Data General

EXTENDED ALGOL User's Manual

093–000052–05

EXTENDED ALGOL

User's Manual

093–000052–05

Ordering No. 093-000052

Printed in the United States of America
Rev. 05, January 1975

Original Release	February,1971
First Revision	September,1971
Second Revision	August, 1972
Third Revision	February,1973
Fourth Revision	March, 1974
Fifth Revision	January,1975
086-000017-00	March, 1975

This revision of the Extended ALGOL User's Manual, 093-000052-05 and its addendum, 086-000017-00, supersedes 093-000052-04 and 017-000016-00 and constitutes a minor revision to the manual. A vertical bar on the outer margin of each page indicates substantially new, changed, or deleted information. A list of changes is given following the index to the Reference Manual section.

INTRODUCTION

Data General's Extended ALGOL compiler for all DGC computers is suitable for business applications, for systems programming, and for research and engineering applications. The extensions to ALGOL 60 were selected to make DGC Extended ALGOL a general-purpose language offering those features most wanted by users rather than merely a language in which complex mathematical algorithms could be concisely written.

The features of standard ALGOL 60 which differentiate ALGOL from other commonly used languages include recursive procedures, dynamic storage allocation, a modular "block" organization, long variable names, integer or character labels, and a very flexible generalized arithmetic.

Some of the major DGC extensions to ALGOL 60 provide for character string manipulation, file manipulation, DGC supplied I/O procedures that allow free-form or formatted output and provide a cache memory management facility, use of pointers and based variables, multi-precision arithmetic allowing the user to achieve, for example, up to 60 digits of precision, and subscripted labels.

Character strings are implemented as an extended data type to allow easy manipulation of character data. The program may, for example, read in character strings, search for substrings, replace characters, and maintain character string tables efficiently. Dynamic conversion of data types includes conversion of strings to and from integer real, pointer, and Boolean data types, allowing the user an unusual degree of freedom both in use of character strings and in their output format.

The simplified I/O procedures for use by most ALGOL programmers use one call for all data types. Free-form read and write or formatted output according to a "picture" specification of the output line are available. The I/O procedures provide for random as well as sequential access of individual data values, a number of bytes, a line of information, or all or part of a file for reading and writing.

Cache memory management I/O procedures may be used when very large procedure and data files must be manipulated as in compiler writing. The procedures allow access to single words, blocks of words, and the contents of active files using a fast buffering mechanism.

Pointers and based variables provide a programming technique that allows a systems programmer, in particular, and other programmers as well to achieve a high level of object code efficiency. Pointers and based variables allow programmers to explicitly manipulate machine addresses. For example, the programmer can force a subscript calculation to be performed only once in a frequently executed portion of a program. As another example, if the programmer knows that an external variable will not be modified by a call, he can convey this knowledge to the compiler.

In effect, use of pointers and based variables bypasses compiler generation of extra code usually needed to allow for "worst case" computations where information is not available about a variable until run-time.

Multi-precision arithmetic is available for both fixed and floating point data types, allowing up to 15 computer words of precision in both cases. Precision can be specified for both variables and arithmetic literals. Radix conversion is permitted; any radix from 2 through 10 can be specified.

Recursive procedures are allowed. An array declaration may be any arithmetic expression, including function calls, negative numbers, and subscripted variables. Integer labels and conditional expressions can be used. Some of the other language features and extensions to ALGOL are:

Dynamic conversion of parameter type (integer, real, string, pointer, Boolean), allowing one program to process data of several types.

Dynamic storage allocation, freeing the programmer from many details of data layout and storage assignment.

N-dimensional arrays which may be allocated dynamically at run-time.

Bit manipulation, using logical operators and octal or binary literals. Built-in functions are provided to allow efficient access to data at the bit level.

Efficient object code and commented assembly language output. Code is optimized for register usage, generation of literals, optimal use of machine instructions, and efficient storage allocation.

Explicit diagnostics both at compile time and run-time. Compatibility with the Data General Symbolic Debugger aids run-time debugging.

Object code and run-time compatibility with assembly language to permit referencing not only of external programs and data compiled by the ALGOL compiler but of any object program.

Full label capability, permitting integer labels, identifier labels and subscripted identifier labels to be used.

Declaration of literals permitting an identifier to be subscripted for any type of literal within a program.

Declaration of operators permitting the user to declare, implement and use other operators besides the arithmetic and Boolean operators provided with ALGOL.

HOW TO USE THE EXTENDED ALGOL USER'S MANUALS

The Extended ALGOL User's Manuals is divided into two separate parts. Part 1 is a tutorial called How to Program in ALGOL. The tutorial presents the basic concepts of ALGOL for programmers unfamiliar with ALGOL or with compiler languages.

Part 2 is a complete description of Extended ALGOL called the Extended ALGOL Reference Manual.

Each part contains its own table of contents and separate index.

HOW TO PROGRAM IN ALGOL

HOW TO PROGRAM IN ALGOL

CONTENTS

GENERAL PROGRAM ORGANIZATION

A basic ALGOL program starts with the word *begin* and ends with the word *end*.

```
begin
  .
  .
  .
end                         ←basic program
```

begin and *end* are written in italics because they are reserved words (called keywords). ALGOL recognizes keywords as having a special meaning; the user cannot change the meaning of keywords or use them for his own program names. The user writes a keyword at the teletypewriter either in all upper case letters or all lower case letters.

A basic program is called a block.

Inside a block are declarations and statements. Declarations list user program names and their characteristics. User program names are called identifiers. Statements show the action the program will take.

Declarations of identifiers must precede their use in statements.

```
begin declarations;
declarations;               ←declarations precede statements
statements;
statements;
end;
```

An example of a block, containing declarations and statements is:

```
begin
real pi;                    ←declarations
integer k;
real R [300] , AREA [300];

pi := 3.1416;
for k :=1 step 1 until 150 do ←statements
AREA [k] := pi×R[k]  ↑2;
end;
```

GENERAL PROGRAM ORGANIZATION (Continued)

An ALGOL program can be written in free form. This means that a declaration or a statement can be continued from one line to the next and that more than one statement or declaration can be written on a line. For example, the previous program could be written:

```
begin real pi; integer  k;
array R[300], AREA[300
 ] ;pi:=3.1416; for k
:= 1 step 1 until 150 do
AREA[k]   := pi×R [k] ↑ 2; end;
```

←but the program is hard to read if it does not have some format.

Since the end of a line is not a delimiter in ALGOL as it is in the DGC assembler, other delimiters must be used. A few common ALGOL delimiters are the keywords themselves and the symbols:

; - usually ends a declaration, statement, or a comment.

, - separates items in a list.

: - terminates a label definition.
separates the lower and upper bounds of array dimensions.

() - enclose parameters of procedures and built-in functions.
enclose precision of numeric variables.
enclose the maximum declared length of strings.
enclose expressions to be evaluated as entities.

[] - enclose dimensions of an array in a declaration or the subscripts of an array or label in a statement.

space - separates identifiers that are not otherwise separated, such as two keywords together or a keyword followed by an identifier.

Examples of required blank spaces are shown below as triangles. The other blanks are not significant and are used only for legibility.

```
begin △ real △ pi; integer △ k;
real △ array △ R[300], AREA [300];
```

Other delimiters will be introduced later in this manual. The Reference Manual contains a complete list in Chapter 4.

DECLARATIONS

Why Declarations Are Needed in ALGOL

When a programmer writes a program for compilation in a high-level language such as ALGOL, he uses several, sometimes a very large number of program variables that are assigned different values during execution.

A declaration tells the ALGOL compiler the name of a program variable, called an identifier. In addition, a declaration shows:

How much storage space the identifier needs.

How and when storage is allocated and released.

What kind of identifier is involved.

Much of this information does not actually appear in most declarations but is given by default. For example:

```
integer k;
```

← declaration of k.

tells the compiler:

The identifier is k.

k can have integer values.

Default storage for integers should be used for k.

Size of Storage for Identifiers

The basic storage unit is a 16-bit word. The default storage for the various types of ALGOL identifiers is:

Integers	-	one word
Real (decimal) values	-	two words
Boolean values	-	one word
Pointers	-	one word
Strings	-	32 characters (two characters per word)

DECLARATIONS (Continued)

Size of Storage of Identifiers (Continued)

The default storage allocations can be overridden by the programmer by including precision in parentheses immediately following the data type in the declaration. For numeric values, the precision indicates the number of machine words used to store the datum. For strings, the precision indicates the maximum number of characters the string may have.

```
integer (2) k, y;        ←k and y are each stored in 2 words.
real (5) array x [10]; ←each element of array x stored in 5 words.
string (50) line;        ←line has a maximum of 50 characters.
```

To approximate the number of decimal digits of precision that can be stored in a given number of 16-bit words, use the following formulas. n represents the declared precision in words.

integer digits=5(n-1) +4 integer range=$\pm 2^{16n}-1$

real digits=5(n-1) +2 $10^{-75} \leq$ real range $\leq 10^{78}$

Allocation and Release of Storage for Identifiers

By default, an identifier is allocated storage when the block in which it is declared is entered (*begin* keyword) and the storage is released when the block is terminated (*end* keyword).

A large ALGOL program can be made up of many basic blocks. Some blocks are entered and exited many times. Allocating and releasing storage by block makes more storage available for other identifiers.

However, suppose a programmer wants to enter and exit a block many times during program execution. The block contains a real identifier, R. The programmer wants to enter the block each time with R having the same value it had when the block was last terminated.

If the programmer declares R with the keyword *own*, R will be stored in a separate area from the other identifiers. In the *own* area, space allocated to identifiers is never released until the entire program terminates.

```
own real R;
```

DECLARATIONS (Continued)

Data Types

The declaration of a data type tells the compiler the kind of values an identifier can have. The programmer must declare a data type for all identifiers. There is no default declaration of data type.

```
integer  x;        ←has values like +15,3,-25
real     y;        ←has values like 3.1416 and -.22266
boolean  z;        ←has value true or false
string   r;        ←has values like $5.25 or abcde
pointer  p;        ←has an integer value.  See Reference Manual.
```

Labels are explicitly declared by their use as labels; however, formal parameters that are replaced by labels are declared *label*. See section on procedures.

```
100: x :=3;        ←100 is a label on the statement
                    x :=3;
```

Arrays

So far, only identifiers that can have one value at a time have been used. It is possible to declare an array. An array is an identifier of an ordered set of values. Each member of the set is called an array element.

Arrays, like simple identifers, are declared with a data type and storage characteristics. These apply to each element in the array.

```
integer (2) array Matrix;←declaration of array, Matrix.  Each
                          element in Matrix can have an integer
                          value up to 9 digits long.
```

If you look at the declaration of Matrix, you see that the compiler has no way of knowing how many elements Matrix is supposed to have. While this kind of array declaration is used under circumstances described later, the programmer will usually declare:

How many elements are in the array.

How each element is to be numbered. (This also will determine the order in which values are stored into identifiers.)

Arrays (Continued)

This part of the array declaration is called dimensioning the array. For example:

```
integer array Matrix [25];
```

The single number 25 tells the compiler that Matrix is an array containing 26 elements, numbered:

Matrix [0], Matrix [1],..., Matrix [25]

and values are assigned in that order.

An array can have more than one dimension. In fact, it can have up to 128 dimensions. For example, an array containing real values for the lengths and diameters of pipe might be written with two dimensions as follows:

```
real array pipe [5,5];
```

The declaration tells the compiler that the array, pipe, has 6x6 or 36 elements. The elements are

pipe[0,0], pipe[1,0], pipe[2,0],...,pipe[6,0], pipe[0,1], pipe [1,1 ...,pipe[6,1],...,pipe[5,6], pipe[6,6]

The identifying numbers of each element in the array are called the subscripts of the array. If you look at the elements of array pipe, you will see that the first subscript varies most rapidly. In an array of several dimensions, values are assigned in this way: the first subscript varies most rapidly, then the second subscript, then the third subscript, etc.

If the programmer wishes, he can give an array a different starting number from zero. For example, array pipe could have been written:

```
real array pipe[-5:0,1:6];
```

Pipe still has 36 array elements but now they are numbered:

pipe[-5,1], pipe[-4,1],..., pipe[-1,6], pipe [0,6]

DECLARATIONS (Continued)

Arrays (Continued)

The first number of each dimension gives the lower bound of the dimension; the second number gives the upper bound. The lower bound must be a smaller integer than the upper bound. Besides integer and real arrays, arrays of strings can be declared. The maximum length of each element of a string array can be declared; otherwise, the default limit of 32 characters will be set for each element.

```
begin string(8) array ID[9,9];
```

ID is declared as a two dimensional 10x10 array of strings. The maximum length of each string is eight characters.

Variable strings are an extension to ALGOL. Some of the ways in which they can be used are discussed later.

Lists of Identifiers in Declarations

The programmer does not have to write a separate declaration for each and every program variable. Quite often a number of program variables have the same data types and storage characteristics. In this case, the programmer can write one declaration, listing all the identifiers.

```
begin integer i,i1,i2,i3;
real x,y,z;
real array M[5,5], z[8,8], A,B[2,2]; ←A and B have the same
                                       dimensions.
```

Local and Global Identifiers

The block structure of ALGOL permits blocks within other blocks. In the following diagram, three blocks are shown. The blocks labeled B2 and B3 are inside the block labeled B1.

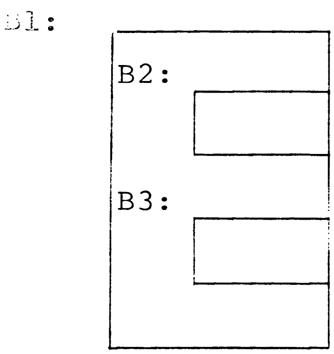
```
B1: ┌── begin   real A;          ←A is declared in block B1.
    │
    │  B2: ┌─begin boolean B;     ←B is declared in block B2.
    │      │
    │      └─end B2;              ←B2 ends. end can be followed by a
    │                              string of characters.
    │  B3: ┌─begin real C;        ←C is declared in block B3.
    │      └─end B3;              ←B3 ends.
    │
    └── end B1;                   ←B1 ends.
```

Local and Global Identifiers (Continued)

Since B2 and B3 are both within block B1, any identifier declared in B1, such as *real* A, is defined for blocks B2 and B3.

Identifier A is said to be local to block B1 (the block in which it is declared) and global to blocks B2 and B3 (the blocks in which it is defined.)

Identifier B is local to block B2 and identifier C is local to block B3. Elsewhere, both these identifiers are undefined. Why this is so can be seen in the following diagram of the blocks.



←storage is allocated for *real* A.

←storage is allocated for *boolean* B.

←storage is released for B.

←storage is allocated for *real* C.

←storage is released for C.

←storage is released for A.

Labels are declared by their appearance as labels within a given block. For example, the blocks B1, B2, and B3 might each contain labeled statements.

```
B1: begin real A;
    B2: begin boolean B;
        lab: ---;          ←lab is a label local to B2.
        end B2;
        ttg: ---;          ←ttg is a label local to B1 and
                            global to B2 and B3.
    B3: begin real C;
        22 : ---;          ←22 is a label local to B3.
        end B3;
    end B1;
```

Like declared identifiers B and C, labels lab and 22 are undefined except in their own blocks. Note, however, that the labels of the blocks, B2 and B3, are outside the blocks they label and are local to block B1 and global to blocks B2 and B3 as shown in the following diagram.

DECLARATIONS (Continued)

Local and Global Identifiers (Continued)

```
B1:
 ┌──────────────────┐
 │ B2:              │
 │     ┌────────────┤
 │     │lab: ---;   │
 │     └────────────┤
 │ttg: ---;         │
 │ B3:              │
 │     ┌────────────┤
 │     │22: ---;    │
 │     └────────────┤
 └──────────────────┘
```

Even though a label does not appear in a declaration, it is a data type; if an identifier is used as a label in a block, it cannot be used as any other type of datum.

Arrays can be declared with variable dimension bounds such as:

```
real array z[i,j];        The bounds are 0 to i and 0 to j.
```

The appearance of variable dimension bounds in an array declaration constitutes a use of the identifiers. Identifiers must be declared and defined before they are used. Thus i and j must be global to the block containing the declaration of array z. For example, the following is legal:

```
B: begin integer    i,j;        ←i and j declared in block B.
     i:=50; j:=100;             ←i and j defined in block B.
   C: begin real array A[i,j];  ←i and j used in dimensioning
      end C;                     array A in Block C.
   end B;
```

However, the following is illegal:

```
B: begin integer i,j;
   real array A[i,j];
   end B;
```

A later section describes procedures and formal parameters of

DECLARATIONS (Continued)

procedures. Formal parameters are not allocated storage as are actual program variables and therefore the rules of declaration and definition before use do not apply to formal parameters.

STATEMENTS

Statements are programming instructions. They indicate how operations are to be performed using the declared identifiers.

ALGOL statements are very flexible so that programmers unfamiliar with ALGOL can use short, simple statements. Experienced ALGOL programmers, however, can nest statements within other statements. In fact, an entire block may be treated as a single statement.

Some examples of simple statements are:

```
A :=B+1.0;
```

←Assignment. B+1.0 is evaluated and placed in location A.

```
go to Lb_13;
```

←Unconditional transfer to the statement labeled Lb_13.

```
if bool then go to B;
c :=c/d;
```

←Conditional transfer.

bool is a Boolean variable. If bool has the value *true*, a transfer is made to the statement labeled B. If bool is *false*, the assignment statement is executed.

```
tag2:;
```

←Dummy statement providing a label to which to transfer.

```
for i :=0,2,25 do
x[i] :=y[i]+i;
```

←*for* statement.

The *for* statement causes a loop. The variable i is assigned the first value (0) of the list 0,2,25, and the assignment statement is executed. Then i is assigned the second value (2) and the assignment statement is executed, etc.

```
proc23(x,y,z);
```

←procedure call

STATEMENTS (Continued)

A call to a procedure named proc23 is made from the current block.

```
comment: Comments contain explanatory information;
```

ALGOL comments are written as statements, beginning with the keyword *comment* and ending at the first semicolon.

Often, a programmer wants a group of statements to be treated as a single statement. A common example is a group of statements following a *for* statement, where the programmer wants the loop to include the group of statements. He can use the keywords *begin* and *end* to "block" his statements.

```
for p :=5,10,15,20
do begin
     A[p] :=p↑2;
     B[p] :=A[p] -x;
     C[p] :=B[p] +A[p];
end;
```

The three assignment statements will be executed for each value of p.

Statement Termination

Statements shown previously have generally been terminated by a semicolon. However, statements may be terminated in some instances by the keyword *end* or the keyword *else*. For example, the previous compound statement could be written:

```
for p :=5,10,15,20
    do begin
        A[p] :=p↑2;
        B[p] :=A[p] -x;
        C[p] :=B[p] +A[p]     ←end terminates this statement
      end;
```

The keyword *else* can terminate a statement in a conditional clause.

STATEMENTS (Continued)

Statement Termination (Continued)

```
if x=0 then go to LABLAA else ←else terminates go to LABLAA
if x>0 then  y :=x
else                          ←else terminates y :=x
x :=x +1;
```

Although *end* terminates one statement, the keyword does not signal that another statement or declaration can begin. The keyword *end* can be followed by a string of characters. Anything following *end* will be treated as a string wntil the next statement terminator is encountered, that is, the keyword *else*, the keyword *end*, or a semicolon.

```
end of block 25;   ←string "of block 25" is terminated by a
                    semicolon.
```

Forgetting to terminate an *end* can lead to difficulties such as the following:

```
  .
  .
  .
end
begin integer i,j;  ←"begin integer i,j" is treated as a string
  .                  following end, not as a declaration.
  .
  .
```

To avoid the problem, put a semicolon after the keyword *end* when it is needed.

Assignment Statement

The basic statement is the assignment statement that permits the value of an expression to be stored in a location represented by an identifier.

```
variable := expression;
          ↑
  Assignment symbol
```

STATEMENTS (Continued)

Assignment Statement (Continued)

```
begin real B,C; integer A;boolean boo;
     .
     .
     .
A :=0;                          ← assignment of constants to variable
B :=C :=2.5;                      locations.  Note that 2.5 is
boo := true;                    ← assigned both location B and to
                                  location C.
```

```
a :=a+2                         ← assignment of simple expressions
b :=c↑3;                          to variable locations.  ↑ shows
boo := ¬boo;                      exponentiation. ¬ means logical
                                  not.
```

ALGOL expressions can be relatively simple as shown above or can represent highly complex processes. A few more simple expressions might be

```
v
z+4
(-b+sqrt (d) )/2/a               ← / means division.
d+abs (w[0] -y×w[1] -sin (x/2)   ← × means multiplication.
w[k[i]]                          ← subscripts can be nested to
                                   any depth.
```

Note the terms abs, sin, and sqrt in the expressions. These are references to functions, and the parenthesized expressions following the function reference are the actual parameters passed to the function when it is referenced. Functions and how they are referenced are described later in a section on procedures.

The variable on the lefthand side of the assignment and the expression on the right must have compatible data types. Each variable type can be assigned an expression of the same data type as in:

STATEMENTS (Continued)

Assignment Statement (Continued)

```
begin real x,y;
integer i,j; pointer p;
boolean b,c; string (8) char;

     i :=j-4;                 ←integer to integer

     x :=x/y×3.5;             ←real to real

     b :=¬c;                  ←boolean to boolean

     char :="$25.10";         ←string to string

     p :=address (y);         ←address is a pointer function
```

In addition, many conversions are possible:

```
begin integer i,j; boolean b,c;
   :
i :=b∧c;             ←boolean to integer.  (∧ is logical and).
c :=j;               ←integer to boolean.
```

If b ∧ c evaluates to *true* (1), integer i will contain one and if the expression evaluates to *false*, then i will contain a zero. In integer to boolean conversion, the integer expression (j in this case) is evaluated. c will be assigned the value false if j contains all zeroes and will be true in every other case.

```
begin integer i,j; pointer p;
   :
j := p+5;                 ←pointer to integer
p := i;                   ←integer to pointer
```

A pointer is one word long and contains a memory location (integer). Therefore, integer to pointer and pointer to integer conversion is permissible with the limitation that the integer must be one word long (default precision).

STATEMENTS (Continued)

Assignment Statement (Continued)

```
begin pointer p, q;string S,T;
   .
   .
   .
S :=p+2;            ←pointer to string
q :=T;              ←string to pointer
```

The integer value of the pointer expression will be assigned as a character string of all digits to S. When converting to a pointer, T will be examined and the result assigned to the pointer q up to the first non-digit or up to the one-word limit of q.

```
begin string S,T; boolean c, b;
    .
    .
    .
S := c;             ←boolean to string
b := T;             ←string to boolean
```

The boolean expression is evaluated to a zero or one. A zero or one will be assigned as the character of S. When T is evaluated, the result will be assigned to b as *false* (zero) if the string contains all zeroes. Otherwise, the value *true* (one) will be assigned.

```
begin string ST,V; integer i,j;
    .
    .
    .
ST := i-25;         ←integer to string
j := V;             ←string to integer
```

The integer expression evaluates to the following format:

[-] n ...n

where: n is a digit

[-] indicates that a minus sign is optional (negative integer).

Assignment Statement (Continued)

The evaluated expression is assigned to the string ST. In converting from string to integer, characters will be assigned to the integer variable up to the first character that does not follow the format above, such as a decimal point, or up to the limit of the precision of the integer, which in this case is one word.

```
begin string S,T;real a,b;
    :
S := a/1.5;              ←real to string
b := T;                  ←string to real
```

The real expression evaluates to the following format:

```
[-]n...n[.n...n][E[-]nn]
```

where: n is a digit.

[] surround optional parts of the format.

E indicates an exponent following.

The evaluated expression is assigned to the string S. In converting from string to real, characters will be assigned to the real variable up to the first character that does not follow the format above, or up to the limit of the precision of the real variable, which in this case is two words.

```
begin integer i,j; real x;
    :
x := i+2;                ←integer to real
j := x/3;                ←real to integer
```

An integer expression is converted to real by evaluating it and placing the decimal point after the last digit. A real expression is converted to integer by evaluating it and using a function, called the entier function, to select the nearest integer value.

All the expressions described in this section are simple expressions. There are also conditional expressions which may be used in assignment statements. Conditional expressions are described in the Reference Manual.

go to Statement

A *go to* statement transfers control to another statement in the program. The keywords *go to* are followed by a label or an expression that evaluates to a label. The expression can be a subscripted label variable or a switch identifier.

Labels are either identifiers (alphanumeric characters beginning with a letter) or unsigned integers.

```
tag1:  x := x+1.0;          ←identifier label
  :
go to tag1;
  :
go to 10;
  :
10:  y6 := y×x;             ←integer label
```

A subscripted label variable in a *go to* statement evaluates to a subscripted label. Labels can have a single subscript.

```
tag[1] :  x := x+pi/4;  ⎫
          :             ⎪
tag[2]:   x := pi/2     ⎬ ←subscripted labels
          :             ⎪
tag[3]:   x := x+pi;    ⎭
          :
          go to tag[I];   ←I evaluates to 1, 2, or 3.
```

Switch designators are described in the Reference Manual. They also appear as subscripted expressions to be evaluated in the *go to* statement.

