists, 20-8, 20-9, 20-10, 20-16
Automatic conversion, 5-2
Automatic conversion of constants, 4-2
Automatic type conversion, 16-8
BACK, 20-14, 20-17
BACKSPACE, 16-10
Batch, 18-10
BEGIN, 6-3, 10-1, 20-2, 20-10
BEGIN, delimiter word, 2-3
BEGIN-END, 20-2
Binary image mode, 16-2
Blanks, 20-2
Blocks, 18-8
Block structure, 10-1
BOOL, dummy function, 5-5
Boolean (and octal) constants, 4-3
BOOLEAN, delimiter word, 2-3
Boolean, expressions, 4-3, 5-3, 7-2
Boolean, scalar variables, 3-2, 3-3
Boolean variables, 20-4
Bound, array dimension procedures, 17-2
Bounds, 9-2
Bounds, OWN arrays, 15-1
Brackets, 4-3, 4-4, 9-1
BREAK, 20-17
BREAKOUTPUT, 16-6
Break output, special editing character, 16-7, 16-12, 21-10
Buffering, 16-3
Buffers, I/O, 18-10
BUFFERS:n, 18-2
Byte manipulations, 17-3
Byte, read, 21-9
Byte size, string, 20-3
Byte strings, 13-1, 16-6
Byte subscripting, 2-1, 13-2, 14-1, 16-10

^C (Control C), 20-2, 20-4, 20-7, 20-13, 20-14, 20-15, 20-16
CALL, 17-4
CALL BY NAME, 11-1, 11-6, 21-3
CALL BY VALUE, 11-1, 11-2, 21-3
Card punch, 16-1, 16-2
Card reader, 16-1, 16-2
Carriage return, 4-3, 20-6
CDP, device name, 16-2
CDR, device name, 16-2
Channel, release, 21-9
Channel number, 16-1, 16-4
Channels, 16-2, 16-3, 16-10

Channels, undefined, 18-6
Character constant, ASCII, 4-3
CHECKOFF, 18-2
/CHECKOFF, compiler switch, 18-7
CHECKOFF, delimiter word, 2-3
CHECKOFF, directive, 18-6
CHECKON, 18-2
/CHECKON, compiler switch, 18-7
CHECKON, delimiter word, 2-3
CHECKON, directive, 18-6
CLOSEFILE, 16-4
COMMENT, 2-4
Comments, 2-2, 2-4, 20-2
Comment after END, 11-10
COMMENT, delimiter word, 2-3
Commentary, 11-10
Comparison operators, 13-1
Compilation, independent, 18-3
Compilation, programs, 18-1
Compiler extensions, 1-2
Compiler restrictions, 1-2
Compiler switches, 18-2
Compiler version words, 17-5
Compound statements, 6-3, 14-2
Compound symbols, 2-2
CONCAT, 17-2
Concatenation, 13-3
Conditional operands, 14-1
Conditional statements, 7-2, 14-2
%CONDL, 20-6
Constants, 4-1, 4-2, 4-3, 5-2
Constants, REAL, compiler extension, 1-2
Constraints, compiler, 1-1
CONTINUE, 18-11, 20-4, 20-10, 20-12, 20-17
Control-backarrow, 20-2, 20-6
Control C, (^C), 20-2, 20-4, 20-7, 20-13, 20-14, 20-15, 20-16
Control transfers, 7-1, 7-2
Control-underline, 20-2, 20-6
Control-back arrow, 4-3
Controlling listing of the source program, 18-7
Conversion type, 17-2
COPY, 13-3, 13-4, 17-2
Core size, 17-5
COS, 17-1
COSH, 17-1
CREF, 18-8, 20-8
/CREF, 18-7
Current data-base register, 20-6