STATEMENTS (Continued)

go to Statement (Continued)

Because of the way identifier storage is allocated and de-allocated by block, a statement must transfer control within the block or to an identifier global to the block.

```
A: begin integer j;

        tag: ---;

        B: begin real z;          ←global transfer


        go to tag;
        go to Lab;

     Lab: ---;                    ←local transfer
     end B;
     end A;
```

if Statement

if statements use a truth value as a switch to determine transfer of control. There are two formats..

```
if boolean_expression then unconditional_statement;

if boolean_expression then unconditional_statement else statement;
```

If the boolean expression evaluates to *true* the *then* statement is executed; otherwise, the *then* statement is skipped. The arrows in the example below show how control is passed.

```
if true then statement; next_logical_statement;

if false then statement; next_statement;

if true then statement else statement; next_logical_statement;

if false then statement else statement; next_logical_statement;
```

STATEMENTS (Continued)

if Statement (Continued)

Boolean expressions and the logical and relational operators used in forming them are described in the Reference Manual, which should be consulted if you are not familiar with Boolean logic. Briefly, a boolean expression consists of

a+b≠c	←simple arithmetic expressions (a+b and c) are used with relational operators (= ≤ < ≠ > ≥)
boo∧loob	←boolean expressions (boo and loob must be declared *boolean*), used with operators: ┌ *(not)* ∧ *(and)* ∨ *(or)* ⊃ *(imp)* ≡ *(equ)* ⊕ *(xor)*
a+b≠c∨boo∧loob	←a combination of the above two boolean expressions

The *then* statement can be any statement or set of statements as long as it doesn't contain another *if* statement.

```
if a≠b then a :=b;                        } simple then statements
     :
if c>d then go to 25;
     :
if e<5 then begin x :=y :=x↑2;            } blocked then statements
                  y :=y+25.25;
                  go to 30 end;
```

The *else* clause can be an *if* statement. This means that a series of switches can be set up. For example, the previous statements could be rewritten

```
if a≠b then a :=b else
if c>d then go to 25 else
if e<5 then begin
            x :=y :=x↑2;
            y :=y+25.25;
            go to 30 end;
```

if Statement (Continued)

Simple expressions were discussed in the section on the assignment statement. The sequence

```
if boolean_expression then ...
```

is a conditional expression and can appear anywhere a simple expression can be used, except following the keywords *then* and *go to*. Conditional expressions follow the rules for data typing. See the Reference Manual for information on conditional expressions.

for Statement

The *for* statement allows a given statement or statements to be executed repetitively with a controlled variable set to different values. The statement or statements are executed as many times as there are values for the controlled variable. The statement format is:

```
for controlled_variable := list_of_values_and_expressions
do statement(s);
```

At its simplest, the list can contain only values as in:

```
for j :=1, 25, 350, 4, -6 do A[i,j] :=B[j];
```

However, the list can contain variables and expressions.

for Statement (Continued)

```
for j :=1, a+3,x/y, if x≠y then 25 else -6 do A[i,j] :=B[j];
```

In addition, a list item can contain either the keyword *while* or the keywords *step* and *until*. A *while* clause would be:

```
for x :=y/2 while y≠z do...
```

The keyword *while* is followed by a boolean expression. The statement following *do* executes as long as the boolean expression is *true*.

A *step-until* clause would be

```
for a :=1 step    2 until  101 do...
          ↑          ↑         ↑
    initial     incre-     final
    value       ment       value
```

The list item is equivalent to the simple list: 1,3,5,...,101. The initial, incremental and final values can be any expression or value. Some examples of *for* statements are:

```
for i :=0.1 step -0.01 until .005
    do x :=i×ln(x);                  ←ln(x) is the natural logarithm
                                      function
for j :=1 step 1 until 100 do
      A[i] :=B[i]-C[i];

for k :=1, k+1 while z[k]>k do
    begin z[k] := k;                 ←compound statement following
          y[k] := k-1;                begin. Both assignment state-
        end;                          ments are executed as part of
                                      the loop.
```

PROCEDURES

Procedures are basic ALGOL programs that are called for execution. Begin blocks can be entered by sequential execution of statements. Procedures are only entered when they are called.

Declaring a Procedure

The format of a procedure declaration consists of a heading and the text or body of the procedure. The body of a procedure can be a single statement, a group of statements delimited by *begin* and *end* as described on page 11, or a block containing declarations and statements.

At a minimum, the heading of a procedure must contain the word *procedure*, followed by the procedure identifier. In addition, the heading may contain additional information about the procedure, described later in this section.

The procedure identifier follows the word procedure in the declaration. Then the text of the ALGOL procedure is written.

```
Z:   begin

procedure  ZERODIV;        ←procedure ZERODIV is declared in block Z
     :                     ←statement containing procedure body
```

Rules that apply to other identifiers apply to procedures as well. A procedure must be declared before it is used (called). It must be declared in the block in which it is called unless, like some identifiers, it is an *external* procedure.

Assume that ZERODIV is a program that is used to prevent errors resulting from division by zero. ZERODIV sets up the following algorithm:

given: c :=a/b the following results are produced:

a value	b value	resulting c value
any	≠ 0	a/b
>0	0	999999
=0	0	0
<0	0	-999999

PROCEDURES (Continued)

Declaring a Procedure (Continued)

The full declaration of ZERODIV could then be:

```
Z:  begin
       .
       .
       .
procedure ZERODIV;
          if b≠0 then c :=a/b else
          if a=0 then c :=0 else
          if a<0 then c :=-999999 else
             c :=+999999;
```

Calling a Procedure

A procedure is called by writing its name as a statement.

```
Z: begin real array R[10,10], z[10,10,10], Y[10,10,10];
         real a, b, c;
      .
      .
      .
procedure ZERODIV;
     if b≠0 then c :=a/b else
     if a=0 then c :=0 else
     if a<0 then c:=-999999 else
        c := +999999;
           .
           .
           .
     a :=R[i,j];            ←Assign array elements to dividend and
     b :=z[i,j,k];           divisor.
     ZERODIV;                Call ZERODIV.
     Y[i,j,k] :=c;          ←Put result in proper location.
```

Returning from a Procedure

When a procedure is called, it executes until the end of the procedure is reached. The procedure then returns control to the statement immediately following the calling statement. In the ZERODIV example control returns to the assignment statement:

```
Y[i,j,k] :=c;
```

PROCEDURES (Continued)

Identifiers Used in Procedures (Continued)

ZERODIV is a block inside the block named Z. Both Z and ZERODIV use the identifiers a, b, and c. If a, b, and c are declared within ZERODIV, they will be undefined in block Z by the rules of block structure. Therefore, a, b, and c are declared in block Z.

```
Z:  begin real a,b,c;
    procedure ZERODIV;
         .
         .
         .
```

There are identifiers that are used only in a given procedure and they can be declared in the procedure.

External Procedures

An ALGOL procedure declaration can be compiled separately from any enclosing block. It can then be used as an external procedure by many programs. Assume that the procedure ZERODIV was compiled separately from any other block. Now, any block can call ZERODIV if the block has a declaration of ZERODIV as *external*.

```
begin
     external procedure ZERODIV;
     .
     .
     .
ZERODIV;                                ←call to ZERODIV
     .
     .
     .
```

Parameters of Procedures

The previous example showing ZERODIV as an *external* procedure raises the problem of identifiers a, b, and c once more. Must they be declared in each and every program that wants to call ZERODIV? ALGOL solves this problem by allowing the user to put dummy identifiers, called formal parameters into a procedure declaration. Then, the procedure can be called with real identifiers, called actual parameters.

PROCEDURES (Continued)

Parameters of Procedures (Continued)

With formal parameters, the declaration of ZERODIV could be:

```
procedure ZERODIV(a,b,c);            ←a,b, and c are formal parameters
real a,b,c;                          ←a,b, and c are declared.
if b≠0 then c :=a/b else
if a=0 then c :=0 else
if a<0 then c :=-999999 else
   c :=999999;
```

A parenthesized list of formal parameters follows the procedure identifier. These formal parameters will be replaced when the procedure is called.

The formal parameters must have data types specified. In the example, a,b, and c are specified as *real*.

If the body of the procedure is a block, formal parameters must be specified in the procedure heading, not in the block. If parameters are declared inside the block that is the procedure body, they will be undefined in the procedure heading.

Assume the same block used previously to call ZERODIV now wishes to call it to obtain a value for Y[i,j,k].

```
begin real array   R[10,10], z[10,10,10], Y[10,10,10];
      .
      .
      .
   ZERODIV(R[i,j], z[i,j,k], Y[i,j,k]);    ←Call to ZERODIV
```

When ZERODIV is called, array element R[i,j] replaces a, z[i,j,k] replaces b and Y[i,j,k] replaces c. There is no need to assign the values in the calling block. The assignment is made when the actual parameters are passed in the call.

The rules governing formal and actual parameters are given in the Reference Manual. As a general rule, formal and actual parameters must have the same shape; for example, a procedure or an array cannot replace a simple variable. Some examples of legal substitutions are:

PROCEDURES (Continued)

Parameters of Procedures (Continued)

```
begin real x,y,z; external procedure sum;
procedure XX(a,b,c,d,e,f);
     real a,b,c; boolean d;
label e; real procedure f;                     Procedure
begin.                                         Declaration
     :   procedure body
     .    (statement or                                      Calling
end;         block)                                          Block
     .
     .
     .
XX(x,y,z, true, Exit, sum);          ←call
     .
     .
     .
Exit:---
end;
```

Because formal parameters are only dummy identifiers, their declarations are not as restrictive as that of real identifiers. Note in the example that a label can be declared. Also, it is often useful to leave a parameter declaration somewhat vague to allow a larger number of possible replacements. For example an array formal parameter could be declared without dimensions.

Functions

A function is a procedure which, upon execution, results in a value. In fact, at some point in the function, an assignment statement assigns a value to the function identifier.

Since a function represents a value, it must have a data type. A data type is included in the declaration of a function.

```
real procedure arctanh (x) ;   ←real preceding procedure declares
real x;                         arctanh as having a real value
arctanh := 0.5×ln((1+x)/(1-x));←value is assigned to arctanh
```

Since a function represents some value, a function call is part of an assignment statement or other statement:

PROCEDURES (Continued)

Functions (Continued)

```
      .
      .
      .
real procedure arctanh (x);
      .
      .
      .
z :=z×arctanh(y);    ←function call to arctanh with actual param-
                      eter y
```

When execution of arctanh is complete, the value of arctanh replaces the call in the assignment statement.

A function has one of the ALGOL data types: *integer*, *real*, *string*, *boolean*, *pointer*, or *label*. (A label can be specified as a function type.)

Recursive Procedures

ALGOL permits recursive procedures. A procedure is recursive if it calls itself. An example is factorial computation.

```
integer procedure factorial(I);                          Declaration of
integer I;                                               integer function,
factorial := if I=0 then 1                               factorial.
else factorial (I-1)×I;    ←factorial calls itself
```

I/O Procedures Supplied to the User

ALGOL does not provide for I/O operations. Some externally compiled procedures are supplied with Extended ALGOL to handle user I/O. The I/O routines are described very briefly here, and the user should consult the Reference Manual before using the I/O package.

Before proceeding with I/O operations, the user must open a file for input or output. The "file" can be a data file in secondary storage or an I/O device. To open a file the user writes the call:

```
open(number,string);
```

PROCEDURES (Continued)

I/O Procedures Supplied to the User (Continued)

The number is one of the channels (0 to 7) that can be associated with a given file and the string is the name of the file.

```
open (1, INDEVICE);          ←INDEVICE is a string containing the
                              file name.

open (2, "myfile");          ←myfile is the literal name of a
                              disk file.

open (3, "$TTO");            ←$TTO is the teletypewriter on output.
```

Once a file has been opened, data can be read or written from it. The read and write calls are:

```
 read (number, list);

write (number,list);
```

The number is again the channel number associated with the file. The list is a list of variables, expressions, and string constants to be read from or written to the file.

```
open (2, "myfile");

write (2, a, b, c, d "<15>  timings follow: ", MATRX);
```

In the example, the user opens myfile and associates channel 2 with it. He then requests that certain variables a,b,c,d be written to the file. They will be written out according to the way they are formatted in the file and their data type; the user does not have to format them. The user then inserts a string constant 'timings follow:'. After this, MATRX, which is presumed to be an array of timing information, will be written to the file.

Within the string constant are the characters, <15>. The value 15 is the octal equivalent of the ASCII character for carriage return. Enclosed in angle brackets, the value is passed to the assembler and interpreted as a carriage return. As shown in the example, the data for a,b,c, and d are written on one line, then a carriage return is given. The string, "timings follow:"

and the matrix values are then written on a line.

When I/O operation for a given file is completed, the file must be closed. This insures proper updating of the file and releases the association between the file and the channel number. The format for the call is:

```
close (number);
```

Another I/O routine allows the user to generate data output in a large number of possible formats. The call is:

```
output (number, format, variable list);
```

The number is the channel number, The variable list is a list of variables, expressions, and string constants to be output.

Format is a format parameter that determines the format of the output values. The format parameter is enclosed in either accent marks or double quotation marks. The user can put text in the format parameter, and the text will be output exactly as written.

```
output (2,"             RESULTS OBTAINED ARE:");

            RESULTS OBTAINED ARE:
```

The user can also set up a field format for his data, using the character # to represent each character position of the data.

```
output (2, "RESULTS OBTAINED ARE:   #<15>", A);

RESULTS OBTAINED ARE:   345
```

In the example, the list consists of the variable A. The datum in location A is written in the format given.

If the output number is smaller than the field format, the number is right justified in the field with leading blanks.

PROCEDURES (Continued)

I/O Procedures Supplied to the User (Continued)

If the output number exceeds the field, the length of the field will be increased, thus, one # may be used to output any integer regardless of length.

A decimal point can be used in a field format. Assume variables have the following values: x=-456.78, y=999.123, z=.08

```
output (2, "#####.# ",x,y,z);

  -456.8   999.1      .1    ←note rounding of fractional values.
```

Signs + or - can be used in a field format. Without the sign, as previously shown, only negative values are output with a sign, and the minus sign requires a field format position.

If a plus sign is given, both positive and negative values are output with signs. If a minus sign is given, only negative values are output with signs. However, in both cases, the sign does not require a field format position.

```
output (2, "-####.# ",b,c)

  -4567.2  5858.0            ←both positive and negative numbers
                              can have four digits before the
                              decimal point
```

Character strings are output in the same format as decimal numbers, using # for each character position. The character string output can be from a variable in a file or can appear as a literal in the list of variables of the output statement.

```
output (2, "##########", i,j, "Price");  ←i and j are string
                                          variables; Price is a
                                          string literal
Item No.  Stock No. Price               ←possible output
```

Character strings are left justified in the field format with following blanks.

If the character string is longer than the field format, the entire string will be written.

```
output (2, "###", "ADDRESS");

ADDRESS
```

In the I/O procedure calls, read, write, and output, an array identifier in the output list causes all elements of the array to be transferred in order. In the next example, assume A is an array of seven integer elements.

```
output (2, "#####", A);

  345  777  567   23 4577  890  230
```

The octal equivalents of ASCII carriage control characters can be enclosed in angle brackets and included in the format field. The control characters are passed to the DGC assembler for interpretation, and allow many special formats to be set up.

```
output (2, "##########<15>", "STOCK ITEM", A);

STOCK ITEM
       345
       777
       567
        23
      4577
       890
       230
```

In the example above, octal 15 is the ASCII carriage return code. Octal control characters, enclosed in angle brackets, can be given in any literal string, not just in the format field of the output parameter list.

The Reference Manual contains additional examples of how a user can format output, such as preparing tables of values using *for* loops with output calls.

PROCEDURES (Continued)

Functions Supplied to the User

ALGOL has certain standard arithmetic functions that are supplied to the user, such as those for taking a sine, cosine, or square root (sin, cos, sqrt). In addition, Extended ALGOL has a number of additional functions, such as those permitting the user to manipulate bit strings and character strings. Some of these special functions will be discussed in sections following and others are described in the Reference Manual.

STRING VARIABLES AND ARRAYS

By an extension to ALGOL, character strings can be manipulated. Strings can be declared with a maximum length.

```
string (9) a,b;                ←a and b have a maximum of 9 characters.
string (2)  array c[1:8];      ←each element of c has a maximum of 2
                                 characters.
```

The default string length is 32 characters: the maximum length that can be declared is 16,283 characters. String literals are delimited by either accent marks (ASCII characters 140_8 and 047_8) or double quotation marks.

```
string (9) a,b;
a:=`xxxyyyzzz´;
b:="$25.67";
```

Subsets can be taken of strings, using built-in function substr.

```
string (9) a,b;
a:=`xxxyyyzzz´;                ←a contains "xxxyyyzzz"
b:= substr(a,3,7);             ←b contains "xyyyz"
a:= substr(b,3);               ←a contains "y"
```

The first parameter of substr names the string to be subset. The second gives the position of the first character in the string; the third is the position of the last character of the string. String array elements, as well as scalar variables, can be subset.

STRING VARIABLES AND ARRAYS (Continued)

```
string (9) a; string (2) array c,d[1:8];
a:="xxxyyyzzz";
for i:= 1 step 1 until 8 do begin
c[i] := substr (a,i,i+1);
d[i] := substr (c[i],2,2); end;
```

When the *for* statement is executed, the contents of the elements of arrays c and d will be:

```
c[1]:="xx"; c[2]:="xx"; c[3]:="xy"; c[4]:="yy"; c[5]:="yy";
c[6]:="yz"; c[7]:="zz"; c[8]:="zz"; d[1]:="x";  d[2]:="x";
d[3]:="y";  d[4]:="y";  d[5]:="y";  d[6]:="z";  d[7]:="z"; d[8]:="z";
```

Concatenation of strings can be handled by the length built-in function that returns the integer length of a string as its value.

```
string (10) a,b;
a:="xxx";
b:="yyy";
substr(a, length(a)+1, length(a)+length(b)) :=b;
```

The substring taken of a contains the original contents of a, to which are added the contents of b; thus, the substring contains "xxxyyy".

The index built-in function returns a value that represents the character position of a given character in the string.

```
string (10) a; integer b;
a:="xyzzz";
b:=index(a, "y");            ←statement is equivalent to b:=2;
```

Coding of the index function shows how string variables and the length and substr built-in functions can be used in programming.

STRING VARIABLES AND ARRAYS (Continued)

```
integer procedure index(a,b); string a,b;
begin integer i;
for i:= 1 step 1 until length (a) do
if substr(a, i, i+length(b)-1) = b then go to done;
i:=0;
done: index:=i;  end;
```

The ascii function can be used to convert a single character of a string to its numeric value in the ascii collating sequence.

```
s:="ABCDEF";
i:=ascii(s,5);              ← statement is equivalent to i:=105R8
```

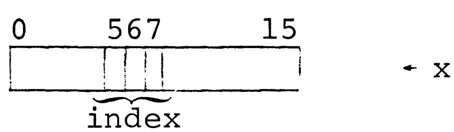
In the example, the fifth character of the string, E, is converted to its equivalent value, 105_8.

BIT MANIPULATION

Extensions to ALGOL allow programmers to manipulate bits at a level comparable to assembly language by using binary and octal literals with boolean operators and by using the built-in functions, shift and rotate. Bit manipulation is normally performed upon integers of default precision (one word) or the equivalent, such as boolean or pointer variables. Examples of shift and rotate functions are

```
x:= shift(y, -4);  ←logical left shift y by 4 bits and assign to x.
y:= rotate(y, +2); ←logical right rotate y by 2 bits.
```

The programmer can use logical operations and binary and octal literals to set bits, mask unwanted bits, or select bits from an integer. For example, suppose x is an integer containing a 3-bit index into an array in bits 5,6,7.



BIT MANIPULATION (Continued)

A variable, i, can be set to contain the index as follows:

```
i := shift(x,+8) and 111r2;     ←r2 means radix 2.
    or
i := shift(z,+8) and 7r8;       ←r8 means radix 8.
```

		index
where:	shift(x,+8) causes:	00000000xxxxxxxx
and either 111r2 or 7r8 causes:		0000000000000xxx
		index

When using a 6-digit octal literal as a mask, for example:

```
107777r8
```

a one-word precision must be specified for the literal. Otherwise, the leading two zeroes of the bit configuration will be considered significant, and the literal will be generated as a multi-precision (two-word) integer. Precision is specified as the letter p followed by the number of words of precision, which is one (1) in the case of a masking integer:

```
107777r8p1

  or
107777p1r8
```

CHANGING A RADIX

The programmer can set any radix up to and including 10, as shown for base 2 and base 8 in the section on bit manipulation. Simply follow the literal with the letter r and the desired radix:

.1R3	←base 3.	
1.3E9R4	←base 4.	The exponent is 4^9
101E-10R2	←base 2.	The exponent is 2^{-10}

WRITING AN ALGOL PROGRAM

The steps to follow in writing an ALGOL program are:

1. Study the problem. Can it be broken into several algorithms? Can you further generalize the algorithms for repetitive use? The first decisions are how to structure the problem - nested blocks, separately compiled procedures, etc.

2. When you decide upon the structure of your program you should decide what identifiers - variables, arrays, parameters, etc. - need to be declared in each block. Declaration of identifiers may be new to some programmers. It is essential to ALGOL programming.

   Be sure the data types you select are suitable not only for data storage but also as to compatibility of formal and actual parameters and variables that will be used together in expressions.

   Decide on the precision of integer and real data that you will need.

3. When the declarations have been written, the statements that implement the program can be written. Be sure to label statements you will transfer to and to write comments. Comments will help both you and other programmers.

4. Before attempting compilation, make a source-program debugging check. Have you put in the proper delimiters, blank spaces, and spelled the identifiers correctly?

5. When you attempt compilation, check the error messages carefully against your source program and make the necessary changes.

6. When you get your first ALGOL programs to compile, chances are they will not be very efficient. Check the compiled code carefully. Have you made full use of supplied functions, nesting of procedures, and external procedures? Have you used bit manipulation facilities? Experiment with your source program and see if you can improve the coding.

7. As you become more proficient in writing ALGOL programs, try to use the additional facilities described in the Reference Manual such as pointers and based variables. These facilities for sophisticated programmers such as systems programmers will also improve your coding efficiency.

★ ★ ★

HOW TO PROGRAM IN ALGOL

INDEX

EXTENDED ALGOL REFERENCE MANUAL

CONTENTS

CHAPTER 1 -- IDENTIFIERS AND KEYWORDS

An identifier is a string of one to 32 letters, digits, and underscore symbols (_) that must begin with a letter. Identifiers are names assigned by the programmer to variables and other program entities. Upper or lower case letters may be used. No blank spaces are permitted.

Examples:

a	get_symbol	ROUTINE2
A25	Aa	omega

Identifiers serve to identify simple variables, arrays, labels, switches, procedures, and pointers.

KEYWORDS

Certain keywords are completely reserved in ALGOL. They are:

and	*do*	*go to*	*operator*	*switch*
array	*else*	*if*	*or*	*then*
based	*end*	*imp*	*own*	*true*
begin	*eqv*	*include*	*pointer*	*until*
boolean	*external*	*integer*	*procedure*	*value*
comment	*false*	*label*	*real*	*while*
	for	*literal*	*step*	*xor*
		not	*string*	

Keywords must be written in all upper case or all lower case letters.

FUNCTION KEYWORDS

Certain functions are provided with the ALGOL compiler. Names of these functions can be redefined by the programmer provided no ambiguity results from an attempt to use the identifier both as an ALGOL function and as a programmer variable. The function keywords are:

abs	cos	hbound	ln	sign
address	entier	index	memory	sin
arctan	exp	lbound	rotate	size
ascii	fix	length	setcurrent	sqrt
byte	float		shift	substr
classify				tan

★ ★ ★

CHAPTER 2 -- SCOPE OF IDENTIFIERS

Simple variables, arrays, labels, switches, and procedures are quantities which have a given scope. Scope is defined as the set of statements and expressions in which the declaration of the identifier associated with the quantity is valid.

Variables, Arrays, Switches, and Procedures

Variables, arrays, switches, and procedures must be declared, and their scope is the block in which they are declared. By extension, their scope includes inner blocks. An identifier is considered local to the block in which it is declared and global to any inner blocks, unless the identifier is redeclared in an inner block to represent a different quantity as shown in the example.

```
1:   begin
     real A,B;          ←A and B are real quantities, local
                         to block 1
  2:   begin            ←A and B are not redeclared in 2 and
                         are real quantities global to block 2
       end  2;
  3:   begin
       integer A,B;     ←A and B are redeclared as integer
                         quantities local to block 3.  (A
                         and B as real quantities are not
                         valid.)

       end 3;           ←When block 3 ends, A and B are
     end                 again real quantities local to
                         block 1
```

Labels

Labels may be explicitly declared by their use as a label in a given block. When a label precedes the start of a block (*begin*), the label is declared by its use in the block immediately <u>outside</u> the one that it serves to label and is global to the <u>block it labels</u>.

Labels (Continued)

```
A: ┌──── begin
   │B: ┌─begin            ←B is declared in block A.
   │   └─end  B;
   └──── end  A
```

Labels may appear in declarations. A formal parameter that is to be replaced by a label is declared with the *label* declarator. When the same label or subscripted label appears in more than one block of a program, the *label* declarator may be used to indicate that the local, rather than global, label is meant.

Parameters

Formal parameters that are replaced by name follow the scope conventions of variables. Note that no conflict arises when a formal parameter list is replaced by an actual parameter list containing one or more of the same identifiers but associated with different quantities. The actual parameters simply replace the formal parameters, which have no scope in the real sense of the term.

For example:

```
begin integer a,c; real b;
procedure sample (a,b,c);         }  procedure        }
real a; integer b,c;              }  declaration      }
statement of procedure;           }                   }  calling
  .                                                   }  block
  .
  .
  .
  .
sample (b,a,c);         ←procedure call               }
```

Actual parameters a and c replace formal parameters b and c; actual parameter b replaces formal parameter a.

SCOPE OF IDENTIFIERS (Continued)

Parameters (Continued)

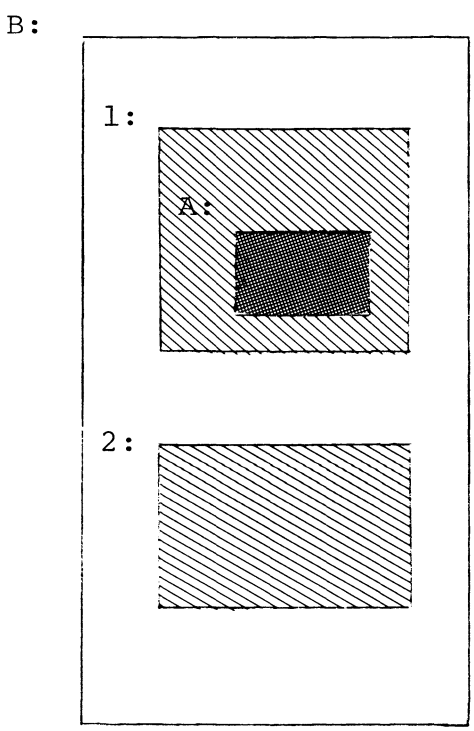
An actual parameter that replaces a formal parameter by value is not altered in the calling procedure because of the call. The called procedure uses a copy of the parameter during procedure execution.

Scope and Blocks

Storage for identifiers is normally allocated when the block in which the identifier is declared is entered. Storage is freed when control passes from the block in which the identifier is declared.

In the diagram below, presume that each rectangle represents a block and that the labels of the blocks are B, 1, A, and 2. Identifiers declared in block B are defined in all blocks unless a given identifier is redeclared in another block. Identifiers declared in block 1 are defined for blocks 1 and A unless redeclared in A.

Identifiers declared in block A are undefined in any other block; the same is true of the identifiers of block 2. Note that the labels of the blocks are clearly defined in the block outside the block for which they act as labels.



Identifiers declared in darker shaded blocks are undefined in lighter shaded blocks

SCOPE OF IDENTIFIERS (Continued)

Identifier Scope Not Associated with Blocks

If an identifier is defined with the *own* declarator, storage for the identifier is allocated in an area separate from block-dependent identifiers. The identifier is then valid until the program terminates.

External Identifiers

External variables and procedures are those that are allocated storage in a manner independent of any of the blocks of the program being executed. To reference an external identifier, that identifier must be declared *external* in the block in which it is referenced or in an outer block.

★ ★ ★

CHAPTER 3 -- BLOCKS

DEFINITION OF A BLOCK

In structure, a block is set of declarations and statements that starts with the keyword *begin* and terminates with the keyword *end*. Semantically, a single block is the smallest set of statements within which a given declaration of an identifier of a quantity is valid for that quantity.

Procedures are treated as blocks. Procedures usually contain one or more identifier declarations, and these declarations are local to the procedure.

Storage is allocated and deallocated to identifiers dynamically. When a block is entered, storage is allocated to those identifiers declared in the block. Storage for those identifiers is released when exit is made from the block.

Storage is not deallocated when a block inside a block is entered. For such a block, identifiers declared in the outside block remain valid, global quantities. However, the identifiers may be redeclared in an inner block to represent different quantities. If so, the block cannot reference the same identifier outside the block. For example:

```
A:__      begin
|  real X; integer i,j;        ←X is declared as a real quantity
|               .
|               .
|               .
|  Tag:  X :=X+sin(X);         ←Tag is declared a label in block A.
|               .
|               .
|               .
|  B:   ┌begin
|       │real array Tag[i,j]; ←Tag is redeclared an array in block B.
|       │    real Z;           ←Z is declared a real quantity in B.
|       │       .
|       │       .
|       │       .
|       └──── end B;           ←When block B terminates, Tag is
|               .               again valid as a label.
|               .
|               .
|    go to Tag;
|               .
|               .
|               .
└_ end A;
```

DEFINITION OF A BLOCK (Continued)

In the example, X is valid and can be referenced in block A and in block B; Z is valid and can be referenced only in block B; and Tag is valid only in A as a label and is valid only in B as an array. Note that the variable dimensions of Tag are valid as integer quantities in both blocks.

CONTENTS OF A BLOCK

Every identifier that is local to a given block must be declared within that block. This rule applies to all identifiers, including the controlled variable of a *for* statement and variables appearing on the lefthand side of assignment statements.

All identifier declarations must be made before any statement can be given in a block. (Comments, although sometimes considered to be statements, can appear before all declarations have been given.) It is important to note that ALGOL has a null statement consisting only of the terminating semicolon(;). An extra semicolon appearing in the declaration section of a block will cause declarations following the semicolon to be disregarded.

The statement section of a *begin* block can consist of a number of separate statements. A procedure always consists of one statement, which may be a *begin* block including other statements and blocks.

BEGINNING AND TERMINATING BLOCKS

ALGOL permits the keyword *end* to be followed by a string of characters that may include any characters except the keyword *else*, the keyword *end*, or a semicolon (;). This allows the programmer to describe preceding material. However, it also means that the keyword *end* terminates a block but does not allow the programmer to start a new block. For example:

```
     .
     .
     .
     end
 21: begin integer A,C;   ←everything up to the semicolon follow-
     real b;               ing C is simply a string following end,
     .                     i.e., there is no begin block labeled
     .                     21.
     .
```

To prevent errors, put a semicolon after the keyword *end* or after the string that follows the keyword, unless an *else* clause follows.

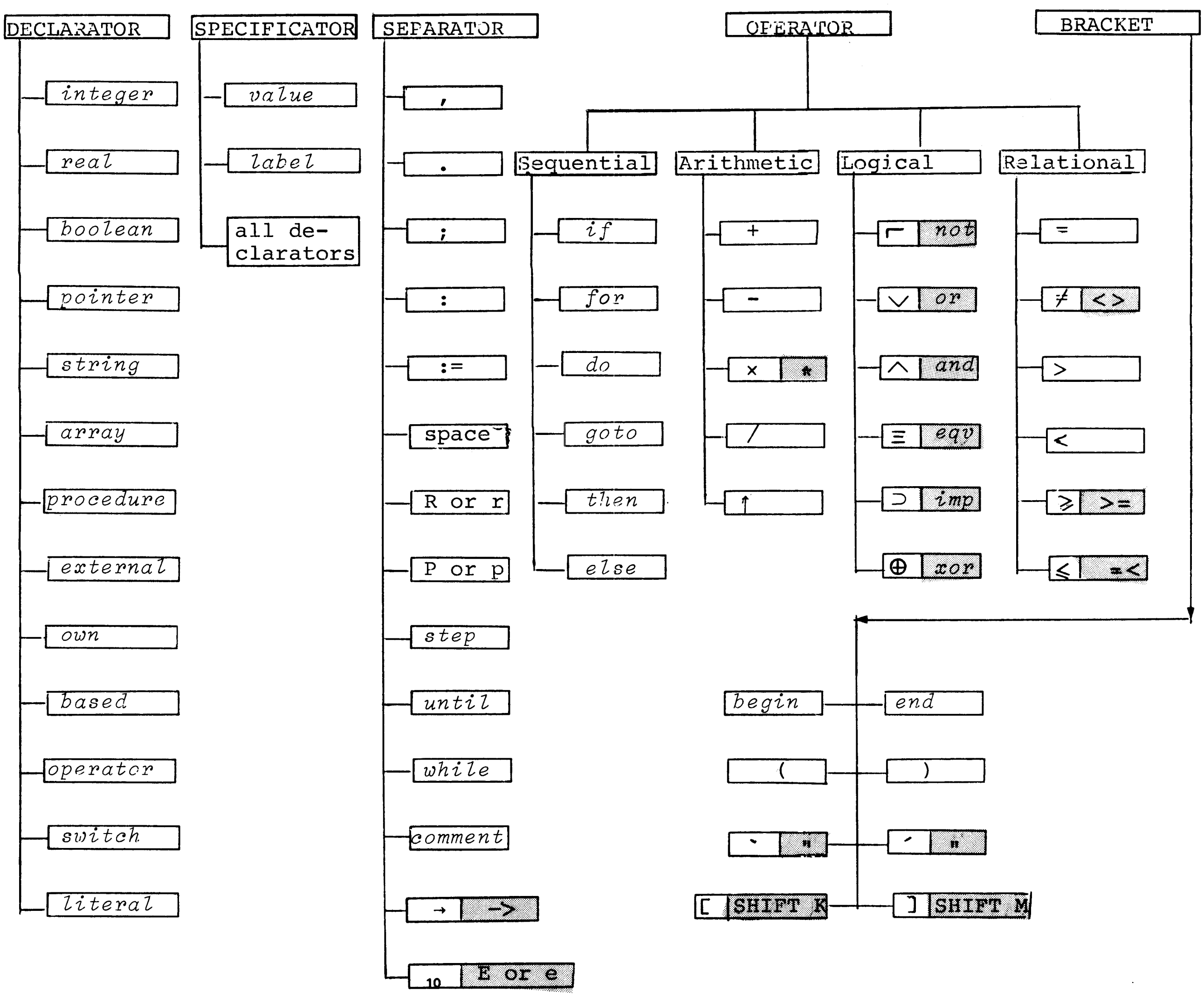
Since procedures contain only a single statement, they normally terminate with the semicolon that ends the statement.

★ ★ ★

CHAPTER 4 -- DELIMITERS

The ALGOL delimiters are separators, operators, declarators, specificators, and brackets, as listed in the table following. Since some ALGOL delimiters are represented by symbols that do not appear on all consoles, the appropriate transliteration for these characters is shown in the shaded area next to the character.

TABLE OF DELIMITERS



SEPARATORS

Symbol		
,	Separate items of lists.	*procedure* RT(a, b, c); *real array* A[i,jk,kj,k];
.	Decimal point in real numeric values.	0.011 2567.202E-6
$_{10}$	Separate base from power of a number, indcating a power of 10.	$25.2_{10}{-3}$ $.1_{10}{+5}$
;	Terminate a statement, declaration, or comment.	*integer array* D[1:20]; *go to* 101; *comment*: Transfer to test results;
:	Terminate label or separate the upper and lower bounds of an array dimension.	a: b: I:=I+2; *real array* A[1:10, 1:i];
:=	Separate a variable or variables from the expression to be evaluated and assigned to the variable.	c:= c+1; c:= d := f:=sin(x+1);
→	Separate a based variable from its pointer.	ptr→a p→(a+2)
(space)	Separate variables and keyword identifiers not otherwise separated.	*if* a=2 *then go to* 20 *else* a:=b;
R or r	Separate radix from number.	0.001R2 .555r8
P or p	Separate precision from number.	0.001R2P4 .555p5r8 -65.8888P3
step	Separate initial and incremental values of *for*.	*for* i:=1 *step* 2...
until	Separate incremental and terminal values of *for*.	*for* i:=1 *step* 2 *until* n ...
while	Separate conditional expression from value in *for* statement.	*for* I:=(x+2) *while* a≠0 ...
comment	Begin a comment.	*comment*: Test program;

BRACKETS

Symbol

Symbol		
( )	Parentheses enclose formal and actual parameters, enclose the precision of numerics and length of character strings, and enclose expressions to be evaluated.	*procedure* main (a,b,c); *string* (8) B; *integer* (12) *array* B[i,j]; ((A+B)/C)↑2.5
[ ]	Square brackets enclose the dimensions of arrays, subscripts of array elements, and subscripts of switches and labels.	*integer array* M[i,j]; c :=A[1,2]; *go to* B[i];
begin *end*	Keywords *begin* and *end* enclose blocks and compound statements.	*begin real array* act [0:20]; . . . *begin* act[m] :=j; k :=i *end* . . . *end*
\` ´	Grave (ASCII character 140_8) and acute (ASCII character 047_8) accents enclose string values. Note that strings can be nested.	\`This is a \`string´.´
" "	Double quotation marks can also enclose a string value. Use of a single accent mark is possible in a double quotation string. Strings enclosed in double quotation marks cannot be nested.	"DON'T GO!"

ARITHMETIC OPERATIONS

Operator	Operation	Resulting Value Type
+	Addition	If both operands are integer the result is integer. Otherwise, the result is real.
-	Subtraction	
×	Multiplication	
/	Division	
↑	Exponentiation	Permitted combinations and results are described in the table below for real and integer values.

Base	Exponent	Type of Result
integer = 0	real or integer ≤ 0	undefined
	real > 0	real 0.0
	integer > 0	integer 0
integer < 0	any real	undefined
	any integer	integer
integer > 0	any real	real
	any integer	integer
real = 0	real = 0	undefined
	real ≠ 0	real 0.0
	integer ≤ 0	undefined
	integer > 0	real 0.0
real < 0	any real	undefined
	any integer	real
real > 0	any real or integer	real

NUMBERS

Numbers are real or integer. Integers are signed or unsigned. Real numbers may be signed or unsigned, have an optional decimal point, and have an optional exponent part.

Integers	Real Numbers		
0	-200.845	0	$-9.3_{10}+02$
1775		1775	
-25	1.01	+606	$25_{10}-4$
+606		-25	
	+.0083		$_{10}+2$

Numbers having an integral power of 10 can be represented on the teletypewriter with either an upper case E or lower case e in place of the lowered 10.

ALGOL Representation	TTY Transliteration
$-976.33_{10}+02$	-976.33E+2
$25_{10}-4$	25E-4
$-_{10}7$	-1E7
$_{10}+02$	1E+02

Note in the third and fourth examples that a 1 appears before the E or e to prevent interpretation of the number as an identifier.

To approximate the number of decimal digits of precision that can be stored in a given number of 16-bit words, use the following formulas. $\underline{n}$ represents the declared precision in words.

integer digits $= 5(\underline{n}-1)+4$ integer range $= \pm 2^{16\underline{n}}-1$

real digits $= 5(\underline{n}-1)+2$ $10^{-78} \leq$ real range $\leq 10^{+75}$

The maximum value of a single precision integer is $\pm 32767_{10}$.

NUMBERS (Continued)

A number can be written with any radix from two through ten. The numeric literal is written, followed by the letter R (or r), followed by the number defining the radix.

1001R2	Base 2.	
.12122R3	Base 3.	
77E-6R8	Base 8.	The exponent is 8^{-6}, where the power, -6, remains base 10.
.3E+5R4	Base 4.	The exponent is 4^{+5}.

A number has the default precision of its type unless otherwise specified. When computation involves a multiprecision value and a fractional literal of default precision, results of computation lose precision because the fraction cannot be expressed exactly in binary representation. To control the precision of the computation, the programmer may specify a precision in words for the repeating binary fraction. The literal is followed by the letter P, followed by an integer representing words of precision.

.3P6

.111R2P6

1.7E-2P4

.1R3P7

To build a 16-bit single-precision mask, force a precision of one, e.g.,

177777R8	is a 2-word literal
177777R8P1	is an unsigned 1-word literal

BOOLEAN OPERATIONS

RELATIONAL OPERATORS			LOGICAL OPERATORS		
Symbol	Operation	TTY Symbol	Symbol	Operation	TTY Symbol
$<$	less than	same	$\neg$	logical negation	*not*
$\leq$	less than or equal	=<	$\wedge$	logical and	*and*
			$\vee$	inclusive or	*or*
$=$	equal	same	$\equiv$	equivalence	*eqv*
$>$	greater than	same	$\supset$	implication	*imp*
$\geq$	greater than or equal	>=	$\oplus$	exclusive or	*xor*
$\neq$	not equal	< >			

LOGICAL OPERATOR TRUTH TABLE							
Operands		Operations					
Y	Z	*not* Y	Y *and* Z	Y *or* Z	Y *imp* Z	Y *eqv* Z	Y *or* Z
false	*false*	*true*	*false*	*false*	*true*	*true*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*	*true*
true	*true*	*false*	*true*	*true*	*true*	*true*	*false*

RULES OF ARITHMETIC AND BOOLEAN EXPRESSION EVALUATION

The sequence of operations within an expression is generally from left to right, with the following additional rules:

1. Precedence of operator evaluation

OPERATOR	
$\rightarrow$	←Highest precedence (evaluated first)
⌐ ↑	
× /	
+ -	
≤ < ≠ > ≥ =	
∧	
∨	
⊃	
≡ ⊕	
:=	←Lowest precedence

2. ⌐ and ↑ operations are evaluated from right to left.

3. Parentheses are used to alter the order of operator precedence. A parenthesized expression is evaluated as an entity before further evaluation proceeds.

The type of the result is determined according to the rules of precedence, as follows:

first:	*real*
second:	*integer, pointer*
third:	*boolean*
fourth:	*string*

BIT OPERATIONS

Bit operations use binary and octal literals combined with logical operators to manipulate bits of integer data.

A ∧ B *(and)*	Result is 1 if and only if A is 1 and B is 1 in that bit position.	A :=11001R2; B :=10100R2; A ∧ B :=10000R2;
¬A *(not)*	Result is the bit complement of A.	A :=110011R2; ¬A :=001100R2;
A ∨ B *(or)*	Result is 1 if either A or B is 1 in that bit position.	A :=100111R2; B :=110000R2; A ∨ B :=110111R2;
A ⊕ B *(xor)*	Result is 1 if and only if A and B are complements in that bit position.	A :=100100R2; B :=001101R2; A ⊕ B :=101001R2;
A≡B *(eqv)*	Result is 1 if and only if A and B have identical bits in that bit position.	A :=100100R2; B :=001101R2; A≡ B :=010110R2;
A ⊃ B *(imp)*	Result is 1 if A is 0 in that bit position or if both A and B are 1 in that bit position.	A :=100100R2; B :=110001R2; A ⊃ B :=111011R2;

For example, assume x is some integer.

x :=x *and* 111111R2;	First 10 bits of x set to zeroes,
x :=x *and not* 777R8;	Last 7 bits of x set to zeroes,
x :=x *and not* 52525R8;	Alternate bits, beginning at bit 1, are set to zeroes,
x :=x *and* 52525R8;	Alternate bits, beginning at bit 0, are set to zeroes.

★ ★ ★

CHAPTER 5 -- EXPRESSIONS

The primary constituents of an ALGOL program - which represents algorithmic processes - are expressions. Expressions are arithmetic, Boolean, designational, or pointer.

Each type of expression may be either a simple expression or a conditional expression. Simple expressions are similar to expressions in other programming languages; conditional expressions are a unique ALGOL feature. In a conditional expression, one out of several expressions (arithmetic, Boolean, designational, or pointer) is selected for evaluation on the basis of the truth value of a Boolean expression in an *if* clause. An *if* clause has the form

if Boolean - expression *then* ...

Constituents of expressions (except for certain delimiters such as ( ) and [ ] and :=) are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, sequential and pointer operators. Expressions may be nested to any depth.

ARITHMETIC EXPRESSIONS

An arithmetic expression is a rule for computing a numerical value.

A simple arithmetic expression is a collection of one or more numbers, arithmetic variables and function designators combined with arithmetic operators to form a meaningful mathematical expression which always defines a single numerical value. Each variable of the expression must already have a defined value.

Examples:

A+B/f	x↑(k-4) × (y-z)	sum↑cos	(y+zx3)/7.394$_{10}$-8
c-d×g↑i	(-b+sqrt(d))2/a		

Real numbers are stored in floating-point and integers are stored in fixed point. An arithmetic expression consisting of a real value and an integer value will require conversion of the integer to floating-point. For example:

ARITHMETIC EXPRESSIONS (Continued)

```
begin real x;
      y :=x+1;          ←conversion required
          .
          .
          .
      y :=x+1;          ←no conversion required
```

A conditional arithmetic expression contains at least one *if* clause with a Boolean expression, two or more arithmetic expressions, and may contain other sequential operations besides *if* and *then*.

```
if g>0 then S+3×Q/A else 2×S+3↑q

if a<0 then U+V else if a×n>17 then U/V else if k≠y then V/U

A[i] :=  if i<j then B[j]+i
else B[j+1];
```

The subscripts of an array element may be given as simple or conditional arithmetic expressions whose value is an integer.

The length of a string or the dimensions of an array can be declared as simple or conditional arithmetic expressions evaluating to integers if the values of the variables of the expressions are defined when the block is entered.

```
A[n]  :=A[if y<0 then n else n+5];

real array A[i,j,k];
```

BOOLEAN EXPRESSIONS

A Boolean expression is a rule for computing a logical value (true or false).

Simple Boolean expressions are collections of logical values, Boolean variables and functions, and logical and relational operations. Relational operations consist of simple arithmetic expressions and relational operations..

Example: Assume that A:= *true*; B:=*true*; W:=2; X:=4; Y:=6;

Statement		Logical Value
D:= *not* A;		*false*
E:=W>X;		*false*
F:=W<X *and* W<Y;	(*true* and *true*)	*true*
G:=W≠X *and not* A;	(*true* and *false*)	*false*
H:=*not* A *or* W=X;	(*false* or *false*)	*false*
J:=*not*(A *and* W>X);	([*not*(*true* and *false*)] i.e., *not* *false*)	*true*

A conditional Boolean expression contains at least one *if* clause and two or more Boolean expressions, and may contain certain other sequential operators besides *if* and *then*.

```
if k<l then s>w else h<c

if(if(if a then b else c) then d else f) then g else h<k
```

POINTER EXPRESSIONS

A pointer expression is a rule for obtaining a pointer to an address.

A simple pointer expression is a pointer identifier or a subscripted pointer identifier, which may be combined with integer numbers or arithmetic expressions that evaluate to an integer using the arithmetic operators + and -.

A conditional pointer expression contains at least one *if* clause, two or more pointer expressions and may contain other sequential operators besides *if* and *then*.

A pointer expression is often followed by the pointer operator → and a based variable to which the expression points.

POINTER EXPRESSIONS (Continued)

p→a

if k<1 *then* (p+i)→a *else* (p+1)→a

p[i]→a

DESIGNATIONAL EXPRESSIONS

A designational expression is a rule for obtaining the label of a statement.

A simple designational expression is a label identifier, an unsigned integer used as a label, a subscripted label identifier, or a subscripted switch designator. The subscript of a label identifier or switch designator evaluates to an integer value.

A conditional designational expression contains at least one *if* clause, two or more designational expressions and may contain other sequential operators besides *then* and *if*. Conditional designational expressions cannot follow the keyboards *then* and *go to*.

17

p9

Choose[n-1]

TOWN[*if* y<0 *then* N *else* N+1]

if AB<c *then* 17 *else* q[*if* w≤0 *then* 2 *else* n]

★ ★ ★

CHAPTER 6 -- STATEMENTS

The statement is the basic operating unit of ALGOL. There are six kinds of statements:

NAME	EXAMPLE
assignment	i :=i+1;
conditional (*if*)	*if* i+0 *then go to* 25;
transfer (*go to*)	*go to* labelxx;
loop (*for*)	*for* i :=1 *step* 1 *until* n *do*...
procedure call	somefunction (x);
dummy or null	tag:;

Statements are executed consecutively unless the sequence is broken by an unconditional transfer (*go to* statement) or by some condition that causes a statement sequence to be skipped (*if* statement). Statements may have one or more labels.

Basic statements are often combined to form more complex units of operation, for example, the following combination of assignment, condition, transfer and looping statements:

```
if i>0 then for i :=1 step 1 until n do A[i] :=B[i]+i else go to 25;
```

Each statement within the combination of statements may be labeled:

```
T1:if i>0 then T2:for i :=1 step 1 until n do
T3:A[i] :=B[i]+i else T4: go to 25;
```

A further level of freedom in statement sequencing is available. A group of statements can be delimited by *begin* and *end* keywords forming a compound statement. A compound statement is a block in which there are no declarations.

STATEMENTS (Continued)

```
Z:  begin integer i,k; real w;
    for i :=1 step 1 until m do
    for k :=i+1 step 1 until m do
        begin w :=A[i,k]; A[i,k] :=A[k,i];   } Compound     } Block
              A[k,i] :=w end i and k;        } Statement
              .
              .
              .
        end Z;
```

Note that a compound statement can contain other compound statements.

Conditional expressions, which can be used whenever a simple expression can be used except following the keywords *then* and *go to*, provide another degree of freedom. Such constructions as:

```
if(if...then...else...)then...
```

are permitted in ALGOL.

ASSIGNMENT STATEMENT

Format:

```
v :=e;
```

where: v is a variable or list of variables.

e is an expression.

Purpose: To assign the value of the expression on the righthand side of the statement to the variable or list of variables on the lefthand side.

Notes:

1. v may be a subscripted variable.

2. v may be a procedure identifier if the assignment statement appears in the body of the function that defines the procedure identifier.

3. A list of variables on the lefthand side has the format:

```
v1 :=v2 :=...vn
```

Variables in the list need not have the same data type. The expression is converted to match the data type of each variable, starting at the rightmost. Conversion is made according to the rules given below.