Dangling ELSE ambiguity, 7-2
Data, numeric output, 16-9
Data transmission, 16-1
Date, 17-5
%DB, 20-6
DCALL, 17-4
Debugging programs, 18-1, 18-5
Debugging, system, dynamic, 17-6, 20-1
Declaration of an array, 10-3
Declarations, 3-2, 10-1
DECtape, 16-1, 16-2, 18-1
Default Input/Output, 16-10
DEFINE, 20-11, 20-17
Delete, files, 16-4
DELETE, typeless procedure, 13-4, 17-2
Delimiter words, 1-2, 1-3, 2-2, 2-3, 5-3, 18-2
Designational expressions, 14-3
Device allocation, 16-1
Device modes, 16-2
Devices, 16-1
DIM, 17-2
Disk, 16-1, 16-2, 18-1
DIMENSION, 20-4
DIMENSION, 20-17
DIV, 5-1, 5-2
DIV, delimiter word, 2-3
Division, 2-1
Division, operator precedence, 5-1
%DL, 20-6
DO, 8-1
DO, delimiter word, 2-3
DSK, device name, 16-2
DTA, device name, 16-2
Dummy functions, BOOL and INT, 5-5
Dummy variables, 11-1
DUMP, 17-6, 20-13, 20-17
Dynamic bounds, 10-3
Dynamic Debugging System, 17-6, 20-1
Dynamic Trace, 17-6, 18-9, 20-16

Elapsed time program, 20-16
ELSE, 7-2, 14-2
ELSE, delimiter word, 2-3
Embedded assignments, 6-2, 20-13
END, 6-3, 10-1, 20-2, 20-17
END, delimiter word, 2-3, 11-10
End-of-file, 16-11, 16-12
ENDFILE, 16-10
ENTIER, transfer function, 5-2, 5-3, 6-1, 17-2
EQV, Boolean operator, 5-3, 5-4
EQV, delimiter word, 2-3
Error returns, 16-3, 16-5
Error trap numbers, 18-6
Evaluation, of statements, 19-1

Execution time program, 20-16
EXP, 17-1, 18-6
EXPERT, 20-14, 20-17
Exponent, optional (real constants), 4-1
Exponentiation, 2-1
Exponentiation, operator precedence, 5-1
Exponents, 2-2, 16-8
Expressions, evaluation of, 6-2
EXTEND, 20-11
EXTERNAL, compiler extension, 1-2
EXTERNAL, declaration, 11-9, 17-4, 18-4
EXTERNAL, delimiter word, 2-3
External procedures, 11-9, 11-10, 20-2

F-10, FORTRAN subroutine, 17-4, 21-1
F10CALL, 17-4
F10DCALL, 17-4
F10LCALL, 17-4
F10RCALL, 17-4
F10ICALL, 17-4, 21-1
F-40, FORTRAN subroutine, 17-4
FALSE, 3-2, 5-3, 5-4, 20-4
False, Boolean constant, 4-3
FALSE, Boolean OWN variable, 15-1
FALSE, delimiter word, 2-3
FDATE, 17-5
Field manipulations, 17-3
File devices, 16-1, 16-4
Files, 16-4, 16-11, 18-1, 18-6, 21-9
FINISH, 20-13, 20-16, 20-17
Fixed-point, printing, 16-9
Floating-point data, 16-8
Floating-point, format, 16-9
FOR, 7-2, 8-1, 8-2, 8-3, 9-2
FOR, delimiter word, 2-3
Form feeds, 2-4
Formal parameters, 1-3, 11-1, 11-2, 11-3, 11-5, 11-10, 21-3
Formal parameters, compiler restrictions, 1-2
FORTRAN, 3-1, 5-1, 16-8, 17-6, 21-1
FORTRAN double precision, compiler extension, 1-2
FORTRAN, interface procedures, 17-4
FORTRAN logical variables, 3-2
FORTRAN, terminology equivalents to ALGOL, 1-3
FORWARD, 11-9
Forward declarations, compiler, 1-2
FORWARD, delimiter word, 2-3
Forward references, 11-8
Forward references, compiler restrictions, 1-2
FTGETCHK, assembly switch, 18-11
FUDGE2, 21-8
Function procedures, 11-1

GETOWN, 21-8
GFIELD, 17-3
Global variables, 10-2
GO, delimiter word, 2-3
GO TO, 7-1, 20-12, 20-17
GOTO, 7-1, 12-1, 14-3, 19-1, 20-12, 20-17
GOTO, delimiter word, 2-3