4. The following data type conversions are permissible:

```
integer v := boolean e;
```

The *boolean* expression is evaluated to 0 or 1. A full word of either 0's or 1's is assigned to v.

```
boolean v := integer e;
```

ASSIGNMENT STATEMENT (Continued)

The *integer* expression is evaluated. If the expression has a value of 0, the value *false* is assigned to the variable; otherwise, the variable is assigned the value *true*.

```
integer v := pointer e;
```

A pointer expression evaluates to an integer that is one word long and points to some location. The pointer value can be assigned to an integer variable if the variable has the default precision of one word.

```
pointer v := integer e;
```

The value of the *integer* expression is assigned to the *pointer* variable. The integer must be of default (one word) precision.

```
real v := integer e;
```

The *integer* expression e is evaluated and a decimal point is placed after the last digit when assigning a *real* value to v.

```
integer v := real e;
```

The *real* expression is evaluated. The value assigned to the *integer* variable is entier(e+0.5). See the built-in function entier.

```
string v := integer e;
```

The *integer* expression is evaluated and assigned to *string* v as a string of characters of the form: [-]nn...n where each n is a digit.

ASSIGNMENT STATEMENT (Continued)

integer v:= *string* e;

Characters of the string expression will be assigned to the value v up to the first non-integer character, such as a decimal point. The precision of v governs how many characters will be assigned. An acceptable form of string is: [-]nn ...n where each n is a digit.

string v:= *real* e;

The *real* expression is evaluated and assigned to *string* v as a string of characters of the form:

[-]nn...n[.nn...n] [E[-]nn]

where each n is a digit and bracketed portions of the form are optional.

real v:= *string* e;

The *string* expression is evaluated. Characters of the string will be assigned to the value v up to the first non-real character or up to the limit of the precision of v. The acceptable form of string is shown above for *real* to *string* conversion.

string v := *boolean* e;

The *boolean* expression is evaluated to a zero or one (*false* or *true*). The zero or one is assigned to the string v.

boolean v := *string* e;

The *string* expression is evaluated. The result will be assigned to v as *false* (zero) if the string contains all zeroes. Otherwise the value *true* (one) will be assigned.

ASSIGNMENT STATEMENT (Continued)

string v := *pointer* e;

The *pointer* expression is evaluated. The result, having the form: nn...n, will be assigned to v.

pointer v := *string* e;

The *string* expression is evaluated. The result is assigned to *pointer* v up to the first non-digit or up to the one-word limit of the pointer.

Examples:

```
S[a,k+2]  := 3-arctan(S×zeta);
```

The lefthand subscript is first evaluated, the arithmetic expression is evaluated and assigned to S[a,k+2].

```
T  := AIJ  →  N;
```

The pointer expression is evaluated and assigned to pointer T or T may be an integer of default precision.

```
string STR(20); real x;
integer i; pointer  p;
boolean b; literal STR ("$2504.25 FOR 12")
    .
    .
    .
x := substr(STR, 2,8);        x contains 2504.25
i := substr(STR, 3,6);        i contains 504
b := substr(STR, 4).          b contains false
p := substr(STR, 14,15);      p contains 12
```

The substr function, as described in Chapter 9, takes a substring of a string from the character whose position is given in the second parameter through the character whose position is given in the third parameter. If the third parameter is not present only a single character forms the substring. Note that only allowable characters are converted and assigned; in the second assignment; i will contain only 504, and the character in character position 6 (.) is ignored and processing ceases when such a character is encountered.

```
Boo :=b>c and d;
```

A truth value is assigned to Boo when the Boolean expression b>c *and* d is evaluated.

```
p :=address (f);
```

The pointer p is assigned the address of f.

```
Formula :=diff/ (x - 2);
```

Formula is a function procedure and the assignment statement appears as the body of the function.

for STATEMENT

Format:

```
for cv :=list do s;
```

where: cv is a controlled variable, which may be subscripted.

list is a list of values the controlled variable can assume.

s is a simple or compound statement.

Purpose: To permit repetitive execution of statement s with the controlled variable set to values specified by list.

for STATEMENT (Continued)

Notes: 1. list may be a simple list of values or expressions to be evaluated. In addition, list can include *for* clauses. A *for* clause contains either keywords *step* and *until*, or the keyword *while*.

```
for i :=1   step 1   until 10 do A[i]   :=i↑i;

        ↑        ↑          ↑
     initial increment  final
     value              value limit
```

The example above is equivalent to the simple list:

```
for i :=1,2,3,4,5,6,7,8,9,10 do A[i] :=i↑i;
```

Values of the list are assigned to i beginning with the leftmost value and terminating with the rightmost value. When the list is exhausted, the next statement in logical sequence will be executed.

A *while* construction is shown in the statement:

```
for j :=0, 1, v×2  while v<n do m:=j/5;
```

Note that the *while* construction is included as part of a simple list. A list may include any number of *for* clause constructions. For example:

```
for j :=i+k,2,i+2,1 step 1 until n, x while x≠0 do...
```

Notes: 2. The statement following *do* may be a *for* statement, or a compound statement that includes a *for* statement, i.e., *for* statements may be nested.

3. Parts of a *for* statement may be labeled, but an attempt to transfer to a label within a *for* statement from outside the statement will cause an undefined result.

for STATEMENT (Continued)

Examples:

```
for I := 1 step 2 until n do
    X[I] := X[I] ↑2+I;

for k := 0,n do u[k] := u[k]/2;

for a[bottom] :=min(a[bottom], a[top]) while top>bottom do
    begin top :=top-1;
          bottom := bottom+1; end;
```

go to STATEMENT

Format:

go to d;

where: d is a label or designational expression.

Purpose: To transfer to the statement having the label d.

Notes: 1. Transfer cannot be made from outside a block into the block. Transfer can only be made to labels defined locally or globally in the block containing the *go to*.

2. Designational expressions may be:

   a. Labels with a variable subscript.

   b. Switches.

3. If the value of a switch or a label subscript expression is undefined, no transfer occurs and the statement following the *go to* is executed. (A switch is undefined if the value is greater than the number of labels declared for the switch or is less than or equal to Ø. A label subscript expression is undefined if it evaluates to a subscript for which there is no matching label.)

go to STATEMENT (Continued)

Examples:

```
go to 10;
```

Transfer is made to the statement labeled 10.

```
go to a[i];
```

i is evaluated and transfer is made to the appropriate subscripted label, a[i], a[2], ...

```
switch F :=labone, x1, labtwo, x2;
    .
    .
    .
go to F[j];
```

If j evaluates to 1, transfer is made to the statement labeled labone; if j evaluates to 2, transfer is made to the statement labeled x1, etc.

if STATEMENT

Format:

```
if be then uc;
if be then uc else c;
if be then uc else if ...
```

where: be is a Boolean expression.

uc is an unconditional clause, which may be a statement, compound statement, or block, but cannot contain another *if* clause.

c is any clause, which may be a statement, a compound statement, or a block.

Purpose: To provide conditional transfer of program control. If the Boolean expression be is true, the unconditional *then* clause is executed. If be is false, the next statement or block following the

Purpose: unconditional clause is executed. This may be the next statement or block following a semicolon (Format 1) or the statement or block following the keyword *else* (Format 2).

Since *else* clauses may contain conditional statements (Format 3), it is possible to set up a series of conditions for transfer of program control. The series terminates when a Boolean expression is true, causing a *then* clause to execute.

Blocks and statements contained in *then* or *else* clauses may be labeled.

Examples:

```
if i=0 then go to END_PROG;
```

```
if j<kt then begin
 k := factor[j]+i;  j := j+i;
 lab7:  i:= i+1;  S[i]:= j;go to 5;
end lab7 else go to 15;
```

```
if g<0 ∧ h<0 then isign := -1 else
if g>0 ∧ h<0 then isign := +1 else 0;
```

★ ★ ★

CHAPTER 7 -- IDENTIFIER DECLARATION AND MANIPULATION

Programmers must declare the characteristics of all identifiers to be used in a program. Keyword declarators and certain bracketed information are used to define identifier characteristics.

The characteristics that can be declared for identifiers are their shape, data type, storage class, and precision. Appendix B explains how declaration of these characteristics is used by the compiler to generate parameter descriptor code which, in turn, provides information for allocation and freeing of identifier storage.

SHAPE OF IDENTIFIERS

The four possible shapes of an identifier are scalar, array, procedure, and program. The default shape is scalar and need not be explicitly declared. Program identifiers are recognized as such by the compiler and need not be declared. Arrays are declared with the keyword *array*, and procedures are declared with the keyword *procedure*. The keyword *operator* is used to declare a special kind of procedure.

DATA TYPE OF IDENTIFIERS

There are six possible identifier data types -- integer, real, boolean, string, pointer, and label. All identifiers except labels must be declared with one of the keyword identifiers, *integer*, *real*, *boolean*, *string*, *pointer*, or *label*. A *label* declarator is required for a formal parameter that will be replaced by a label. The *label* declarator may also be used to identify a local from a global label of the same name. However, the appearance of a label preceding a statement usually constitutes its explicit declaration as a label.

STORAGE CLASS OF IDENTIFIERS

The storage classes of identifiers are local, own, based, parameter, value, external, built-in function, and function value. The default storage class is local and need not be explicitly declared. A local identifier is one that is allocated when the block in which it is declared is entered and freed when the block is exited.

The storage classes that can be explicitly declared by the programmer are *own*, *based*, and *external*. Identifiers that are declared with the *literal* declarator have the storage class, value. Formal parameters, built-in functions, and function values are recognized as such by the compiler and are not declared.

PRECISION OF IDENTIFIERS

Default precision for identifiers and the declaration of precision are described in relation to storage in Appendix A. Precision is declared as an integer literal enclosed in parentheses immediately following the data type declarator. Precision can be declared for numeric identifiers, *integer* and *real*, where precision represents words of storage. Precision may also be declared for strings, where precision represents maximum number of characters that the string may have.

DATA TYPES

The data type declarators are *real*, *integer*, *string*, *boolean*, *pointer*, and *label*. They are mutually exclusive. Data types apply to all identifier shapes except those procedures that are not functions.

A *real* declarator declares a scalar, array, or procedure that returns a number value that is not an integer. Default storage of real values is two words. Maximum precision is 15 words.

```
real n, pi, m;
real array a, b, c[i,j];
real procedure X;
real(3) y;
real (4) array z[2,5];
```

An *integer* declarator declares a scalar, array or procedure that returns an integer numeric value. Default storage is integer values is one word. The limit of default integer values is $\pm 2^{15}-1$. Maximum precision of a multi-precision integer is 15 words.

```
integer array A[i,j];
integer i,j;
integer (4) q, r;
integer (2) procedure XX;
```

DATA TYPES (Continued)

A *string* declarator declares a scalar, array, or procedure that returns a character string value. Default storage of string values is 32 characters. Strings have a maximum length of 16,283 characters.

```
string (200) char;
string procedure sym (x,y);
string (20) array mt[10];
```

A *boolean* declarator declares a scalar, array, or procedure that returns a truth value of *true* or *false*. A boolean value is always stored in one word.

```
boolean zero, nosolution;
```

A *pointer* declarator declares a variable array, or procedure that returns an address as its value. A pointer value is always stored in one word.

```
pointer array LOCUS [8];
pointer p1, p2, p3;
```

A *label* declarator declares a scalar or array that returns a value that is an address. A label value is always stored in one word.

```
label tag[10];
```

ARRAYS

An array is declared with the explicit shape *array*, and one of the data types, real, integer, boolean, string, pointer or label.

Precision and storage class may be declared if other than default characteristics are wanted.

In addition, the identifier of the array is followed by dimensioning information, enclosed in brackets. The bracketed information consists of a list of subscript bounds of the general form:

$$\boxed{\underline{sb}_1, \underline{sb}_2, \ldots, \underline{sb}_n}$$

The following rules apply to array subscripts:

1. When a subscript bound consists of a pair of values or expressions, separated by a colon, the first value or expression gives the lower bound and the second value gives the upper bound.

2. If a single value or expression is given as a subscript bound, it represents the upper bound and the lower bound is assumed to be 0.

3. Up to 128 subscript bounds can be given in the list.

4. If an integer expression containing a variable is used in array dimensioning, the variable must be global to the block in which the array declaration appears.

5. The outermost block of a program must have only integer constant subscript bounds, unless it is a procedure with array formal parameters.

6. During execution, subscripts are checked against declared subscript bounds, and an error message results if the subscript exceeds the possible bounds.

7. The lower subscript bound must be smaller than the upper subscript bound.

8. Negative subscript bounds are permitted.

9. *own* arrays can have variable dimensions; however, the total size of the array is bounded by the original dimensions.

ARRAYS (Continued)

Examples:

```
integer array ORG[-10:10,0:20];

pointer array pp[9];

real (3) array A[i,j,k];

own string (5) array NAME[14];

integer array Z[0:i, i:i+5,7,j];
```

In the examples:

1. ORG is a 21x21-element integer array of default precision.

2. pp is a 10-element pointer array.

3. A is a 3-dimensional real array with 3-word precision. The upper subscript bounds, i, j, k, must have been defined in an outer block or must be formal parameters to be replaced by integer values.

4. NAME is a 15-element string array. Each element has a maximum length of 5 characters. *own* storage is used for the string.

5. Z is a 4-dimensional integer array. Note that some subscript bounds are paired while others are not, and that a pair of subscript bounds may contain a constant and an expression.

A number of array identifiers can be included in a single declaration; for example:

```
real (3) array a,b,c,d[1:5, 0:9];
```

where a, b, c, and d are all identifiers of real 2-dimensional arrays of 50 elements.

ARRAYS (Continued)

Each element of an array is a subscripted variable of the form:

array-name [sub_1,sub_2,...,sub_n]

where: array-name is the name of the array.

each sub is an integer value or expression giving a subscript of the array. If the subscript is real, it is converted to type integer by the function: entier (sub-value+0.5).

For example:

```
A[25] B[i,j] C[x+10] D[2,3,4,1]
```

could all be array elements.

The first subscript of an array varies most rapidly, then the second, then the third, etc. For example, if the 360-element array X is declared as:

```
real array X[3,5,4,2];
```

then the values are stored in the following order:

```
1.   X[0,0,0,0]
     X[1,0,0,0]
     X[2,0,0,0]
     X[3,0,0,0]
5.   X[0,1,0,0]
         .
         .
         .
8.   X[3,1,0,0]
         .
         .
         .
357. X[0,5,4,2]
         .
         .
         .
360. X[3,5,4,2]
```

ARRAYS (Continued)

The address of each array element may, if desired, be accessed by pointer manipulation.

The most common use of arrays is in loop manipulation. See *for* statement.

CHARACTER STRINGS

Scalars, arrays, and procedures may be declared with the *string* data type. By default, the precision of a character string is a maximum of 32 characters. The maximum length that can be declared for a string is 16, 283 characters. Examples of string declarations are:

```
string (10) a;
string (20) g,h,i;
```

String a has a maximum of 10 characters, beginning at character position 1. Strings g, h, and i each have a maximum of 20 characters, beginning at character position 1.

String literals are delimited by accent marks (ASCII characters 140_8 and 047_8) or by quotation marks.

```
`$25.00 FOR EACH´
"One Hundred"
```

String literals in accent marks may be nested to any depth.

```
`He said:  `This `string´ is nested.´´
```

A null string may be assigned to a string variable.

```
g:="";
```

CHARACTER STRINGS (Continued)

When a programmer writes a long literal string that requires two or more lines, the carriage returns at the end of each line are invisible and do not require a character position.

Control characters, such as the carriage return and form feed, can be passed as text directly to the assembler, using the
The octal code of the ASCII control character is enclosed in
The ocatal code of the ASCII control character is enclosed in the angle brackets and will be passed directly to the assembler without interpretation by the compiler. For example:

```
"THE END <15>"             ←015 is the octal code for carriage
                            return.
```

Subsets may be taken of strings using the built-in function, substr.

```
string (9) x;

x:="A10=$1.25";
    .
    .
    .
substr(x,1,9)              ←evaluates to the entire string.
substr(x,1,3)              ←evaluates to A10.
substr(x,5,9)              ←evaluates to $1.25.
substr(x,4)                ←evaluates to =.
```

The second parameter of substr gives the position of the starting character and the third parameter gives the position of the last character.

An array of character strings can be declared. Each element of the array must have the same maximum length. For example:

```
string (2) symb[1:100];  ←each element of symb has a maximum
                           length of two characters.
```

Each element of a string array can be subset using the function, substr.

CHARACTER STRINGS (Continued)

```
string (30) a;
string (3) array b, c[1:25];
a:="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   :
for i:= 1 step 3 until 24 do begin
b[i] := substr(a, i, i+2);
c[i] := substr(b[i], 2,3); end
```

Contents of the array elements after the *for* statement is executed will be.

```
b[i]:="ABC";b[4]:="DEF"; b[7]:="GHI",...b[19]:="STU";b[22]:="VWX";
c[1]:="BC"; c[4]:="EF";  c[7]:="HI"; ...c[19]:="TU"; c[22]:="WX";
```

Two other built-in functions are commonly used in string manipulation. These are the length function and the index function. The length function has a string variable as a parameter and returns the number of characters in the string as a value.

The index function searches a specified string variable (parameter 1) for a given character configuration (parameter 2) and returns as a value the starting location in the string of the first character of the configuration.

Examples:

```
string (4) v;
v:="abcd";
i:=length(v);            ←i:=4;
j:=index (v,"cd");       ←j:=3;
```

Some examples of how strings may be used are shown in the following examples:

CHARACTER STRINGS (Continued)

```
comment:   Pattern Match and Replacement;

i := index(a," ");
for i :=i+1 while substr(a,i+1) ≠" " do
substr(a,i) := "*";

comment:   Search the string a for some character delimited
           by blanks and replace the character with an
           asterisk character;
```

```
comment:   Editor Command Table Lookup;

external string (1) procedure Readchar;
string (10) commands;
switch S := Top, Search, Append, Insert;
commands := "TSAI";
loop:  i := index(commands, (Readchar));
         comment: Readchar is a function that
                  reads a character;
         go to S[i];
         error("illegal command");
         go to loop;
```

LABELS

Begin blocks and statements (including statements within compound statements) may be labeled. Declarations cannot be labeled. A label appears as either an identifier or an unsigned integer, delimited from the statement or block by a terminating colon (:) . A block or statement may have more than one label, each of which has a terminating colon. The appearance of an unsigned integer or an identifier followed by a colon constitutes an explicit declaration of that integer or identifier as a label.

LABELS (Continued)

```
begin
  .
  .
15: A1: x:=x+1;             ←15 is an integer label and A1 is an
  .                          identifier label.
  .
go to 15;                   ←transfer to the assignment statement.
  .
  .
go to A1;                   ←transfer to the assignment statement.
  .
  .
end of block;
```

A label is declared in its smallest enclosing block.

```
B:  ┌── begin real S;
    │       .
    │       .
    │ A: ┌── begin real Z;   ←A (like S) is declared in Block B
    │    │       .             and is valid in both blocks A and
    │    │       .             B.
    │    │   x: Z:=Z/S;      ←x (like Z) is declared in Block A
    │    │       .             and is valid only in Block A.
    │    │       .
    │    └── end A;
    └───     end B;
```

Labels can be declared with the *label* declarator. A *label* declarator is often used to identify a label that is not otherwise known in the block in which it is referenced.

LABELS (Continued)

```
          begin integer i; label error; ←error declared as label
           .
           .
          begin real x;
           .
           .
           .
          if x = 0 go to error;          ←transfer outside of block
           .                              to error.
           .
           .
          end;
           .
           .
           .
 error:
           .
           .
           .
          end;
```

A *label* declarator is also used to insure that transfer of control will be made to the correct label whenever a possible ambiguity exists.

```
          begin integer i;
           .
           .
           .
 error:
           .
           .
           .
          begin real x; label error;
           .
           .
           .
          go to error;
           .                 transfer made to label error in the
           .                 block in which it is declared label.
 error:      ◄
           .
           .
           .
          end;
           .
           .
           .
          end;
```

LABELS (Continued)

A dummy statement may be written in ALGOL. A dummy statement provides only a label to which a transfer can be made. For example, a transfer can be made to a labeled *end* delimiter terminating a compound statement or block.

```
begin integer j;
    .
    .
    .
if j = 0 then go to Z;
    .
    .
    .
Z:  end;                    ←labeled end
```

An identifier label may be subscripted with a simple integer subscript. If a block contains ten labels, a[i],a[2],...,a[10], execution of the statement

```
go to [j];
```

causes j to be evaluated and transfer to be made to the corresponding statement label. If j evaluates to a value outside the range of statement labels, e.g., 25, then the next consecutive statement after the *go to* is executed. Numeric labels cannot be subscripted.

Formal parameters of procedures are declared with the *label* declarator if a label is to be passed replacing the parameter.

```
procedure ALPHA (x,y,n,exit); label exit;
real x,y; integer n; value n;
```

Formal parameter exit will be replaced by a label when ALPHA is called.

SWITCHES

Switches are variables that identify a number of alternate labels to which program control may transfer. A switch is declared with a list of labels and designational expressions. The position occupied by a label or designational expression in the list determines whether that label is the one to which transfer is made.

Examples:

```
switch TESTPROG:=a,b,if x>0 then i else d,10,5,c,8,op,3,y3;
```

Switch TESTPROG is defined with 10 alternate labels or expressions evaluating to a label, where a has a position value of 1 and y3 has a position value of 10. If the following statement is encountered during execution:

```
go to TESTPROG [j];
```

j is evaluated. If j=2 transfer is made to label b; if j=3, transfer is made to either label i or d, based upon the evaluation of the designational expression.

```
switch SF:=a,bl,bw,c,d,7;          ←declaration of switch SF
       .
       .
       .
    go to SF [i];                  ←transfer to one of the labels
```

In this example i will be evaluated. If i=1, transfer is made to the statement labeled a, if i=2, transfer is made to the statement labeled bl, etc.

If a switch variable evaluates to a value that is outside the range of the switch, the next statement after the *go to* is executed. For example, in the second example, there are 6 possible values for i: 1,2,3,4,5 or 6. If i evaluates to a larger integer, the next statement after the *go to* is executed.

own DECLARATOR

Storage for a block is dynamic. Identifiers declared within a block are allocated storage when the block is entered, and storage is released at the time of exit from the block. If a block is entered more than once during execution of a program, variables will be undefined each time the block is entered.

The *own* declarator allows the programmer to specify a variable or variables whose value at the time of exit from the block will be retained. When the block is subsequently reentered, *own* variables are defined.

Example:

```
a:  begin integer i, j; own real Hs, s;
        :
        .
    end
```

Each time block a is entered, variables i and j are undefined. However, after the first execution of a, variables Hs and s have a specified value each time a is entered, the values being that of Hs and s at the time the block was last exited.

external DECLARATOR

Variables may be external to a given program. Such variables must be stored in an external area by assembly. They can be used in a given program if the external variable is declared *external* in the program in which it is used.

Example:

```
al:     begin external integer k;
              integer i, j;
        .
        .
        .
        end al;
```

POINTERS AND THE BASED DECLARATOR

Use of pointers and based variables is a programming technique which allows the systems programmer to achieve a very high level of object code efficiency.

In most high level languages certain information is available to the programmer that is not available to the compiler through the source program. The compiler must always assume the "worst case" in order to generate safe code.

For example, any subprogram call can potentially redefine all external variables. An assignment to any element of an array will force the compiler to assume that all values in the array have been modified. In the case of arrays passed as parameters, the compiler must generate "worst case" code for computing subscripts, since neither the bounds, precision, nor number of dimensions may be known until run time.

Pointers and based variables provide a mechanism for explicitly manipulating machine addresses. Using this facility, the programmer can, for example, force a subscript calculation to be performed only once in a frequently executed part of his program. As another example, if the programmer knows that an external variable will not be modified by a call, he can use pointers and based variables to convey this knowledge to the compiler.

The programmer declares an identifier, called a pointer. The pointer's value is the address of some program variable. Pointer expressions are allowed, so that address offsets can be given. When the pointer is used, it points to a *based* variable with the operator →; in effect, the pointer and *based* variable have been substituted for the precise address the programmer wants.

A single pointer can be reset to point to different program variables within a program. There are several ways in which a pointer can be set to a given program variable: use of the address function, use of the allocate procedure or simple assignment.

The declared *based* variable has all the characteristics of the program variable except for storage. That means that the data type of the based variable should match that of the program variable.

POINTERS AND THE BASED DECLARATOR (Continued)

In the example,following, y is declared as a *real based* variable and can be used, together with pointer p, to perform address modification involving either *real* program variable x or z.

```
begin real x,z;
based real y;
   pointer p;
   .
   .
   .
p :=address (x);          ←address function used to set pointer
   .                       p to the address of x
   .
   .
p→y:=p→ y+2;              ←statement is equivalent to x :=x+2;
```

The *based* variable can be considered a template of the program variable. As long as the pointer is set to x, the pointer and *based* variable can be used to modify the address. In this way the programmer can perform address modification and manipulation at very little cost in code generation.

The pointer can be reset to z, and the *based* variable can then be used in a similar way, representing program variable z.

Example:

```
B:  begin real x,z; based real y;  ←x,y, and z are all real. y
 pointer p;                         is declared based.  p is de-
       .                            clared a pointer
       .
       .
  11: p :=address (x);             ←pointer p is assigned the
       .                            address of x.
       .
       .
  22: p→y :=p→ y+2 ;               ←based variable y is super-
       .                            imposed upon x.  Statement
       .                            22 is the equivalent of
       .                            x :=x+2;
       .
  33: p :=address (z);             ←p is reassigned the address
       .                            of z.
       .
       .
  44: p→y :=p→ y+3;                ←based variable y is super-
                                    imposed upon z.  Statement
                                    44 is equivalent to z :=z+3;
```

POINTERS AND THE BASED DECLARATOR (Continued)

The program variable referenced by the pointer can be a simple variable, an array, or an element of an array.

```
begin pointer a;  integer array b; integer i;
      based integer x;
         .
         .
         .
   a:=address(b[i]);                  ←a is assigned the address
         .                             of array element b[i].
         .
         .
   b[i+1] := a→x;                     ←the statement is equivalent
                                       to:  b[i+1]  := b[i];
```

Assignment of a pointer to the address of a program variable made without using the address function is shown in the example below.

```
begin pointer A; real c; based integer b;
         .
         .
         .
A := address(c);
         .
         .
         .
 (A+1)→b := 0;                        ←location c+1 is set to 0.
```

Pointer arrays may be declared and pointer expressions may be used in address manipulation.

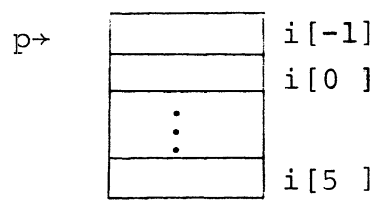
```
begin pointer array A[n];
based pointer array B[n];
based integer i;
         .
         .
         .
p := A[5]→B[4];                ←pointer array element A[5] points to a
         .                      based pointer array element B[4].  The
         .                      pointer value assigned to p can later
         .                      point to another based variable such
y := p→i;                       as i.
```

POINTERS AND THE BASED DECLARATOR (Continued)

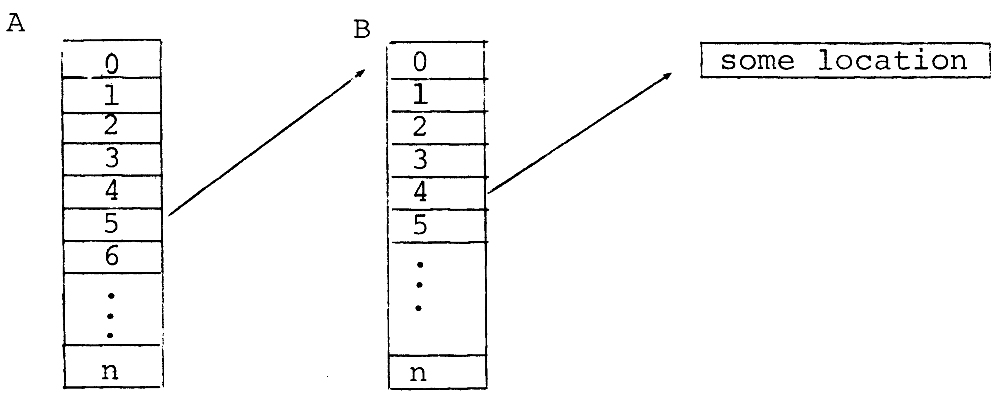
When using a *based* array, it is assumed that the pointer always points to the first word of data in the array, e.g.,

```
based array x[-1:5];
     .
     .
     .
```

if p→i[0]



A diagram of the arrays of pointers indicates the assignment of p in the statement, p:=A[5]→B[4];



A pointer can be set to a given number of words using the allocate procedure.

```
integer L, M;
pointer IL, IU,ILM,IUM;
based integer N;
     .
     .
     .
allocate(IL,8);       ←8 words of storage pointed to by IL
allocate(IU,8);       ←8 words of storage pointed to by IU
     .
     .
     .
ILM := IL+M;
IUM := IU+M;
```

POINTERS AND THE BASED DECLARATOR (Continued)

Use of pointers can be shown in list processing. Suppose the programmer wishes to search a singly threaded list, list_x, for a location called key.

```
begin pointer list_x;              ←a pointer and a default-precision
based integer i;                    integer are interchangeable.
      .
      .
      .
p := address(list_x);
      .
      .
      .
LOOP: if((p:=p→i)=0) then go to EXIT         ←if key does not exist.
      else if (p+1)→i=key then go to EXIT    ←if key is found.
      else go to LOOP;
      .
      .
      .
EXIT: ---;
```

LITERALS

Literals are identifiers that are declared with a given value. They provide a means of generating constants with names, so code will be efficient and all occurrences of a constant may be modified in one place. For example:

```
begin
literal MAX(100);            ←100 will replace all occurrences
literal Size (MAX);           of MAX in the block.  If the paren-
integer array X [0 :MAX];     thesized value is changed, all
    .                         occurrences of MAX will be changed.
    .
    .
```

Literals adhere to block structure. A literal declared in an outer block will be local to that block and global to all inner blocks in which the literal declaration is unchanged.

An identifier declared with a literal value in an outer block can be redeclared with another value in an inner block.

```
begin literal R(0);
    .
    .
    .
begin literal Z(0), R(1);
```

Any legal value may appear in a literal declaration.

```
begin literal Y (true), s("A-1023"), oct(-15R8), z(.01P4);
```

It is convenient to use literals to supply formatting information for output procedures. Several examples are included in the sample programs in Appendix F.

OPERATORS

An identifier can be defined as an operator having a given precedence. Operators are given the data type of the value to be returned as a result of the operation. In effect, use of an operator in a statement is identical to a reference to an external function procedure, as described in the sections following on procedures.

```
external string (100) operator (+) cat;
```

In the example, cat is declared an operator that returns a string value having up to a maximum of 100 characters and which has the same precedence value as the operator, +.

If cat is to be an operator used in concatenating strings, an external procedure must be set up for cat, defining the concatenation function and providing formal parameters that constitute the return value, and the two operands.

```
begin external string (100) operator (+) cat;
string s;
s:="ABC" cat "DEF" cat "GHI";     ←references to cat
end;

procedure cat (a,b,c);            ←the formal parameters must be
string a,b,c;                      positioned so that the first
begin a:=b;                        represents return value, the
substr (a,length (a)+1,            second the first operand and
  length(a)+length(b)):=c;         the third the second operand.
end;                               All procedures representing
                                   operators follow this format.
```

When the assignment statement is executed, control is transferred to procedure cat. For the first concatenation, "ABC" replaces b and "DEF" replaces c. The result, returned in place of a, is concatenated with "GHI", and the result returned to s.

★ ★ ★

CHAPTER 8 -- PROCEDURES

A procedure is a block of code that is executed only when it is called from another block and which returns to the other block when procedure execution is complete. There are two kinds of procedures and procedure calls.

A procedure can be called by a procedure statement in the calling block. The procedure executes and returns to the statement following the procedure statement.

A procedure can be called by a function reference contained in a statement in the calling block, for example in an assignment statement. Such a procedure returns a value of a given data type to the point at which it was referenced.

PROCEDURE DECLARATIONS

The declaration of a procedure consists of defining:

1. The procedure identifier.

2. A procedure data type (if the procedure identifier represents a value, i.e., a function procedure.)

3. A list of formal parameters (if actual parameters are to be passed to the procedure when it is called.)

4. Specification of characteristics of the formal parameters.

5. The body of the procedure, which consists of a simple statement or a block that acts as a statement.

Items 1 to 4 constitute the heading of the procedure.

The usual rules of local and global identifiers apply to procedures when procedures contain other blocks. An example of a procedure declaration is:

```
real procedure arcsin(x) ;
          real x;
               arcsin :=arctan (x/sqrt (1-x↑2));
```

The procedure identifier is arcsin. Arcsin is a function that returns a real value. There is a single formal parameter x, which is specified *real*. The statement body consists of the single assignment statement.

Below is an example of a declaration of a procedure that is not called as a function.

```
procedure innerproduct (a,b,n,sigma);
          comment: compute innerproduct of vectors a and b with
          n components each. Store result as sigma;
          array a,b; integer n; real sigma;
          begin integer k;
          sigma :=0;
          for k :=1 until n do
          sigma :=sigma + a[k] × b[k];
          end innerproduct;
```

Note that in the example the procedure, innerproduct, contains a begin block. Whether a block is contained within another begin block or within a procedure, the rules for local and global identifiers are the same. In the example, integer k is local to the begin block and is undefined in the outer procedure. Arrays a and b, integer n, and real variable sigma are global to the begin block.

Many procedure declarations include formal parameters that are replaced by actual parameters when the procedure is called. However, procedures need not have parameters; for example, a procedure that generates a random number may not require that parameters be passed.

A procedure, like a variable, must be declared in the block in which it is used (that is, called). This means that the calling block must include the procedure declaration, including the full text of the procedure body, as part of the declarations at the beginning of the block, except under the conditions noted in the next section.

All ALGOL procedures are recursive and reentrant.

EXTERNAL PROCEDURES

The declaration of a procedure can be compiled as a separate entity. Such a procedure is called an *external* procedure since it is not declared in some other block.

To be called from some other block, the name of the procedure and its *external* characteristic must be declared in the calling block. For example:

EXTERNAL PROCEDURES

CALLING BLOCK	PROCEDURE
begin real s;	*real procedure* arcsin (x);
integer y;	*real* x;
external real procedure arcsin;	arcsin :=arctan (x/sqrt(1-x↑2));
x:=x × arcsin (x);	

Like *external* variables, *external* procedures can be called (used) by a number of blocks in which they are declared to be *external*.

PROCEDURE CALLS

Calls to procedures are of two forms: procedure statements and function references.

A procedure statement has the form:

```
procedure_name  (p1,p2,...,pn);
```

where: procedure_name is the identifier of the procedure.

p1, p2, ...,pn is a list of actual parameters that replace the formal parameters given in the procedure declaration. The list may be empty.

A procedure statement causes transfer of control to the named procedure and execution of the procedure body using the actual parameters of the calling statement. When the procedure body has been executed, control returns to the calling block at the statment following the procedure statement.

In the example following, the body of procedure A contains the bodies of procedures B and D, and the body of procedure B contains the body of procedure C; calls may be made as follows:

PROCEDURE CALLS (Continued)

```
procedure A                      <-A can call B and D but not C
    procedure B                  <-B can call A and C but not D
        procedure C              <-C can call A and B but not D
    procedure D                  <-D can call A and B but not C
```

An example of a procedure call is:

```
begin real a,b;  real array A[1:100];           )
     .                                          |
     .                                          |
     .                                          |
procedure sub_one (a,b,A);          )           |
     real a,b; real array A;        | procedure |
     statement;                     ) declaration > calling
     .                                          |   block
     .                                          |
     .                                          |
sub_one(a,b,A);      <-procedure call           |
X: ---;                                         )
```

When sub_one is executed, control returns to the statement labeled X.

A function reference has the form:

...procedure_name(p1,p2,...,pn)

where: procedure_name is the identifier of the procedure.

p1, p2,...,pn is a list of actual parameters that replace formal parameters given in the procedure declaration. The list may be empty.

where: ... the initial dots indicate that the function reference is part of a statement.

PROCEDURE CALLS (Continued)

A function reference causes transfer of control to the named procedure and execution of the procedure body using actual parameters. When the procedure body has been executed, a value for the procedure is returned to the calling statement. An example of a function reference is:

```
begin real y;
real procedure arctan(x); real x;     } procedure
statement;                            } declaration      calling
   .                                                       block
   .
   .
z := 0.215 × arctan(y);        ←procedure call
```

Calling a Procedure by Name and by Value

When an actual parameter is substituted for a formal parameter, the actual parameter may be some variable whose value when passed will be altered one or more times in the course of execution of the called procedure. If so, this is a call by name. The values of certain input variables to the procedures, however, will not be altered in the course of executing the called procedure. When such a parameter is passed, it constitutes a call by value.

Formal parameters that are consistently called by value are given the *value* specificator in the procedure declaration.

Example:

```
real procedure tan (x); value x: real x;
     tan := sin (x)/cos (x);
```

The actual parameter to be substituted for x in the example is an input value that is unaltered in computing the tangent function.

The rules of default precision apply to value parameters. In the example above, x would have default *real* precision. It is particularly important to declare precision for *string value* parameters since the default length is limited to 32 characters and any additional characters would be lost.

Calling a Procedure by Name and by Value (Continued)

```
real procedure sort (s); value s; string s [100];
```

Sometimes it is desirable to pass a parameter by value to a procedure that does not include a *value* specificator. In that case the actual parameter in the calling procedure is enclosed in double parentheses to indicate a by value assignment. Example:

```
begin integer input;
        .
        .
        .
        Routine ((input));
```

When a function identifier is passed as a parameter, the distinction between by name and by value call is as shown below;

```
begin external integer procedure input;
    .
    .
    .
Routinel   (input);      ←call to Routinel.  The address of
    .                     input is passed.
    .
    .
Routine2  ((input));     ←call to Routine2.  Function input
                          is called as the parameter of
                          Routine2.
```

FORMAL AND ACTUAL PARAMETERS

The formal parameters that appear in a procedure declaration are replaced by actual parameters when the procedure is called. Actual parameters will be evaluated at the time of the call only. Actual parameters may be values or variables, but they must match the formal parameters of the declaration as shown in the following rules:

FORMAL AND ACTUAL PARAMETERS (Continued)

1. Data types of actual parameters must be compatible with those of formal parameters. There is no conversion of parameters.

```
begin real alpha, beta; integer gamma;                        ⎫
   .                                                          ⎪
   .                                                          ⎪
   .                                                          ⎪
procedure xx(a,b,c);              ⎫                           ⎪
real a,b; integer c;              ⎪                           ⎪
begin ---                         ⎬ PROCEDURE                 ⎪
   .                              ⎪ DECLARATION               ⎬ CALLING
   .                              ⎪                           ⎪ BLOCK
   .                              ⎭                           ⎪
end;                                                          ⎪
   .                                                          ⎪
   .                                                          ⎪
   .                                                          ⎪
xx(alpha, beta, gamma);          ←call to xx                  ⎭
```

2. The number of actual parameters in a parameter list must match the number of formal parameters.

```
DECLARATION                      CALLING BLOCK

real procedure yy(i,j);
integer i,j;                  m :=m/yy(1,k); ←two actual para-
   .                                           meters replace two
   .                                           formal parameters.
   .
```

3. If a formal parameter is an array, it must be replaced by an actual parameter that is an array having the same or fewer array elements. The type and precision of the arrays must match exactly.

```
DECLARATION                   CALLING BLOCK

procedure gnp(fyl,fy5,SET);   begin integer i,j;array I[1:200];
integer fyl,fy5;                 :
array SET[15,15];             gnp  (i,j,I);
```

FORMAL AND ACTUAL PARAMETERS (Continued)

In the example, I[1] replaces SET[0,0], I[2] replaces SET[1,0], ..., I[199] replaces SET[6,12], and I[200] replaces SET[7,10].

4. A formal parameter that is called by value cannot be a switch identifier or a procedure identifier. An exception is a procedure identifier that has no formal parameters and that defines the value of a function designator. For example, if part of the declaration of procedure x is:

procedure x(dd); *value* dd; *integer* dd; *begin*

and if x is called by:

x ((FD)); ←where FD is a procedure, then FD must have the form:

integer procedure FD; ←no parameters

FD :=...; ←FD is assigned some value.

5. A formal parameter that occurs on the lefthand side of an assignment statement and is not called by value must be replaced by an actual parameter that is a variable. This rule is a logical extension of the rules of assignment statements.

6. Specification of formal parameters may place further restrictions upon the actual parameters associated with them. Such restrictions must also be observed in the body of the procedure.

7. The value of a function is parameter 0. The following are equivalent, where x is a function with one input parameter:

x(A,y); and A := x(y);

Specificators of Formal Parameters

Characteristics of formal parameters are specified in the procedure declaration as shown in preceding examples of procedure declaration. The parameter rules indicate that there must be a match between data types of formal and actual parameters and a match on the shape, i.e., a simple variable cannot replace a formal parameter that is used as an array.

The keyword declarators are also used as specificators. In addition there are the previously described *value* specificator and the *label* specificator, which allows the programmer to pass a label identifier as an actual parameter.

* * *

CHAPTER 9 -- LIBRARY FUNCTIONS AND PROCEDURES

Certain functions and procedures are supplied with the ALGOL compiler.

MATHEMATICAL FUNCTIONS

Arguments to the mathematical functions can be *real* or *integer*. Each function (except the sign function) yields a real value. If the argument to these functions is *integer*, the number of words returned is two. If the argument is *real* the number of words returned is the same as the number of words in the argument. The sign function, however, always yields an *integer* value.

FORMAT	MEANING AND EXAMPLES	
abs (x)	Absolute value of expression, x.	abs (g)↑(i/m)
arctan (x)	Principal value of $\tan^{-1}$ (x). Where expression x is in radians.	arctan (y-x)
cos (x)	Cosine of expression x, where x is in radians.	cos (n-pi/2)
exp (x)	Exponential function of the value of x which is the value of the Eulerian constant e raised to x: e^x.	exp (a[10])
ln (x)	Natural logarithm of expression x.	ln (a/2)
sign (x)	Sign of expression x, which is: +1 for x>0 0 for x=0 -1 for x<0	sign (a/b)
sin (x)	Sine of expression x, where x is in radians.	sin (omega x t)
sqrt (x)	Square root of expression x.	sqrt(abs (x-y))
tan (x)	Tangent of expression x, where x is in radians.	tan (a/b)

ENTIER FUNCTION

The entier function returns an integer, resulting from truncation of a real expression. Unlike the fix function, described below, the entier function will return multi-precision values where appropriate.

```
entier (x)
```

where: x is a real expression.

```
entier (y/cos(y))
```

FIX FUNCTION

The fix function returns a single precision integer, resulting from truncation of a real expression. The function has the form:

```
fix (x)
```

where: x is some real expression.

FLOAT FUNCTION

The float function returns a real value of default precision resulting from floating an integer expression. The function has the form:

```
float (x)
```

where: x is some integer expression.

SIZE FUNCTION

The size function returns as a value the number of characters in a scalar string or the number of elements in an array. The

SIZE FUNCTION (Continued)

function has the form:

```
size (v) ;
```

where: v is the identifier of a string or an array.

When v represents an array of strings, the number of elements in the array will be returned as a value.

```
size (alpha)           ←if alpha is a 12-element array, the
                        value 12 is returned.
```

ARRAY BOUND FUNCTIONS (LBOUND, HBOUND)

The hbound function returns an integer giving the upper bound of a specified dimension of an array; the lbound function returns an integer giving the lower bound of a specified array dimension. The functions have the form:

```
lbound (v, n)

hbound (v, n)
```

where: v is the identifier of the array

n is an integer representing the positional value of the dimension.

If array v has less than n dimensions or if n has a value less than or equal to 0, the function value is undefined.

```
real array A [1:9, 25, -2:4];

lbound (A,1)          ←returns 1
lbound (A,2)          ←returns 0
lbound (A,3)          ←returns -2

hbound (A,i)          ←if i=1, returns 9
                       if i=2, returns 25
                       if i=3, returns 4
```

ARRAY BOUND FUNCTIONS (LBOUND, HBOUND) (Continued)

If the second parameter in the examples is not 1, 2, or 3, the value of the lbound or hbound function is undefined.

BIT MANIPULATION FUNCTIONS (ROTATE, SHIFT)

The shift function permits contents of a location to be shifted left or right; the rotate function permits the contents to be rotated left or right. The functions have the form:

```
shift (v, n)

rotate (v, n)
```

where: v is an integer variable or octal literal.

n is an integer constant or variable.

The integer n indicates the number of bits to be displaced. A negative integer indicates left shift or rotate, and a positive or unsigned integer indicates right shift or rotate.

```
i :=rotate (x,-4);     ←value stored in i is contents of x left
                        rotated by four bits.

x :=shift (x,+4);      ←right shift by 4 bits the contents of x.
```

ADDRESS FUNCTION

The address function permits assignment of the location of a variable as the value of a pointer. The function has the form:

```
address (v)
```

where: v is a subscripted or unsubscripted program variable.

As described in Chapter 7, Pointers and the *based* Declarator, the address function is an extension to ALGOL that permits variable addressing on a level comparable to assembly language

ADDRESS FUNCTION (Continued)

programming. Refer to that section for further information on use of pointers and *based* variables with the address function.

Example:

```
begin pointer p; integer array b;
      integer i; based integer x;
            .
            .
            .
      p :=address (b[i]) ;          ←pointer p is assigned the
            .                        address of array element
            .                        b [i].
            .
```

STRING FUNCTIONS (LENGTH, INDEX, SUBSTR, ASCII)

Length Function

The length function returns as a value the length of its character string argument. The function has the form:

length (v)

where: v is a string variable.

Examples:

```
string (10) x; integer i;
x := "abcd";
i:= length (x);       ←The assignment is the same as i:=4;
```

Index Function

The index function searches a specified character string for a given character configuration. The function returns the starting location of the configuration as its value. The function

Index Function (Continued)

has the form:

```
index (v, c)
```

where: v is a string variable.

c is one or more characters of v. If c is not found, index returns a zero value.

Examples:

```
string (10) v; integer i;
v := "abcdefg";
i := index (v,"bc");        ←The assignment is the same as i:=2;
      .
      .
      .
v := "abcdefg";
i := index (v, "b");        ←The assignment is the same as i:=2;
```

Substr Function

The substr function extracts from a given string a substring whose length is defined by the user. Substr will treat an integer or based integer datum as if it were a string, extracting a subset of the datum. Use of the substr function gives Extended ALGOL much of its flexibility in manipulating strings.

The function has the form:

substr (v, n_1 [, n_2])

where: v is a string, integer, or based integer variable.

n_1 is an integer giving the position in v of the first character or digit to be extracted.

n_2 is an integer giving the position in v of the last character to be extracted.

Substr Function (Continued)

If $\underline{n}_2$ is not given, the character indexed by $\underline{n}_1$ is returned. If $\underline{n}_2$ is greater than the maximum number of characters, all characters from $\underline{n}_1$ to the end of string or datum are returned. If $\underline{n}_1$ evaluates to less than 1, the index begins at the first character of $\underline{v}$.

```
literal x("ABCDEFGH");
substr(x, 1,8)          references the entire string ABCDEFGH.
substr(x, 5,7)          references EFG.
substr(x, 4)            references D.
substr(x, 0,3)          references ABC.
substr(x, 6,9)          references FGH.
```

In the following three examples, assume the declarations and assignments to be:

```
string a, b, c;
a := "abcdefg";  b :="xxx";
```

To join contents of b with contents of a, producing "abcdefgxxx":

```
substr(a, length(a)+1, length(a)+length(b)):=b;
```

To replace part of string a by string b, producing "abcdxxx":

```
substr(a, length(a)-3, length(a)):=b;
```

To insert the contents of b into string a, producing "abcxxxdefg", requires a temporary string and the setcurrent procedure, explained later in this chapter.

Substr Function (Continued)

```
c:=substr(a, 4,7);                                    ←a="defg"
setcurrent (a,3);                                     ←a="abc"
substr(a,length(a)+1,length(a)+length(b)):=b;  ←a="abcxxx"
substr(a,length(a)+1,length(a)+length(c)):=c;  ←a="abcxxxdefg"
```

Digits of an integer or based integer are treated as string characters:

```
integer i,j;
i :=1776;
j :=substr(i,2,3);    ←j contains 77.
```

Accessing of multiple substrings of the same string can be accelerated by setting a pointer to the address of the string and using a based string or a based integer in the substr function. The pointed-to based integer is not treated as an integer value as in the example above, but is treated as a string. Faster execution is obtained by making use of substr handling of based integers rather than based strings. This feature is useful, for instance, in accessing various string fields in a large record in core. The user must be careful when using pointers since no core protection is provided.

```
based integer bi; based string bs;
pointer p; string s;
s:= substr (p→bi, 1, 5);  ←The assignment statements are the
s:= substr (p→bs, 1, 5);    same except that the based integer
                            is faster. The pointer must have
                            been set up previous to these two
                            statements.
```

Ascii (byte) Function

The ascii or byte function, like substr, can be used to manipulate either a string character or byte of an integer or based integer. Byte and ascii are equivalent function names. The function returns the numeric value of a character or a byte of a datum. The function has the form:

Ascii (byte) Function (Continued)

```
ascii (v [,n] or byte (v [,n])
```

where: v is a string variable or literal or integer variable.

n is an integer giving the position of the byte of v to be returned.

If n is not given, the first byte is returned. If n is greater than the number of bytes in v, the last byte is returned.

```
literal s("ABCD"); string s2; based integer bi;
pointer p; integer a;

a :=ascii(s,4);  ←returns 104₈ in a.
s2 :=ascii(p→bi, 2); ←returns the second byte of the area
                       pointed to by p in s2.
```

MEMORY FUNCTION

The memory function returns an integer value giving the remaining number of words of core available to the user and is used to keep track of core for allocation, stack space, arrays, etc. The function has the form:

```
memory
```

CLASSIFY FUNCTION

The classify function permits the user to obtain an integer which represents the predefined class of ASCII characters to which the first parameter passed by classify belongs. The function reference has the format:

```
classify (integer, class-table-ptr)
```

CLASSIFY FUNCTION (Continued)

where: integer is an integer or an expression evaluating to an integer in the range of octal equivalents of ASCII characters.

class-table-ptr is a pointer to a user-written table classifying ranges of ASCII characters. The class, table and pointers are usually external to the block in which referenced.