HEAP, 18-10, 18-11, 21-8, 21-11
HEAP:n, 18-2
HELP, 18-2, 20-16, 20-17
HLP:ALGDDT.HLP, 20-16

ICALL, 17-4
Identifiers, 1-3, 2-1, 3-1, 5-2
Identifiers, compiler restrictions, 1-2
IF, 7-2, 14-2
IF, delimiter word, 2-3
/IGNORE, 20-10
IMAX, 17-3
IMIN, 17-3
IMP, Boolean operator, 5-3, 5-4
IMP, delimiter word, 2-3
INCHAN, 16-12
Independent compilation, 18-3
Indirect command files, 20-16
INFO, 17-5, 18-11
Input, 16-1, 16-2, 16-3, 16-6, 16-10
Input channel, 16-4
INPUT, library procedure, 16-1
INPUT, statement, 16-2
Input/Output, 21-8
INSYMBOL, 16-6
INT, dummy function, 5-5
INTEGER, delimiter word, 2-3
Integer constants, 4-1
Integer conversions, 5-5
Integer remainder, 5-5
Integer, scalar variables, 3-2
INV, 18-6

I/O channel status, 16-10
I/O directory, 21-9
IOCHAN, 16-10, 16-11, 16-12
%IODR, 21-8

Jensen's Device, 11-6

KA10, 18-2
KI10, 18-2
KILL, 20-8, 20-10, 20-17
KL10, 18-2

Labels, 1-3, 2-2, 7-1, 11-1, 11-2, 11-9, 12-1, 14-3, 16-3, 16-5, 18-8, 20-7
Labels, compiler restrictions, 1-2
LABEL, delimiter word, 2-3
LARCTAN, 17-2
Layout of declarations, 11-7
LB, 17-2
LCALL, 17-4
LCOS, 17-2
LENGTH, 13-3, 17-2
Length, string, 20-3
LEXP, 17-2
LIBENT, 21-8
Library procedures, 13-3, 18-10
LINE, 18-5, 18-7
LINE, delimiter word, 2-3
Line feed, 4-3
Line numbers, 18-7, 20-5
Line printer, 16-1, 16-2
LINK, 11-10, 13-3, 18-4, 20-1, 20-2
Linking loader, 1-3, 18-4
LINKR, 13-3
LIST, 18-2, 20-9, 20-17
Listing of the source program, 18-7
LISTOFF, 18-7
LISTOFF, delimiter word, 2-3
LISTON, 18-7
LISTON, delimiter word, 2-3
LLN, 17-2
LN, 17-1, 18-6
Loading programs, 18-4
Local variables, 10-2
Logical device name, 16-1
Logical I/O, 16-10
LONG, delimiter word, 2-3
LONG REAL, compiler extension, 1-2
Long real constants, 4-1, 4-2
Long real, scalar variables, 3-2
Long real variables, 4-2
LPT, device name, 16-2
LSIN, 17-2
LSQRT, 17-2

MACRO, 18-4, 20-5, 21-1
Magnetic tape, 16-1, 16-2, 16-11
MAKLIB, 21-8
Mathematical procedures, 17-1
Matrix, 9-1
Modes, device, 16-2, 16-3, 18-6
MODES, 20-5
MTA, device name, 16-2
MULTIPLE ASSIGNMENTS, 6-2
Multiplication, 2-1
Multiplication, operator precedence, 5-1

Name, calling parameters by, 11-1
New line, special editing character, 16-7
NEWLINE, symbol procedure, 16-7
NEWSTRING, string procedure, 13-4, 17-2
Newton-Rapheson Method, 11-4
NEXT, 20-12, 20-17
NEXTSYMBOL, 16-6
NOERRORS, 18-2
NOLIST, 18-2
/NOLIST, compiler switch, 18-7
NONUMBERS, 18-2
NOQUOTES, 18-2
NOSYMBOL, 18-3
NOT, Boolean operator, 5-3
NOT, delimiter word, 2-3
NOVICE, 20-14, 20-17
Null strings, 13-2
/NUMBERS, 18-7
Numeric constants, 2-1, 4-1
Numeric data, 16-9
Numeric labels, 1-3, 7-1
Numeric labels, compiler restrictions, 1-2
Numeric procedures, 16-8

OBJECT, 20-2, 20-5, 20-17
Object code, 20-5
Object Time System, 17-1
Octal addresses, 20-2, 20-5
Octal (and Boolean) constants, 2-2, 4-3
Octal input/output, 16-9
Octal representation of strings, 20-3