The user defines a class table for ASCII characters as a series of ranges of the form:

$$\left.\begin{array}{l}\text{character-min}_1\\ \text{character-max}_1\\ \text{result}_1\end{array}\right\}\ \text{range}_1$$

$$\vdots$$

$$\left.\begin{array}{l}\text{character-min}_n\\ \text{character-max}_n\\ \text{result}_n\end{array}\right\}\ \text{range}_n$$

Any number of ranges may be defined. For example, all uppercase alphabetics could constitute one range, digits 0 through 9 could constitute another range, the single character, left-parenthesis, could constitute a third range, etc. The final range in the table, however, must include the entire ASCII character set, providing the default range with default return classification.

```
i:= classify (ascii(x,1),ptable)
```

I/O PROCEDURES

Since standard ALGOL was designed to be a language independent of specific processors or devices, no I/O statements or conventions are included in the ALGOL specification.

For user convenience, a number of I/O procedures are implemented in Extended ALGOL to handle I/O. These procedures are run-time routines that can be called by a user program using a procedure statement. If the user wishes, he can implement additional I/O features by writing his own external procedures to handle input and output.

Open a File

Call Format:

```
open (channel, string [error-label]);
```

where: channel is one of 8 channels (0-7) that can be associated with a given file. Under RDOS up to 63 channels can be made available using the RLDR local C switch.

string is the character string giving the file name. It can be either a literal such as "$LPT" or "DATAFILE" or a string containing the file name.

error-label is an optional identifier label of a statement in the calling program to which transfer is made if an error occurs in opening a channel. If an error-label is given and the file does not exist, transfer will be made to the error-label without creating the file. If the file does not exist, and no error-label is supplied, the file will be created.

Purpose: The procedure opens a file for reading or writing and associates a channel with the file.

Examples:

```
open (2, infilel, openerr);
open (3, "$TTI");
open (4, "$TTO", no_open);
```

Close a File

Call Format:

```
close (channel);
```

where: channel is the channel number currently associated with the file through an open procedure.

Purpose: The procedure is called to close a file after I/O is completed.

Example:

```
close (1);
```

Read a File

Call Format:

```
read (channel, list [,eof-label, error-label]);
```

where: channel is the channel number associated with the file to be read.

list is a list of input data.

eof-label is an optional label of a statement in the calling procedure to which transfer is made if an end-of file is encountered on reading. For console input, an end-of-file is defined as a CTRL Z. For all other devices and files, an end-of-file is written automatically by the system.

error-label is an optional label of a statement in the calling procedure to which transfer is made if a read error occurs.

Purpose: The procedure is called to input data from a file.

Read a File (Continued)

Input data: Data will be read in free format. All legal numeric or string literals are acceptable as input. If a string begins with a quotation mark, the string will terminate at the next quotation mark. If a string does not begin with a quotation mark, the remainder of the input line will be considered as part of the string, excluding the carriage return.

Generally, only one record (that is, only data up to the first carriage return or form feed) is input by read. If list specifies more data than is on a single record, the next record is read automatically until the number of arguments in list is input. Additional data, if any, in the record are lost.

Examples:

```
read (1, B[I], OMEGA, EOFTAG, ERROR25);

for I:=0 step 1 until 10 do
read(2, A[I], B[I]);
```

Write a File

Call Format:

```
write (channel, list [,error-label]);
```

where: channel is the channel number associated with the file to be written.

list is a list of output data.

error label is an optional label of a statement in the calling procedure to which transfer is made if any error, including an end-of-file, occurs.

Write a File (Continued)

Purpose: The procedure is called to output data to a file.

Output Data: Data may be variables, or numeric or string literals. The write procedure provides no formatting of output; for complete control of formatting, the output procedure should be used. For limited format control, control characters interpreted by the assembler can be included in the list.

A null character is appended to each output datum in list. The read procedure ignores nulls following list. The read procedure ignores nulls following input data. However, if the output from write is to be used by a Data General Assembler, all nulls must be deleted from the output. The user can first input the output file to the Text Editor, which deletes all nulls, then use the Editor output as input to the assembler.

Output can be input by the read procedure without change.

Examples:

```
write (2, "END SORT<15>", A, "<15>");
```

END SORT is a string literal. Inclusion of the characters "<15>" following END SORT causes a carriage return. The value of the variable A will then be printed, followed by another carriage return.

```
write (3, y, x, z, sub[i], errortag);
```

The list of variables to be written is y, x, z, and sub[i]. Values for the variables will be written with a single space between value fields. If a write error occurs, a transfer is made to the error label, errortag.

Write Formatted Output

Call Format:

```
output (channel, "format", list [,error-label]);
```

where:

- channel is the channel number associated with the file to be written.
- format is a string specifying output format.
- list is a list of variables to be written out according to the given format.
- error-label is an optional label of a statement in the calling procedure to which transfer is made if any error, including an end-of-file, occurs on output.

Purpose: The procedure permits the programmer to set up his own format for data being output, rather than using the default format of the write statement.

The format specification may include literals to be output, formats for numeric and string values of variables given in list, and carriage control, tabulation, and form feed information.

A null character is appended to each output datum in list. The read procedure ignores nulls following input data. However, if the output from this procedure is to be used by a Data General assembler, all nulls must be deleted from the output. The user can first input the output file to the Text Editor, which deletes all nulls, then use the Editor output as input to the assembler.

Formatting information for list variables must precede the variables to be output. Literal strings containing carriage control, form feed, and tabulation information and character string literals may appear where needed within list. An example of a literal to be output, precisely as given in format would be:

```
output (1, "Data Reduction");

Data Reduction                    ←resultant output
```

Write Formatted Output (Continued)

A "picture" specification of data to be output is set up in the format field, using the character # to represent each character position of the datum. Numeric values that have fewer characters than the positions given by the format field will be right-justified in the field. Numeric values having more characters than the positions given by the format field will be output in full; i.e. a single # can be used to output numbers of any length.

```
output (1, "DATA REDUCTION: ####", A);

DATA REDUCTION: 901495             ←resultant output if A
                                    has the value 901495
```

When formatting floating point numbers, a decimal point can be part of the field format, indicating the number of digits that should follow the decimal point in the output format. The programmer should round the data to the number of digits desired.

```
output (2, "#####.# ", w+.05, x+.05, y+.05, z+.05);

ΔΔΔΔ 1.2ΔΔ-99.0ΔΔΔΔΔ .1ΔΔ 999.9   ←possible resultant output;
                                   Δ represents a blank
                                   position
```

To round each datum to the nearest tenth, .05 is added to each datum.

A field format may have a positive sign, negative sign, or can be unsigned. Resultant output will differ in the following manner:

Unsigned Field:	If the datum is positive, the output value is not signed. If the datum is negative, the output datum is signed and requires a field position, e.g., the range of field ###.### would be from -99.999 to 999.999.

Write Formatted Output (Continued)

Positive (+) Field:	The sign will be output for both positive and negative numbers and requires no field position, e.g., the range of field +###.### would be from -999.999 to +999.999.
Negative (-) Field:	The sign will be output only for negative numbers. It requires no field position, e.g., the range of field -###.### would be from -999.999 to 999.999.

An exponent field is allowed as part of a decimal field that has an explicit decimal point. An exponent field is signalled by the letter E followed by # signs representing exponent digit positions. The exponent will be right justified in the exponent field. Output of signs for the exponent follows the sign conventions given above.

```
output (2, "-#####.##E##", a+.005, b+.005, c+.005, d+.005);

12345.25E-4△△△.99.04E△0△△ 9876.97E-6△△-555.55E△0  ←possible
                                                      output
```

The # symbol can also be used to represent string variables in the list of the output procedure call. The string will be left justified in the output field with trailing blanks. However, if the string or substring is longer than the field format the entire string will always be written out.

```
output (2, "########", ST1, ST2, ST3);

TITLE△△△NUMBER△△CHARACTERISTICS        ←possible output
```

Numeric values for output can be converted to strings. They will then be left justified in the output field.

String literals for output may appear anywhere within the output list.

Write Formatted Output (Continued)

```
output (2, "#######","SERIAL NUMBER",A[2], "FIVE ON ORDER");

SERIAL NUMBER  201555 FIVE ON ORDER
```

ASCII carriage control characters, written in octal code and enclosed in angle brackets, can be incorporated into the format. In the examples below, 011 is the octal code for the tab character and 015 is the carriage return character.

```
output (2, "####.##<11>", a,b,c,d "<15>");

4678.23      -234.40      1678.49     -233.43
```

```
output (2, "####<11>####<15>", a,b,c,d);

4678          -234
1678          -233
```

An array identifier in a variable list causes all elements of the array to be written out in normal array sequence.

```
output (2, "#####", A);                  ←A is a ten-element array

 34 5781  777 1234  354   9  100 4555 9000  888      ←possible
                                                      output
```

By setting up loops containing output procedure calls, it is possible to produce output data in a number of formats.

Write Formatted Output (Continued)

```
for j := 1 step 5 until 100 do begin
for i := j step 1 until j+4 do
output (2, "####   ", a[i]);
write  (2, "<15>");
end;

   0  1020  4545  6123  9081
7060  -354   765    20    -1    ←part of possible output
 555  9000    34   -10   563
```

```
begin literal s(" A[###] = ####  ####  ####  ####  ####<15>");
based integer array ba[0:4]; pointer p;

for j := 1 step 5 until 100 do
begin p := address(a[j]);
      output (2,s,j,p→ba);
end;

A[  1] = 1005  1195  3142  5222  1110
A[  6] =   19  3001  -100    25  5111
A[ 11] =  211    -4  4321     2   444   ←part of possible output
      .
      .
      .
A[ 96] =   35  -567  2378   888   200
```

Note in the last example that the format field for the array has been set up as a literal.

Read or Write a Line

Call Formats:

```
line read (channel, pointer, count [,error-label]);

linewrite (channel, pointer, count [,error-label]);
```

where: channel is the channel number associated with the file to be read or written.

pointer is a pointer to the word in core at which reading or writing begins.

count is a return value giving the number of bytes read or written.

error-label is an optional label of a statement to which return is made if any error, including an end-of-file, occurs.

Purpose: The procedures provide for reading and writing a line of data into an area, rather than into variables (read and write procedures). Otherwise, the procedures are identical to read and write.

The pointer contains the address of a core word at which reading or writing begins. The data is transferred from that point up through the first carriage return, null or form feed character.

If an EOF occurs on a lineread, count will contain the number of bytes read up to the EOF.

In using lineread with strings, note that it is necessary to provide the count of bytes read using the setcurrent procedure as shown in the example:

Example:

```
lineread (0, address(s), n, er);  ←s is a previously declared
setcurrent (s, n);                 string.  setcurrent (see
                                   setcurrent procedure) sets
                                   the length of s to the count
                                   of bytes read in lineread.
```

Read or Write a Number of Bytes

Call Formats:

```
byteread (channel,pointer,count [,error-label]);

bytewrite (channel,pointer,count [,error-label]);
```

where: channel is the channel number associated with the file to be read or written.

pointer is a pointer to the word in core where reading or writing begins. pointer can be an address expression.

count specifies the number of bytes that the user wishes to be read or written.

error-label is an optional label of a statement to which return is made if any error, including an end-of-file, occurs.

Purpose: The procedures provide facilities for binary reading and writing of data, rather than ASCII.

Example:

```
allocate (buffer,100);
byteread (0, buffer,200);
```

←allocate allocates 100 words in the file starting at pointer, buffer. byteread reads 200 bytes starting from buffer.

POSITIONING A FILE

Position Procedure

Call Format:

```
position (channel, byte [,error-label]);
```

where: channel is the channel number associated with the file to be positioned.

byte is the number of the byte to which the file is to be positioned.

Position Procedure (Continued)

where:

error-label is an optional label to which a return is made if the file given cannot be positioned to the indicated byte.

Purpose: The procedure permits random access to records and is called before attempting to read or write random. The byte specified may be an integer, real, or multi-precision integer whose value is between 0 and 4,294,967,296 bytes (the limit of file bytes).

Examples:

```
position (2, 5000);        ←position to byte 5000 in file
                            on channel 2 and read A[i].
read (2, A[i]);
```

```
position (0, bytpos);           ←position to a byte (bytpos)
                                 which is beginning byte of
bytewrite (0, buffer, 200);      previously allocated area
                                 pointed to by pointer, buffer,
                                 and write 200 bytes. (See
                                 allocate procedure.)
```

Filesize Procedure

Call Format:

```
filesize (channel, n);
```

where: channel is the channel number associated with a file.

n is the identifier of the value to be returned, representing the length in bytes of the file. n may be integer, real, or multi-precision integer.

Filesize Procedure (Continued)

Purpose: The procedure returns the current length in bytes of a disk file, providing information useful in positioning the file.

Example:

```
filesize ( 0, n);               ←A call to filesize makes it
position ( 0, n);                possible to position to the
bytewrite ( 0, ptr, 200);        end of the file for writing.
```

Fileposition Procedure

Call Format:

```
fileposition (channel, n);
```

where: channel is the channel number associated with a file.

n is the identifier of the value to be returned, representing the position of the byte currently pointed to in a disk file. n may be integer, real, or multi-precision integer.

Purpose: The procedure returns the position of the byte currently pointed to in the given file, providing information useful in positioning a file.

Example:

```
fileposition (1,n);             ←After finding the current
position (1, n+300);             position, the user positions the
                                 file to a byte 300 bytes
                                 beyond.
```

STORAGE ALLOCATION PROCEDURES

The programmer can designate that a certain number of words of storage be allocated with a pointer to the first word. At a later time, the storage can be deallocated for reuse.

Allocate Procedure

Call Format:

```
allocate (pointer, number);
```

where: pointer is the identifier of a previously declared pointer.

number is the number of words of storage to be addressed by the pointer.

Purpose: To allocate a number of words of storage for manipulation by the pointer-based variable method.

The algorithm used for allocate is a first fit method only if the size in words on the free list equals the size requested. Otherwise, a new area is allocated.

```
begin integer m,i,j; pointer il,iu;
      based integer n;
allocate (il, 8);
allocate (iu, 8);
    .
    .
    .
i:=(il+m)→n;
j:=(iu+m)→n;
```

Free Procedure

Call Format:

```
free (pointer);
```

where: pointer is the identifier of a pointer that appears in a previous allocate call.

Free Procedure (Continued)

Purpose: To make available for reallocation the previously allocated words of storage.

```
free (il);
free (iu);
```

Setcurrent Procedure

Call Format:

```
setcurrent (string, bytes);
```

where: string is the identifier of a previously declared string.

bytes is the number of bytes (characters) to be set as the current maximum length of string.

Purpose: To insure that the current length of a given string is the length desired for manipulation.

If bytes is larger than the declared maximum length of the string, string will be set to the declared maximum.

The procedure is of particular value in insuring that the length of the buffer for reading and writing random is correct.

```
string (128) S;                        ←declare 128-byte string S.
        .
        .
        .
position (1, 155);
byteread (1, address(S), 128);         ←transfer 128 bytes to core.
setcurrent (S, 128);                   ←insure current length of S
                                        is maximum length.
```

Comarg Procedure

Call Format:

```
comarg (channel, string [,boolean-array] [,eof]);
```

where: channel is the channel number of an RDOS command file.

string is the identifier that will contain an argument of the command file.

boolean-array is a 26-element boolean array that may optionally contain switch settings of the command argument.

eof is an optional label of a statement to which return is made when the end of the command file is encountered.

Purpose: The procedure is used to read RDOS commands from a command line into a command file. (The creation of a command file, COM.CM, is described in Appendix C of the RDOS User's Manual.) Briefly, COM.CM contains a given command line in the following format:

Comarg Procedure (Continued)

Purpose:

Command File Format — Command: FOO/B/L/N A MB/A/X/Z

byte	file content	
0	F	
1	O	← First argument FOO.
2	O	
3	null	
4	01000000	
5	00010100	Global settings of switches;
6	00000000	set for B, L, and N.
7	00	
8	A	← Second argument A.
9	null	
10		
11		No local switches for A.
12		
13		
14	M	
15	B	← Third argument MB.
16	null	
17	10000000	
18	00000000	Local switches set for MB
19	00000001	are A, X, and Z.
20	01	
21	377	← End of file indicator.

To obtain the contents of COM.CM as given above:

```
string  COM1,COM2,COM3;
boolean array  B1,B2,B3[25];
      .
      .
      .
open (1, "COM.CM");

comarg (1, COM1, B1);
comarg (1, COM2, B2);
comarg (1, COM3, B3);
```

Comarg Procedure (Continued)

The open procedure associates COM.CM with channel 1. When the three comarg procedures are executed, COM1 will contain FOO, COM2 will contain A, and COM3 will contain MB.

Those boolean array elements of B1, B2, and B3 that correspond to the bit positions set in the command file will be set to true. Thus, the elements of B2 are all set to false (0) while elements B3[0], B3[23], and B3[25] are set to true.

FILE MANIPULATION PROCEDURES

The file manipulation procedures are useful when files are maintained on disk. They permit deletion and renaming of given files. The named file must exist and must be closed at the time of a deletion or renaming.

Delete a File

Call Format:

```
delete ("file");
```

where: file is the name of a previously created file.

Purpose: The routine deletes the named file from the disk.

Example:

```
delete ("oldfile.SR");
```

Rename a File

Call Format:

```
rename ("file1", "file2");
```

where: file1 is the name given a previously created file.

file2 is the name to be substituted for file1.

Rename a File

Purpose: The routine allows a file to be renamed.

Example:

```
rename ("main", "sub2");
```

ERROR PROCEDURE

Call Format:

```
error ("error-message");
```

where: error-message is a string

Purpose: The procedure allows the user to write his own error messages. The error message will be output at the console when an error occurs, processing will terminate, and return will be made to the operating system.

PROGRAM SWAPS - CHAIN PROCEDURE

Call Format:

```
chain ("filename");
```

where: filename is the name of a save file. The loader adds the extension .SV to filename, so the programmer should include the extension when giving a literal filename in the chain procedure.

Purpose: The procedure allows an executable program file on disk to be brought into core for execution, replacing the currently executing program. The call to chain should be the last statement in the program. Any number of saved files can be chained, providing each ends with a call to chain.

PROGRAM SWAPS - CHAIN PROCEDURE (Continued)

Example:

```
chain ("plot.sv");
```

REAL TIME CLOCK PROCEDURES

The ALGOL real time clock procedures, stime and gtime, allow the user to change or retrieve the current date and time in an ALGOL program.

Stime Procedure

Call Format:

```
stime (year, month, day, hour, minute, second);
```

where: year is an integer constant or variable representing the current year less 1968; e.g., 1974 is represented as 6.

month is an integer constant or variable representing the current month, in the range of 1 through 12.

day is an integer constant or variable representing the current day, in the range of 1 through 31.

hour is an integer constant or variable representing the current hour, in the range of 0 through 23: 0 is the midnight hour; 23 is 11 PM.

minute is an integer constant or variable representing the current minute, in the range of 0 through 59.

second is an integer constant or variable representing the current second, in the range of 0 through 59.

Stime Procedure

Purpose: The procedure sets the real time system clock and calendar to the specified date and time.

Example:

```
stime (6, 1, 1, 0, 0, 1);
```

Gtime Procedure

Call Format:

```
gtime (year, month, day, hour, minute, second);
```

where: year, month, day, hour, minute, and second are integer variables for which real time clock values are returned. The range of these variables is the same as for the stime procedure.

Purpose: The procedure returns the current date and time in the user-specified variables.

Example:

```
integer year, month, day, hr, min, sec;
     .
     .
     .
gtime (year, month, day, hr, min, sec);
```

MULTIPLY AND DIVIDE PROCEDURES

The multiply and divide procedures perform unsigned multiplication, expressing the result as a product and overflow, and division, expressing the result as a quotient and remainder.

Umul Procedure

Call Format:

```
umul (multiplicand, multiplier, adder, overflow, product);
```

where: multiplicand is an integer constant or variable containing a value to be multiplied.

multiplier is an integer constant or variable containing a value to be multiplied.

adder is an integer constant or variable containing a value to be added to the result obtained from multiplying the first and second arguments.

overflow is an integer variable to which the overflow of product, if any, is returned.

product is an integer variable to which the result of multiplication is returned.

Purpose: The umul procedure provides unsigned multiplication of the form:

(multiplicand × multiplier) + adder → (product + overflow)

Example:

```
integer plicand, plier, adder, ovflo, prodt;
        .
        .
        .
umul (plicand, plier, adder, ovflo, prodt);
```

Rem Procedure

Call Format:

```
rem (dividend, divisor, quotient, remainder);
```

where: dividend is an integer constant or variable containing the value of the dividend.

divisor is an integer constant or variable containing the value of the divisor.

quotient is an integer variable to which the quotient obtained by the division is returned.

remainder is an integer variable to which the overflow of division is returned.

Purpose: The rem procedure obtains the result of division as a quotient and remainder.

Example:

```
integer a, b, quotn, mander;
     .
     .
     .
rem (a, b, quotn, mander);
```

CACHE MEMORY MANAGEMENT

A capability of Extended ALGOL, which is designed to meet specialized file access needs of certain programmers, is called Cache Memory Management (CMM). Extended ALGOL without CMM will handle efficiently most scientific and business programming applications. Cache Memory Management is a powerful tool for programmers who deal with very large programs and large data bases -- primarily systems programmers such as compiler writers. CMM provides more efficient means of file access when the size of a file is considerably larger than available memory (for example, three or more times larger).

CMM provides means of buffering large files into 256 word blocks and determining which blocks reside in core on a usage basis. For example, suppose a new block is required in core from a disk file. The block will be swapped in, replacing the block currently in core which has the oldest reference time. Thus, CMM will replace the least recently used block with the block from disk.

To use CMM the programmer sets up a buffer pool consisting of a fixed number of blocks and a header area (buffer procedure). He can then open a given file (or create and open the file) to a given file number and set up access to the file through the buffer (access procedure). The remaining general procedures and functions used in Cache Memory Management are listed below:

wordread/wordwrite	-	used in reading from or writing to any area of any file. These are the most general of the routines for reading and writing.
hashread	-	used in reading any area of a file into a core buffer and returning the precise location of any word of data as the address of a block of core and an offset into the block of the specific word. No actual data transfer happens. This routine has been typically used in files hash-coded by the user for the purpose of finding data without its being modified.

hashwrite	-	used to mark hashread data as having been modified.
flush	-	used to write to disk the contents of all modified data buffers before a file is closed.
close	-	used to close a file that has been previously accessed. The file should be flushed before being closed.

Besides the general procedures, there are a group of specialized procedures available in CMM. These procedures provide extra speed and simplicity through two features:

1. They are only used to access file number 0.

2. They allow the user to make block 0, the first 256 words of the file, an area of restricted access to be used for vital information and pointers into the general data area.

The specialized procedures are:

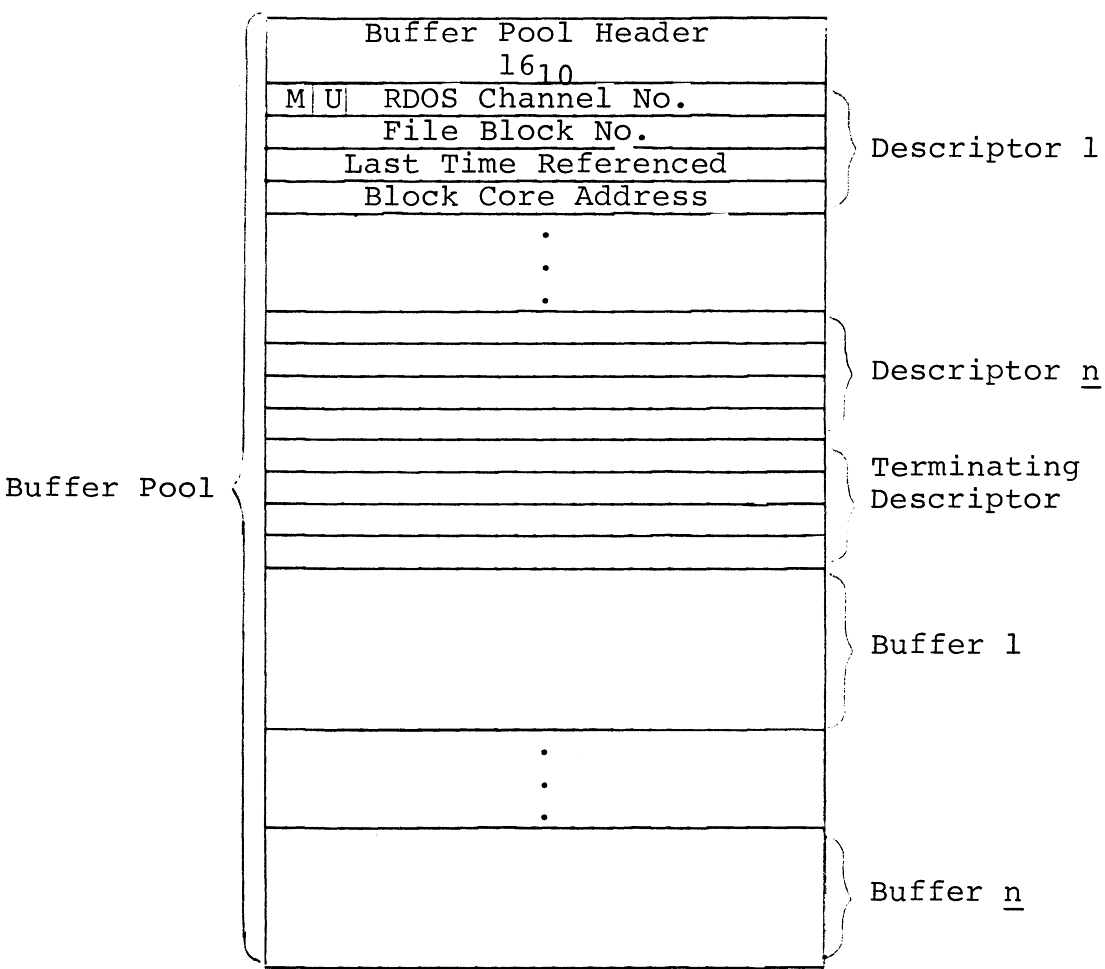
noderead/nodewrite	-	used in reading from or writing to file number 0, excluding the first 256 words. File access is on a nodal basis, where a node (which resembles an ALGOL array) is described later.
fetch/stash	-	used in reading from or writing to file number 0. The first 256 words of the file may optionally be read or written. A single word is accessed by these routines, where access is on a nodal basis.
nodesize	-	used to obtain the number of words in a given node.

CACHE MEMORY MANAGEMENT (Continued)

Setting up a Buffer Pool (buffer)

Before a file can be accessed using the cache memory facility, the programmer must establish a buffer pool to be associated with the file. To establish the buffer pool, the programmer must determine the size required, which is the number of words of the file that can be maintained in core at a given time. The buffer pool is allocated at the high address end of user stack space. The buffer pool consists of a buffer pool header of 16_{10} words, 4 words of descriptors for each buffer, four words of terminating descriptor, and the required number of 256_{10} word buffers.

A buffer pool of n buffers is configured as shown:



Setting up a Buffer Pool (buffer) (Continued)

The buffer pool header is 16 words of information needed by CMM to control the buffer pool. It includes a save area, data pointers, counters, and a clock that maintains the current time.

Following the header is a set of buffer descriptors. Each buffer in the pool has a corresponding four-word buffer descriptor of the buffer and its usage. The buffer descriptor, as shown in the previous figure, contains the following:

Word	Contents
1	Bit 0 = M (modify bit) indicating if the contents of the buffer have been modified since last read in. Bit 1 = U (usage bit) indicating if the buffer is currently occupied. Bits 2-15 RDOS channel number, corresponding to a user-assigned file number in the access call.
2	File block number.
3	Core address of the first word of the buffer.
4	Time of the last reference to the buffer.

The descriptors are allocated by CMM in the same order as the buffers. The first descriptor corresponds to the first buffer, etc. Following all descriptors are four words terminating the descriptors.

Immediately after the four words terminating the descriptors are the actual CMM buffers. The buffers are 256_{10} words (one block) in length.

The user sets up the buffer pool and thus makes it possible to use CMM through the buffer procedure. The call to buffer is

buffer (pointer, poolsize) ;

where: pointer is a previously declared integer variable or pointer. CMM selects the buffer pool area and returns the address of the start of the buffer pool in pointer.

Setting up a Buffer Pool (buffer) (Continued)

where: poolsize is the size of the buffer pool, in words. The user can specify an integer expression, indicating the number of words in the buffer or use the built-in function memory (or memory/n, where n is an integer), indicating that the rest of available memory (or the indicated fraction of available memory) is to be used for the buffer pool.

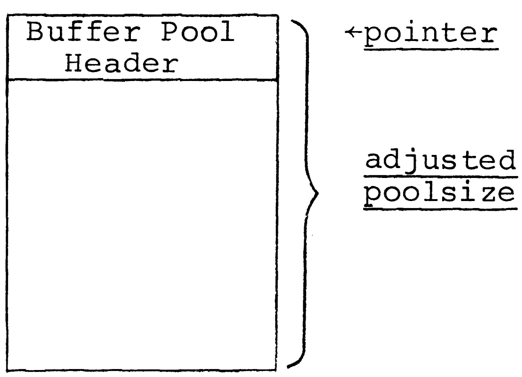
The actual size used by CMM for the buffer pool is always the largest available memory space less than or equal to poolsize. The formula used by CMM to compute this value is:

$$(256 + 4)\ \underline{i} + 20 \leq \underline{\text{poolsize}}$$

where 256 is the number of words in a block, 4 is the number of words in the buffer descriptor, and 20 is the number of words needed for the buffer pool header plus terminating descriptor. Thus, the number of buffers allocated (i) is the largest integer satisfying the inequality:

$$\text{number of buffers } (\underline{i}) \leq \frac{\underline{\text{poolsize}} - 20}{260}$$

Thus, once allocated, the buffer pool is configured as:



The user can then open the file via the access routine for disk access.

The following example reserves half of remaining memory for the buffer pool and returns the address of the start of the pool in PTR1.

Setting up a Buffer Pool (buffer) (Continued)

```
pointer  PTR1;
     .
     .
     .
buffer  (PTR1, MEMORY/2);
```

Opening Buffered Files (access) (Continued)

Once the buffer pool has been set up with a pointer to the starting address, the programmer may open files for accessing via the buffer pool. Files are opened, or are created and then opened, using the access routine call, which has the format:

```
access (filenumber, filename, pointer [,elementsize]);
```

where: filenumber is the file number that is associated with a file.

filename is the character string giving the file name. It can be either a literal in quotation marks or a string variable containing the file name.

pointer is the buffer pool pointer used in the buffer routine.

elementsize is an integer representing the size range of the file to be opened as follows:

Size of the File	elementsize
0 - 65K words	1
65 - 131K words	2
131 - 196K words	3
196 - 262K words	4
.	.
.	.
.	.

elementsize has a default value of 1; thus if the size of the file $\leq$ 65K, the parameter need not appear in the routine call.

Opening Buffered Files (access) (Continued)

CMM stores all file positions as a 16-bit unsigned integer. This means that only 65K words (the largest value that can be stored in 16 bits) can be addressed directly. If a file is larger than 65K words, the user specifies an element size (elementsize) of two or more to indicate that two or more consecutive words are to be considered a single element and all file positions are addressable by CMM.

The actual file address of a word becomes:

$$\text{file address} = \text{fileposition}/\text{elementsize}$$

When reading or writing the file, the user must specify the file address of the first word in the element to be assured of accessing the correct data. Given a file address, CMM then calculates the block number of the element using the formula:

$$\text{file block number} = \frac{\text{file address} \times \text{elementsize}}{256}$$

If the file does not exist when access is called, CMM creates a randomly organized file of length Ø with the specified file name. The file is then opened via the RDOS command .OPEN, which associates the file with a channel number and makes the file available for both reading and writing.

The following example opens the file LEXICAL and assigns it to file number 1 with PTR1 pointing to the beginning of the buffer pool. The size of the buffer pool passed is 5000 words, although by computation the buffer will only use 4880 words. Because the file is 127K words long, an element size of two is specified in the call to access.

```
pointer PTR1;
    .
    .
    .
buffer (PTR1, 5ØØØ) ;
access (1, "LEXICAL", PTR1, 2) ;
```

CACHE MEMORY MANAGEMENT (Continued)

Wordread/wordwrite Routines

The routines WORDREAD and WORDWRITE allow the programmer to read from or write into any area of a file. The format of the call to WORDREAD is:

wordread (filenumber, fileaddress, coreptr [, words]) ;

where:

filenumber is a user-assigned integer file number previously associated with the file in a call to access.

fileaddress is an integer constant or variable specifying the file address of the first word of the file to be read.

coreptr is a previously declared integer variable or pointer specifying the first word of memory to contain the data read.

words is an integer constant or variable specifying the number of words to be read. If words is omitted, the routine looks for the count of words as the first word indicated by fileaddress in the file and reads using that count for words.

Before performing the data transfer, CMM determines if the block containing fileaddress is in memory; if it is not, the block is read in.

The following example opens the file LEXICAL and assigns it to file number 1 with BUFPTR pointing to the beginning of the buffer pool. The wordread procedure then accesses file address 200, which is file position 400 because elementsize is 2, and reads two words into the memory area pointed to by COREPTR.

```
pointer BUFPTR, COREPTR ;
   .
   .
   .
buffer (BUFPTR, 5ØØØ) ;
access (1, "LEXICAL", BUFPTR, 2) ;
wordread (1, 2ØØ, COREPTR, 2);
```

To write a block of data onto disk, the user can use wordwrite. The call format is:

```
wordwrite (filenumber, fileaddress, coreptr [, words ]) ;
```

where:

filenumber is a user-assigned integer file number previously associated with the file in a call to access.

fileaddress is an integer constant or variable specifying the file address of the first word of the file to contain the data written.

coreptr is a previously declared integer pointer or variable specifying the first word of memory containing the data to be written.

words is an integer constant or variable specifying the number of words to be written. If words is omitted, the routine looks for the count of words as the first word indicated by fileaddress in the file and uses that count for words.

Wordwrite first sets the Modify bit in the buffer descriptor, indicating that a change has been made to the block. Note that execution of wordwrite does not necessarily cause the words modified to be written back onto disk. The actual data transfer does not take place until the buffer space must be released to bring in another block or until the buffer is flushed. When buffer space must be released, the least recently used block is written back if modified.

Wordread/wordwrite Routines (Continued)

The following example opens DATAFILE to file number 1 with BUFPTR pointing to the beginning of the buffer pool. The wordwrite procedure then accesses file position 200 and writes one word from the memory area pointed to by COREPTR to the appropriate position in the file buffer.

```
pointer BUFPTR, COREPTR;
   .
   .
   .
buffer(BUFPTR, MEMORY/2) ;
access (1, "DATAFILE", BUFPTR) ;
   .
   .
   .
wordwrite(1, 2ØØ, COREPTR, 1) ;
```

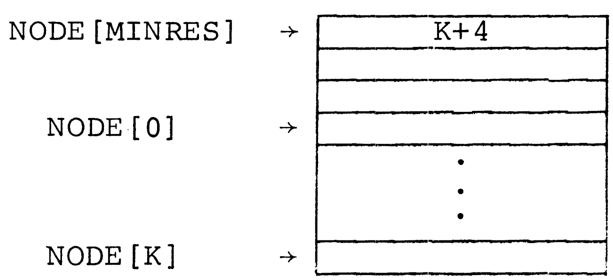
Accessing File 0 Nodes (noderead/nodewrite/nodesize)

As defined for Cache Memory Management, a node is an ordered set of data similar to an ALGOL array. In the ALGOL runtime, the lower bound of the nodal array is named MINRES and has a default value of -3. The upper value of the array, K, is defined by the user.

To use the default value of MINRES, the user declares MINRES as:

```
literal MINRES(-3);
```

The array NODE[MINRES:K] can then be represented as:



The user can change the default value of MINRES in an assembly language program. The maximum value of MINRES, however, is -1, allowing one word that will contain the size of the node. For example, to change the value of MINRES to -1:

```
         .ENT MINRES
         .ZREL
MINRES:  -1
         .END
```

Accessing File 0 Nodes (noderead/nodewrite/nodesize (Continued)

In that case, MINRES can be declared within the ALGOL program as *external integer* MINRES or as *literal* MINRES (-1).

Node access by CMM is only possible within a block of file number 0. All other blocks must be accessed using wordread/ wordwrite or hashread/hashwrite.

An entire node may be transferred by using the noderead/node-write routines. A single word within a node may be transferred using the fetch/stash routines. There is also a function, node-size, that returns the size of a given node.

When using noderead and nodewrite, the first 256 file addresses in a file are protected from user access. These locations can be used for storage of non-nodal data. When transferring data via fetch and stash, the first 256 file addresses can be accessed; however, an optional argument permits the user to protect these addresses from access.

noderead and nodewrite allow the programmer to transfer an entire node. The format of the call to noderead is:

```
noderead  (fileaddress, array);
```

where: fileaddress is the file address of the first word of a node. Access is inhibited if fileaddress is in the range 0 to 255.

array is a user-defined array into which the node is to be read.

Similarly, the format of the call to nodewrite is:

```
nodewrite (fileaddress, array);
```

where: fileaddress is the file address of the first word of a node into which the array is to be written. Access is inhibited if fileaddress is in the range 0 to 255.

array is a user-defined array containing the data to be written.

Accessing File 0 Nodes (noderead/nodewrite/nodesize (Continued)

Note that in both routines the parameter array must be the name of a user-defined array, not a pointer to the array, and that the user must set the contents of the first element of the array, MINRES, to the total count, K+4, before executing a nodewrite. (See examples.) Examples of the procedures are:

```
literal array A[MINRES:6], B[MINRES:10]; ←A and B are declared.
literal MINRES (-3);                     ←MINRES is declared.
    .
    .
    .
noderead(100,A);  ←Read into A starting at file address 100.
    .
    .
    .
K := 10;
B[MINRES]:= K+4;
nodewrite(200,B); ←Write from B into file starting at file
                   address 200.
```

nodesize is a function that allows the user to determine the number of words in a node. The format of the function is:

```
I:= nodesize(fileaddress);
```

where: fileaddress is the file address of the first word of the node.

nodesize reads the number of words in a node from the first word of the node and returns it as its value.

The following example reserves a buffer of 6*260+20 words, giving the CMM six 256-word buffers, with PTR1 pointing to the beginning of the buffer pool. The file DATA1 is then opened on file number 0. The nodesize function is used to return the size of the node at file address 400 into the variable SIZE200.

```
pointer PTR1;
integer SIZE200;
    .
    .
    .
buffer(PTR1, 6*260+20);
access(0, "DATA1", PTR1, 2);
    .
    .
    .
SIZE200 := nodesize(400);
```

Accessing a Single Word in a Node (fetch/stash)

If file number 0 is accessed, the user can read or write a single word in a node using the fetch function or the stash procedure.

The fetch function returns a single word in a node. The format of the function reference is:

```
i := fetch ([fileaddress,] offset);
```

where:

fileaddress is the file address of the first word of a node. If fileaddress is specified, the first 256 file addresses of the file are inaccessible to CMM as described below.

offset is the offset into the node of the word to be accessed. If no fileaddress is given, the offset is from the beginning of the file.

i is the integer identifier that is to contain the fetched value.

If fileaddress is specified, fetch returns the word at (fileaddress+offset-MINRES) as its value. This format does not permit accessing of the first 256 file addresses but allows the user to protect the first 256 elementsize words of a file from modification. These locations can be used for storage of special data, not to be changed during CMM use. If fileaddress is not specified, no checking of file addresses is performed and all addresses are accessible to CMM. In this case, the word at offset is returned.

The following example allocates a buffer one third the size of available memory (or less). PTR1 points to the beginning of the buffer pool. DATAFILE is then opened on file number Ø. The fetch function is used to return the value of 300+3-MINRES in the node that starts at file address 300 into NODE1.

```
pointer PTR1;
integer NODE1;
     .
     .
     .
buffer (PTR1, memory/3);
access (0, "DATAFILE", PTR1) ;
     .
     .
     .
NODE1 :=fetch ( 300, 3);
```

Accessing a Single Word in a Node (stash/fetch) (Continued)

stash writes a single word of a node to file number 0. The format of the call to stash is:

```
stash (i [,fileaddress] , offset);
```

where:

i — is the integer identifier whose value is written to file number 0.

fileaddress — is the file address of the first word of the node to contain the data. If fileaddress is specified, the first block of the file is inaccessible to CMM.

offset — is the offset into the node of the word to contain the datum. If no fileaddress is given, the offset is from the beginning of the file.

If fileaddress is specified in the stash call, the datum at i is written onto the disk file at (fileaddress+offset-MINRES). As with fetch, this format does not allow accessing of the first 256 file addresses. If offset is not specified, no checking of file addresses is performed and the first 256 file addresses are accessible to CMM. In this case, the single word at i is written onto disk at offset.

The following example allocates a buffer of 2000 words (or less) with PTR1 pointing to the beginning of the buffer pool. NODEFILE is then opened on file number 0. The stash procedure writes the value of VAL1 onto disk at the node that starts at file address 100.

```
pointer  PTR1;
integer  VAL1;
     :
buffer (PTR1, 2000);
access (0, "NODEFILE", PTR1) ;
     :
stash (VAL1, 100);
```

Clearing the Buffer Area (flush)

Once the user has completed modification of a file, the buffer area must be cleared using the flush procedure. The format of the call to flush is:

```
flush (pointer);
```

where: pointer — is the buffer pool pointer previously associated with the file in a call to access.

When a call to flush is executed, CMM writes onto disk all blocks that have been modified (as indicated by the Modify bit in the buffer descriptor); unmodified blocks are not written back onto disk.

Flush does not close a file. The user must explicitly close a file using the close procedure:

```
close (filenumber)
```

where: filenumber — is the user-assigned file number associated with the file to be closed.

Hashread/hashwrite Routines

Transfer of data from any file to core can be performed using the hashread procedure. The hashread procedure is particularly efficient for the transfer of files that have been hash-coded by the user; however, it is not necessary that the file be hashcoded to use this method of reading data from the file. Note that if the file to be hashread is hash-coded, the default elementsize must be used in opening the file for access via the access procedure; this provides that the file is one-word addressable.

The hashread procedure differs from other CMM read procedures in that it returns a pointer to the core address of the block of data and an offset into the block, so that the user has immediate access to the datum he may wish to modify.

Hashread/hashwrite Routines (Continued)

There is no special procedure for transferring hashread data back to the file. For example, the data may be transferred when the file is flushed or a hashread block may be transferred when all buffers are full and new data must be read in. If hashread data is modified while in core, the user must immediately set the modify bit in the buffer descriptor. The modify bit is set by issuing a hashwrite call. This insures that the proper data will be written whenever the block is written to the file. Failure to indicate modification of hashread data by a hashwrite call can wipe out a user program.

The hashread procedure brings a specified block of a file into core. The format of the call to hashread is:

```
hashread (filenumber,hashcode,block-pointer,block-offset);
```

where: filenumber is the number of the file to be read.

hashcode indicates the word within the file used to compute the requested file block.

block-pointer is a variable that will contain the core address of the block referenced by hashcore.

block-offset is a variable that will contain the offset into the block given by block-pointer.

The hashwrite procedure marks the last block referenced by a hashread as having been modified. The format of the call to hashwrite is:

```
hashwrite (buffer-pool-pointer);
```

where: buffer-pool-pointer is the buffer pool pointer defined in the buffer routine for the hashread file.

Hashread/hashwrite Routines (Continued)

The hashwrite procedure should be called immediately after data has been modified. The procedure does not write the modified block to the file; this is done when buffer space must be released to bring in another block.

The following example allocates a buffer one twelfth the size of memory with BPTR pointing to the beginning of the buffer pool. FILE4 is then opened on file number 4. Execution of the hashread procedure brings the file block, computed from the hash code at file address FILENODE, into core and returns value for BLOCKPTR and INDEX, so that the user can modify the block. The program checks as to whether the current value of the datum differs from the value that will replace the current value. If there is a difference, an assignment statement modifies the block and the user immediately indicates the modification in a hashwrite. If there is no difference, there is no need to perform a hashwrite.

```
integer FILENODE, INDEX, NEWVALUE, OLDVALUE; ←OLDVALUE is the
                                              current value of
                                              the datum and
                                              NEWVALUE is the
                                              modified value.

pointer BPOOL, BLOCKPTR;
based integer BI;

buffer (BPOOL, memory/12);
access (4, "FILE4", BPOOL);                  ←Access file to be
                                              hashread.
hashread (4, FILENODE, BLOCKPTR, INDEX);     ←Read block, obtain-
                                              ing pointers to
                                              datum to be mod-
                                              ified.

OLDVALUE :=(BLOCKPTR+INDEX)→BI;          }   ←Test whether OLD-
if OLDVALUE = NEWVALUE then go to DONE;  }    VALUE is equal
                                              to NEWVALUE.

(BLOCKPTR+INDEX) → BI := NEWVALUE;           ←Modify the datum
hashwrite(BPOOL);                             and set the mod-
                                              ification bit.

DONE:  OLDVALUE  := NEWVALUE;                ←Value of the
                                              datum is unchanged.
```

★ ★ ★

CHAPTER 10 -- COMPILER ERROR MESSAGES

All ALGOL error messages are printed out and are self explanatory. An up arrow (↑) points from the message to the source statement in which the error was detected.

The up arrow (↑) does not necessarily indicate the exact location of the error. It only indicates the character at which the error was detected. If no error is found where the arrow points, check to the left and the right of the arrow for a possible error. If an error still cannot be found, see if an earlier statement with an error could affect the statement so that the error was caused.

If the message

```
ER nn                                 ←nn is some number
```

should ever occur, notify Data General; this indicates a compiler error.

Some examples of error messages are:

```
I := 3*I+J;
         ↑
*** UNDEFINED VARIABLE ***
```

```
J := J J;
     ↑
***MISSING OPERATOR IN EXPRESSION ***
```

```
BEGIN REAL (I*2) X;
               ↑
*** PRECISION MUST BE AN INTEGER LITERAL ***
```

COMPILER ERROR MESSAGES (Continued)

```
I := I*S;
       ↑
*** ILLEGAL USE OF A STRING ***
```

```
BEGIN INTEGER I; REAL X, Y, I;
                            ↑
*** DUPLICATE SYMBOL DEFINITION ***
```

```
J : = J + * J;
        ↑
*** MISSING VARIABLE IN EXPRESSION ***
```

```
J := J+MATRIX;
       ↑
*** NO SUBSCRIPTS SPECIFIED ***
```

```
PRINT (2, I+4, X, SUBSTR);

*** ILLEGAL OPERAND OR PARAMETER ***
```

```
BEGIN REAL (32) X;
                ↑
*** PRECISION CAN NOT EXCEED 15 WORDS ***
```

```
I :     I := I+1;
  ↑
*** IDENTIFIER IS NOT A LABEL ***
```

COMPILER ERROR MESSAGES (Continued)

```
L[1]: I := I+1;
L[2]: I := I+1;
L[1]: I := I+1;
    ↑
*** DUPLICATE SUBSCRIPT ***
```

```
I := 93R8;
      ↑
*** ILLEGAL DIGIT FOR THIS RADIX ***
*** MISSING VARIABLE IN EXPRESSION ***   ←caused because illegal
                                          number was ignored.
```

```
I := I+;
       ↑
*** MISSING VARIABLE IN EXPRESSION***
```

```
I := J + IF I>Ø;
                ↑
*** EXPRESSION DOES NOT END PROPERLY ***
```

```
BEGIN REAL X; BOOLEAN B;

B := TRUE;
X := X+B;
        ↑
*** BOOLEAN IN REAL EXPRESSION ***
```

```
I := I? + 1;
       ↑
*** ILLEGAL CHARACTER ***
```

COMPILER ERROR MESSAGES (Continued)

```
 19     PROCEDURE X (I); INTEGER I; VALUE I;
                                    ↑
*** ERROR IN DECLARATION ***
```

```
PROCEDURE X(I); INTEGER I; VALUE I;
                           ↑
*** ILLEGAL SYNTAX ***
```

```
I := I + 2..;
         ↑
*** MORE THAN ONE DECIMAL POINT IN NUMBER ***
```

```
L1: I := I+L1;
           ↑
*** ILLEGAL USE OF LABEL ***
```

```
GO := 1.7;
↑
*** ILLEGAL USE OF RESERVED WORD ***
*** STATEMENT DOES NOT END PROPERLY ***
```

```
       FOR I := 1 UNTIL 1Ø STEP 1 DO
                  ↑
*** 'UNTIL' MUST FOLLOW 'STEP' ***
```

```
I := I + (IF (IF I>Ø THEN 1) ELSE 2;
                            ↑
*** PARENTHESES DO NOT BALANCE ***
```

COMPILER ERROR MESSAGES (Continued)

```
I := I+1                        ←no semicolon causes error in
I := I*3;                                  next statement
*** MISSING OPERATOR IN EXPRESSION ***
```

```
MEMORY OVERFLOW     ← A memory overflow occurs in phase 1 of
                      compilation.  Besides a true overflow of
                      memory, it may indicate that the user
                      omitted the second quotation mark that
                      would close a quoted string.
```

Run-time errors are described in Appendix C. Run-time error messages may, by option, be printed out in full or given as a numeric code.

★★★

CHAPTER 11 -- INCLUDING FILES FOR COMPILATION (*include*)

Extended ALGOL has a facility permitting users to bring in files in source language to be compiled as part of an ALGOL program. To do so, the keyword *include* is followed by a filename designating the file to be brought in for compilation at that point. *include* may appear in the declaration section of a program if the file to be compiled contains declarations or *include* may appear in the statement section of a program if the file to be compiled contains statements. In either case, the keyword *include* must follow either the keyword *begin* or the terminator semicolon. The keyword *include* cannot follow such keywords as *else, do, then,* etc.

When the keyword *include* is encountered during compilation, the file designated by the file name following the keyword will be compiled as part of the ALGOL program. When the file has been included, compilation of the program resumes at the declaration or statement following the *include*d file. The name of the file must be complete, including any extensions to the file name. Unlike file names that appear as literals in procedure calls, the file name appearing in the *include* does not have to be in quotation marks.

The *include*d file must consist of complete declarations or statements; a file terminating in a partial statement or declaration cannot be used.