OFFTRACE, 17-6, 18-10, 20-16, 20-17
ONTRACE, 17-6, 18-10, 20-16, 20-17
OPENFILE, 16-4, 16-5, 16-12
Operating environment, 1-3
Operator precedence, 5-1
OR, Boolean operator, 5-3, 5-4
OR, delimiter word, 2-3
Order of evaluation, 5-1
Output, 16-1, 16-2, 16-3, 16-6, 16-9, 16-10
Output channel, 16-4
OUTPUT, library procedure, 16-1
OUTPUT, statement, 16-2
OUTSYMBOL, 16-6
OWN arrays, 15-1
OWN, delimiter word, 2-3
OWN variables, 15-1
Overflow, 18-6

PAGE, symbol procedure, 16-7
Page throw, special editing character, 16-7
PARAM, 21-1, 21-2, 21-3, 21-4, 21-6
Parameters, 1-3, 2-1, 11-1, 21-7
Paper-tape punch, 16-2
Paper-tape reader, 16-2
Parameter, 1-3, 11-1
PAUSE, 17-6, 20-2, 20-5, 20-7, 20-8, 20-9, 20-17
Peripheral devices, 16-1
Plotter, 16-2, 16-11
PLT, device name, 16-2
Post-mortem trace, 18-10
Precedence, 5-1, 6-2
Precedence, Boolean operators, 5-3
Precision, 4-2, 5-2
PRINT, 16-9, 16-10
PRINT, number, 21-10
PRINT OCTAL, 16-9
Procedures, 1-3, 11-1, 11-6
Procedure bodies, 11-3
Procedure calls, 11-5
Procedures, compiler restrictions, 1-2
PROCEDURE, delimiter word, 2-3
Procedure headings, 11-2, 11-3
Processor type, 17-5
/PRODUCTION, switch, 18-10
Program structure, 10-1
PROFILE, 18-8, 20-15, 20-17
Project-programmer number, file, 16-4
Protection, file, 16-4
PTP, device name, 16-2
PTR, device name, 16-2

Quota, 16-11
Quotes, 4-3

RAND, 17-6
Random number, 17-6
Range, real constants, 4-2
RCALL, 17-4
READ, 16-8, 16-10, 16-12
Read, number, 21-10
Readability symbol, 2-1, 3-1, 20-2
READOCTAL, 16-9
Real constants, 4-1
REAL, constants, compiler extension, 1-2
REAL, delimiter word, 2-3
Real, scalar variables, 3-2
Recursion, 11-7, 20-14
Recursive, variables, 15-1
REDIRECT, 17-6, 20-13, 20-16, 20-17
REENTER, 17-6, 20-2, 20-4
RELEASE, 16-5
Relocatable binary, 1-1
REM, 5-1, 5-2
REM, compiler extension, 1-2
REM, delimiter word, 2-3
Rename, files, 16-4
Reserved words, 2-3, 2-4
Reserved word mode, 11-10
Reserved word quotes, compiler extension, 1-2
Restrictions, 11-8, 11-10, 16-8
Restrictions, compiler, 1-2
RETRY, 20-12
Revised report ("Revised Report on the Algorithmic Language ALGOL-60"), 1-1, 1-2, 13-1, 19-1
REWIND, 16-10
RMAX, 17-3
RMIN, 17-3
Rounding, 6-1
Run-time diagnostics, 18-5
Run-time error, 18-10
Run-time general information, 17-5
Running programs, 18-1

SCALARS, 20-13
Scalars, compiler restrictions, 1-2
Scalar variables, 3-2
Scope, 10-2, 11-9, 20-4, 20-7
Scope, switch declarations, 12-1
SELECTINPUT, 16-4
SELECTOUTPUT, 16-4

Semi-colons, 20-6
Setting line numbers in listings, 18-7
SETRAN, 17-6
SFIELD, 17-3, 17-4
Side-effects, evaluation of expressions, 6-2
Side-effects, evaluation of subscripts, 19-1
Side-effects, typeout, 20-4
SIGN, signum function, 5-2, 5-3, 17-2
Significance, precision, 5-2
Significance, real numbers, 4-2
/SILENT, 20-10
SIN, 17-1
Single-pass compiler, 1-1
SINH, 17-1
SIXBIT, 20-5
SIXBIT, strings, 20-3
SIZE, 17-2
SKIPSYMBOL, 16-6
SOURCE, 20-16, 20-17
%SP, 20-6
Spaces, 2-2, 2-4, 4-3, 16-6, 16-8
Space, special editing character, 16-7
SPACE, symbol procedure, 16-7
SQRT, 17-1, 18-6
Stack, 18-10, 21-2, 21-11
Stack analysis, 18-8
Stack pointer, 20-6
Stack shifts, 17-5
Stack trace, 20-14
START, 20-2, 20-12, 20-17
Statements, 6-1
STATISTICS, 20-16, 20-17
STEP, delimiter word, 2-3
STEP-UNTIL, 8-1
STOP, 20-10
String assignments, expressions, 13-1
String comparisons, 2-2, 13-2, 13-3
STRING, compiler extension, 1-2
String constants, 2-1, 2-4, 4-3, 13-1, 13-3, 16-1, 16-6
String declarations, 13-1
STRING, delimiter word, 2-3
String expressions, assignments, 13-1
String, length and byte size, 20-3
String operators, 13-1
String output, 16-6
String procedures, 13-3, 16-8, 17-2
String scalar variables, 3-2
String variables, 3-2, 21-6
String typeout, 20-3
String variables, 13-3, 16-8
Strings, 1-2, 13-1, 13-2, 18-10, 16-6, 16-8, 20-6
Strings, concatenated, 13-3
Subroutine, FORTRAN, 11-1
Subscripting, byte, 2-1, 13-2
Subscripts, 2-1, 9-1
Subtraction, 2-1
Subtraction, operator precedence, 5-1
Switch declarations, 12-1
SWITCH, delimiter word, 2-3
Switch element, 14-3
Switch option, 2-2
Switches, 1-2, 12-1, 14-1, 16-3, 16-5, 18-2, 18-3, 18-5, 18-10, 19-1
SWITCH.INI, 20-14
.SYM file, 20-1
Symbol file, 20-1
Symbol procedures, 16-7
Symbol table, 18-3
Symbols, compound, 2-2
SYS:ALGDDT.HLP, 20-16
SYSERl UUO, 21-5
System parameters, 20-6

Tab, special editing character, 16-7
Tabs, 2-2, 2-4, 16-8, 20-2
TAIL, 13-3
TAN, 17-1, 18-6
TANH, 17-1
Terminal output buffer, 18-10
Terminals, 16-2, 16-3
Terminology, 1-3
THEN, 7-2, 14-2
THEN, delimiter word, 2-3
TIME, 17-5
TRACE, 17-6, 18-8, 18-9, 18-10, 20-15, 20-17, 21-2
/TRACE, switch, 18-10
TRANSFILE, 16-12
TRAP, 18-5
TRAPNO, 18-5
TRUE, 3-2, 5-3, 5-4, 20-4
TRUE, Boolean constant, 4-3
TRUE, delimiter word, 2-4
TTY, device name, 16-2
TYPE command, 20-3, 20-17
Type conversion, 6-1, 11-2, 20-6
Typeless procedures, 11-3, 13-4

UB, 17-2
UNTIL, 8-1, 8-2
UNTIL, delimiter word, 2-4
UNWIND, 20-14, 20-17

VALUE, 11-1
VALUE, delimiter word, 2-4
Variables, labels, 18-8
VDATE, 17-5
%VERSHN, 20-6
Virtual peripherals, 16-1

WHERE, 20-14, 20-17
WHILE, 7-2, 8-1, 8-2, 8-3
WHILE, compiler extension, 1-2
WHILE, delimiter word, 2-4
WRITE, 16-7, 16-12, 17-5

XCTA, 21-4

DECsystem-10/20 ALGOL
Programmer's Guide
AA-0196C-TK

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Please cut along this line.

-------Do Not Tear - Fold Here and Tape-------

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MR1-2/E37
MARLBOROUGH, MASSACHUSETTS 01752

------ Do Not Tear - Fold Here and Tape-------

Cut Along Dotted Line