The *include* keywords cannot be nested, for example, an *include* keyword cannot appear in a file that is to be *include*d.

```
begin include DECLARE.AL;        ←DECLARE.AL contains declarations
                                  for the program to be compiled.
open (0, "INFILE", ERR);
     .
     .
     .
end
```

```
begin integer array B[1,100];
      integer I,J,K;
      include STMTS.AL;          ←STMTS.AL contains statements
                                  for the program to be compiled.
end
```

CHAPTER 12 -- DIFFERENCES BETWEEN EXTENDED ALGOL AND STANDARD ALGOL

EXTENSIONS TO STANDARD ALGOL

external procedures and variables.

Character string variables and arrays. String manipulation using index, length, ascii, and substr built-in functions.

Bit manipulation using binary and octal literals and the shift and rotate built-in functions.

I/O routines providing free-form read and write, random-record read and write, formatted output, and cache memory I/O.

based and *pointer* variables for efficient addressing. Library routines used in pointer addressing are address, allocate, and free.

Subscripted labels.

Functions that return array data: hbound, lbound, and size.

literal declaration. *operator* declaration.

include keyword.

Data type conversions of the form:

integer to *boolean* and *boolean* to *integer*
integer to *pointer* and *pointer* to *integer*
string to *integer*, *real*, *boolean*, or *pointer*
integer, *real*, *boolean*, or *pointer* to *string*

Conversion to any radix from two through ten.

File manipulation library routines, rename and delete.

xor boolean operator.

LIMITATIONS OF EXTENDED ALGOL

Data types must be declared for all parameters.

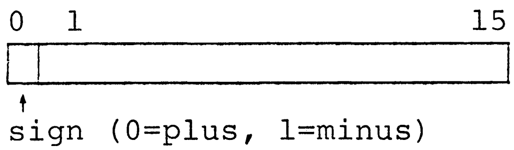
Division of an integer by an integer produces an integer result.

APPENDIX A

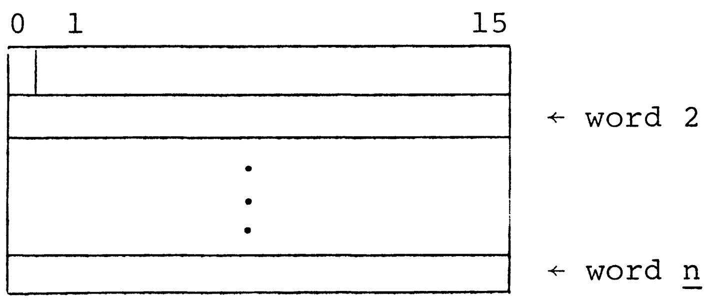
DATA TYPE REPRESENTATION

INTEGERS

Integers are stored in packed, twos complement form. Single-precision integers are one word long.



Multi-precision integers can be defined by giving the number of words of precision in the integer declaration.



Single-precision integers (and pointers) are designed to provide the greatest efficiency in speed of calculation and in amount of core required. To provide this efficiency, no checking for overflow is done; overflow will cause erroneous results.

Multi-precision integers are checked for overflow. The result, if overflow occurs, will be the largest possible number that can be stored. To force overflow checking of single-precision integers declare the integer with a precision of 1.

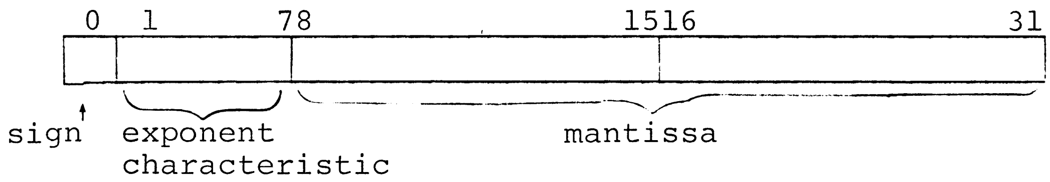
```
integer (1) x;
```

The limit of a single-precision integer is

$$2^{15}-1 = 77777_8 = 32{,}767_{10}$$

REAL (FLOATING-POINT) NUMBERS

Real numbers of default precision are stored in two words.



The contents of bits 1 through 7 are interpreted as an integral exponent in excess 64 (100_8) code. Exponents from -64 to +63 are therefore represented by the binary equivalents of 0-127 ($0-177_8$).

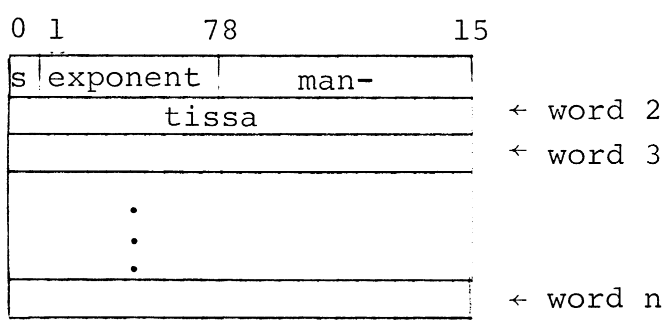
100_8 represents an exponent of 0

177_8 represents an exponent of 63_{10}

1 represents an exponent of -63_{10}

The mantissa is treated as a hexadecimal fraction between .0625000 and .9999999. All floating-point numbers are maintained in normalized form. Default real numbers have 6 or 7 decimal digits of significance, depending upon their normalized hexadecimal representation. Negative mantissas are identical to their positive counterparts, except that the sign bit is 1 instead of 0. Any real number having a mantissa of all zeroes will be represented in true zero form with bits 0-31 set to zero.

The precision of real numbers can be set to a number of words up to a limit of 15. The additional words are used to expand the mantissa.



It is also possible to declare a one-word real number. Note, however, that only very small (2 or 3 decimal digit) mantissas are possible in a one-word real number.

BOOLEAN DATA

A Boolean datum is stored in a single word. If the word contains all zeroes, the Boolean value is false; otherwise, the value is true. If a Boolean value of true is being stored into a word, that word will be set to one; however, any single word that does not consist of all zeroes will be interpreted as the Boolean value of true.

```
0                15
[                  ]
```

POINTER DATA

A pointer datum is stored in a single word. The word contains an address. The pointer datum resembles a positive integer of default precision.

```
0                15
[                  ]
```

STRINGS, NUMERIC ARRAYS, AND ARRAYS OF STRINGS

Previously described data types are of fixed length and are stored on a user's stack in the formats given. When the length of a datum may vary-- such as an array of any type or a string -- the information stored on the user's stack merely describes the datum and points to the beginning location of storage to be generated as needed for the datum at run time. The information about the datum is called a specifier. Storage of data or varying length is described in Appendix B, page B-4 and following.

* * *

APPENDIX B

THE RUN-TIME STACK

ALGOL run-time stack discipline is described in this appendix. A full description of the run-time routines used to maintain the stack is given in Appendix C.

RUN-TIME STACK

After loading ALGOL source program binary tapes and the ALGOL run-time library routines, the beginning address of available memory is set by the Extended Relocatable Loader as NMAX (non-zero maximum). NMAX, together with the end-of-memory address, is used by the ALGOL initialization routine, SPINIT, to initialize three stack areas and an area for permanent allocation.

The first stack is a data block of 50 octal words used by the run-time routines as temporaries. The stack address is pointed to by a page zero word .RP. The .RP pointer to the stack is bumped by the number of temporaries a given routine requires at run-time. This insures that a called routine has, essentially, a free temporary area and is not using the same temporary as the calling routine.

The second stack is a number stack, which is allocated only when arithmetic routines or functions, the I/O package, or number conversion routines are loaded. The stack is a block of 200 octal words used to push or pop numbers in unpacked form to be used by the number routines. The number stack is allocated, for example, when routine FADD (floating addition), function EXP (exponentiation), or routine STCV (string-number conversion) is required. The stack is pointed to by page zero location, NSP.

The third stack is the ALGOL stack, used by ALGOL procedures and most run-time routines for variables, arrays, strings, and pertinent information such as procedure level and hardware registers. Different areas of this stack are pointed to by three page-zero words, .FP, QSP, and .SP. The ALGOL stack will frequently be referred to as the stack.

The permanent allocation area initially has zero length. It is found at the end of memory and is pointed to by page zero location, .SSE. .SSE is used for the allocation of data to *own* arrays and *own* strings. As space is allocated, the address of .SSE is pushed back in memory, reducing the size available for the stack. In general, the area is never released for other use.

RUN-TIME STACK (Continued)

A general diagram of memory allocation after initialization is:

.LOC 1000 ↓	ALGOL User Programs	
	ALGOL Run-Time Routines	
.RP ↓	Temporaries' Stack	
NSP ↓	Number Stack (if allocated)	
	ALGOL Stack - contains the main portion of program data	stack shortens as permanent area is allocated and .SSE pointer is moved back from the end of memory
↑ .SSE		End of Available Memory.

ALGOL STACK

The stack is actually a list of sub-stacks, called user stack frames. Each stack frame has the same format but a variable size. Stack frames are allocated during run time for use by each ALGOL procedure. They contain storage for the variables, arrays, and strings of the procedure, and certain information required by the procedure. The stack frame is divided into three sections: the fixed area, which contains information required by the procedure; the variable area in which are stored variables of fixed length (assigned storage); and the allocated variable area in which data for arrays and strings are stored (allocated storage).

The fixed area is allocated at the top of the stack, which is set at -200_8 words from the frame pointer (.FP). The fixed area contains eleven octal words as follows:

ALGOL STACK (Continued)

-200	Stack frame length
-177	Previous stack address pointer
-176	Current stack level (of procedure)
-175	Stack parameter list pointer
-174	Save for hardware Carry
-173	Save for hardware register 0
-172	Save for hardware register 1
-171	Save for hardware register 2
-170	Subroutine return location address

Since .FP points 200 octal words past the first word of the frame, the remainder of the page, -167 to +177 octal words or 366_8 words are available for the variable area (assigned variables). However, more or less space can be used for the variable area as needed. If less than 366_8 words are used, the remainder will become part of the allocated variable area.

If more then 366_8 words are needed for the variable area, a second pointer, .SP, is used. .SP points to offsets from .FP in increments of a page (377_8 words) and is set by the run-time routine GETSP, described in Appendix C. The sub-sections of 377_8 words are denoted as sublevels and passed as parameters to GETSP.

The last storage area of the user stack frame is called the allocated variable area and is used to store data for arrays and strings. The allocated variable area is divided into blocks, each block corresponding to a block in the ALGOL program.

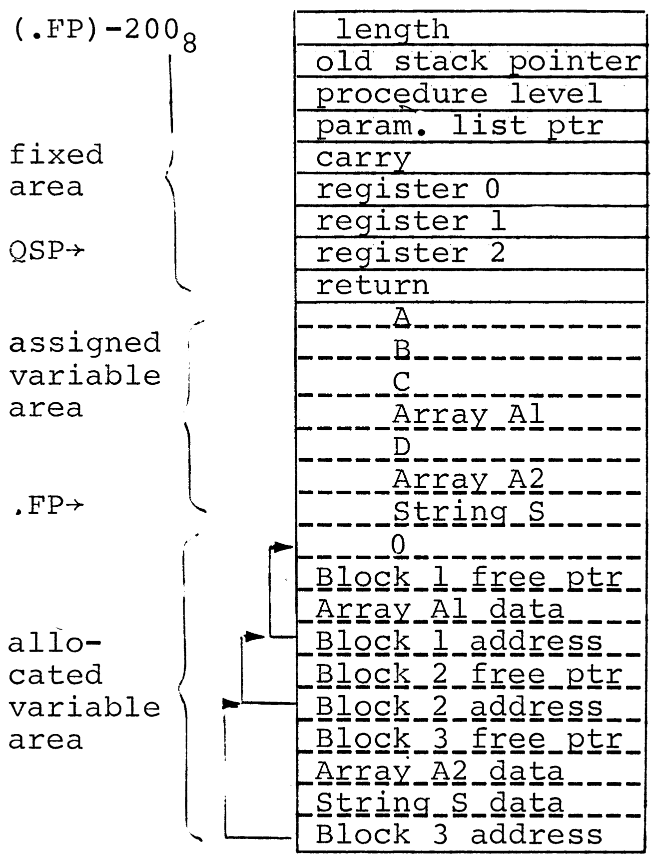
At the beginning of each block is a two-word area. The first word is used for freed string data area pointers (described later). The second word is a pointer to the beginning of the previous block. In the case of the first allocated variable block, the pointer is set to the beginning of the allocated variable area. As data areas are allocated within the block, the address pointer (second word) is pushed down the list while the free list pointer is untouched.

When a block is exited, the block start address is loaded and the previous one is reset, thereby deleting all arrays and strings allocated in the block just terminated. This provides for dynamic allocation and freeing of areas within blocks for arrays and strings.

On the page following is a portion of ALGOL source code and the

ALGOL STACK (Continued)

portion of the stack which would be built to correspond to the code. Note that data stored in the variable area and in the allocated variable area will vary in length; the use of one line for each datum does not mean allocation of one word to each datum.




```
begin integer A, B, C;
      integer array A1[10];
            .
            .
            .
   begin integer D;
            .
            .
            .
      begin real array A2[3];
            string S;
            .
            .
            .
      end;
            .
            .
            .
    end;
            .
            .
 end;       .
```

In the diagram of the stack frame, a thread pointer, QSP, is indicated, pointing to the register 2 save storage. QSP is the "quick stack pointer" used by run-time routines to save the AC's without destroying the contents of any.

All page zero requirements for run-time (such as .FP) are allocated in a relocatable library routine called ZERO. ZERO contains entries to all the page zero writable data.

ASSIGNED AND ALLOCATED STORAGE OF THE STACK

Execution of a SAVE call, described in Appendix C, causes a new

ASSIGNED AND ALLOCATED STORAGE OF THE STACK (Continued)

stack frame to be created with an initial size equal to that needed for the fixed information and for assigned storage. The size needed is determined from the first word following the call to SAVE.

The coding of the SAVE contains the level of the procedure, where 1 is the first or outer procedure, 2 is the first level of internal procedures, etc.; the number of parameters, including any function return value; and a list of parameter descriptors.

A parameter descriptor consists of two words: the parameter address and a parameter specifier. The parameter specifier contains the information that identifies all necessary characteristics of the parameter.

In the coding below, BESSEL is an outer procedure (level 1) that has three parameters, the function return value BESSEL, X, and N.

```
; REAL PROCEDURE BESSEL (X,N);

        .TITL   BESSEL

        .EXTU        ← defines all external displacements in
        .ENT    BESSEL                                program
        .EXTN   EXP
        .EXTN   ALG
        .EXTN   FM
        .EXTN   FD
        .EXTN   FENTL

        .ZREL
.LP:    LP+200

        .NREL
        .TXTM   1

BESSEL: JSR     @SAVE
        FSØ      ← initial frame size
        1B7+3    ← 1 = procedure level; +3 means 3 parameters
        SP+Ø     ← address       ;BESSEL          } Descriptor of
        ØØ3442   ← specifier     ;REAL PARAMETER  } BESSEL
        SP+2     ← address       ;X               } Descriptor
        ØØ2Ø42   ← specifier     ;REAL VALUE      } of X
        SP+4     ← address       ;N                 } Descriptor
        ØØ1Ø21   ← specifier     ;INTEGER PARAMETER}  of N

;         VALUE X; REAL X; INTEGER N;
```

Parameter Descriptor Address Word

The first word of each parameter descriptor is a parameter address of the form:

SP	n

where: SP = a positive stack offset. If set to 0, the address of a scalar is a machine address. If the bit is 1, the address refers to a word in the current or previous stack frame that points to the actual data (or to an array or string as described later).

n = an offset in words to the frame pointer for the current or previous frame. n indicates the offset for the parameter. In the previous example, the first parameter of the SAVE call has an offset of 0, the second of 2, and the third 4.

As previously indicated, .SP is the same as .FP as long as the assigned storage area requires only a single page (the zeroeth page of the stack frame). This is true for the example given.

If more than one page is needed for assigned storage, run-time routine GETSP is called to add a page to the stack frame and move the temporary pointer .SP to that page as shown in the diagram:

page 0 of stack frame	.SP = .FP
page 1 of stack frame	.SP = .FP+1×377_8
page 2 of stack frame	.SP = .FP+2×377_8
⋮	
page n of stack frame	.SP = .FP+n×377_8

Parameter Descriptor Specifier Word

The second word of the parameter descriptor is a specifier that contains all the information needed to define the parameter and set aside the proper amount of storage. The

Parameter Descriptor Specifier Word (Continued)

parameter specifier has five fields:

0	1-3	4-7	8-11	12-15
I	shape	storage class	data type	precision

where: I is set to 1 to indicate the parameter descriptor address word is indirect.

The specifier fields define the parameter. Precision shows the number of words of precision. For a string, precision is either 2 or 3 depending upon the size of the string specifier in assigned storage. Meaning of other field contents is given below:

Shape	Storage Class	Data Type
0 scalar	0 local	0 undefined
1 array	1 own	1 integer
2 program	2 parameter	2 real
3 procedure	3 based	3 boolean
	4 value	4 label
	5 external	6 pointer
	6 built-in function	7 multi-precision
	7 function value	8 string

Contents of Assigned Storage

Information in the parameter specifier determines how many words are set aside for the parameter in assigned storage. Numeric scalars are stored in their entirety in assigned storage. If the scalar is a real datum of default precision, 2 words are used to store the datum; if the specifier indicates a real datum of 6-word precision, 6 words are used to store the datum. Appendix A shows the format in which scalar numerics are stored in assigned storage.

However, each array has two words in assigned storage and each string has either two or three words in assigned storage. The assigned storage for arrays and strings has a fixed format; array and string data is allotted in the allocated storage area.

The two words for an array in assigned storage are called an array specifier. Both words point to areas of allocated storage. The first word points to array data; the second word points to

Contents of Assigned Storage (Continued)

another area in allocated storage containing the control table for the array. The table is called the array dope vector.

Each string has two or three words in assigned storage, called a string specifier. The first word is a pointer to the first byte of the string in allocated storage. The second word (if the string is ≤ 127 characters) contains the current and maximum length of the string. The short string specifier has the form:

byte pointer			
0	maximum length		current length
0	1 – 7	8	9 – 15

unused (bit 8)

If the string is greater than 127 characters, the second word of the specifier contains the maximum length and the third word contains the current length.

Each substring has two or three words in assigned storage, called a substring specifier. The first word is a pointer to a word in allocated storage that contains the string specifier. (The string specifier of a substring is in allocated storage.) If the string that is being subset is ≤ 127 characters, the second word contains an index into the string that shows the starting character of the string and the length of the substring in characters.

If the string is greater than 127 characters, the index is contained in the second word and the length in the third word.

word address			
1	index to string		length of substring
0	1 – 7	8	9

unused (bit 8)

word address	
1	index to string
	length of substring

unused (bit 0 of third word)

Contents of Assigned Storage (Continued)

The address for a substring specifier is a word address; the string specifier has a byte address.

The indirect bit is set for a substring; the direct bit for a string.

The length of the string, not the length of the substring, determines whether the substring specifier is two or three words long.

Contents of Allocated Storage

As described in the section 'Run-Time Stacks', allocated storage for a block is created when a block is entered at run-time. It contains one word for the allocated storage free list, dope and data for arrays and strings, and a terminal word pointing to the beginning of allocated storage for the block.

The allocated storage free list is described in detail in Appendix C in the section 'Routines that Perform Allocation to Run-Time Stacks'.

The threaded block pointers, as indicated on page B-3, assure a proper return through a number of block levels. When a block is entered, the allocated storage free list and the block pointer are created whether or not the block contains any declarations of arrays or strings.

0
allocated storage free list
end of stack

When a block containing data for allocated storage is entered, the allocated storage area appears as shown:

1. Block A entered

0
allocated storage free list
arrays and strings for block A
block A pointer

Contents of Allocated Storage (Continued)

When a second block is entered:

2. Block B entered

0
allocated storage free list
arrays and strings for block A
block A pointer
allocated storage free list
arrays and strings for block B
block B pointer

Array Information in Allocated Storage

Two areas in allocated storage are set aside for an array. The first is the data area calculated to be needed for the array. The second is the area for array dope.

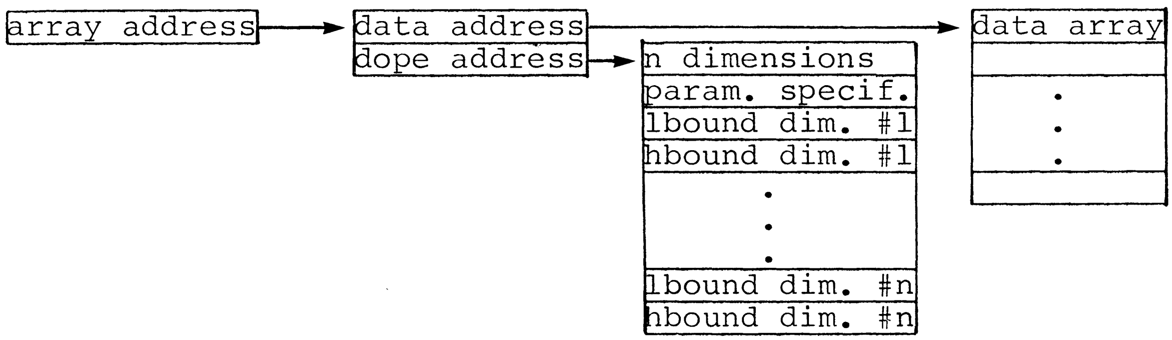
Array dope is a variable number of words, depending upon the number of dimensions of the array. The first word contains the number of array dimensions, the second is the parameter specifier, and the remaining words of array dope contain the dimensions. No specifiers are given for the dimensions, since they have been converted to single-precision integers.

The fields of the parameter specifier are identical to those of the scalar parameter specifier described earlier. The shape field will be 1, indicating an array.

The run-time routine ARRAY is used to build array information and is described in Appendix C. In the coding of the call to ARRAY, array address is a pointer to the array specifier in assigned storage. A diagram of array information as it is stored in assigned and allocated storage is:

Array Information in Allocated Storage (Continued)

CODING OF ARRAY   ASSIGNED STORAGE   ALLOCATED STORAGE



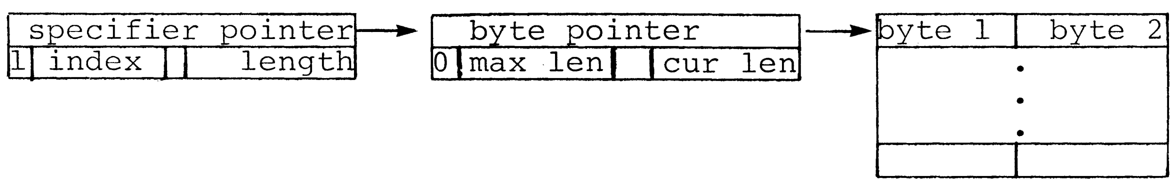
The data area of an array of strings contains the two or three-word specifier for each string. The first word of the specifier is a byte pointer, followed by one or two words containing the current and maximum lengths of the string.

Scalar String and Substring Information in Allocated Storage

For a string scalar, enough words are reserved in allocated storage to store the maximum byte length of the string. For example, if the maximum number of characters in the string is 25, 13 words will be reserved.

For a substring, the string specifier (defined in the section on assigned storage) is in allocated storage and points to the allocated area that is used to store the string, as shown below.

ASSIGNED STORAGE



If a substring is taken of a long string, three words of allocated storage are needed for the string specifier.

Based Arrays and Strings in Allocated Storage

When a based array or string is referenced, its array or string specifier, respectively, is built dynamically.

The first word of a based string specifier is a byte pointer corresponding to the pointer variable used to reference the based string. For example, for p→x, the first word of character data begins at the word specified by p. The pointer is followed by the current and maximum lengths, but in a based string the current length is always set equal to the maximum length. The byte pointer points to the first word of string data.

Except for building the array specifier dynamically, based arrays of scalars and strings resemble local arrays. The based array of strings has a data area with a specifier for each string in the array. The strings are word aligned.

OWN AND EXTERNAL STORAGE

As described in the section "Run-Time Stack", storage for *own* strings and arrays is handled in a separate area that is grown by moving pointer .SSE down in available memory.

Assigned storage for *own* and *external* variables is in page zero. Both assigned and allocated *own* storage contain the same data as that stored for local variables in the ALGOL stack. *external* variables are allocated by the user in assembly language.

★ ★ ★

APPENDIX C

RUN-TIME ROUTINES

Following are descriptions of all routines used to allocate and manipulate the run-time stack. On return, all routines restore AC0, AC1, AC2, and Carry, and set AC3=.FP.

In describing run-time routines, the following conventions are used:

- desc - A parameter descriptor consisting of two words -- an address and a parameter specifier.
- n - An integer used to represent a count of items to follow, such as parameter descriptors.

STACK ALLOCATION AND DEALLOCATION ROUTINES

Routine	Coding	Meaning
CALL	JSR @CALL subr n $desc_1$ . . . $desc_n$	CALL saves the subroutine return address (pointing to n) which points to the subroutine parameter list of descriptors. Note rules given under SAVE if parameter descriptors differ in properties or number between the CALL and the SAVE lists.
SAVE	JSR @SAVE size level+n $desc_1$ . . . $desc_n$	SAVE builds a new user stack frame, having an initial size in words equal to size (the assigned storage size). SAVE copies the parameter descriptors into assigned storage with the following limitations: If one or more parameter descriptors differs in properties between the CALL and SAVE lists, the descriptor(s) will be converted to those of the SAVE descriptor and later converted back. If the number of parameter descriptors differs between the CALL and SAVE lists, the number stored will be the shorter list.

STACK ALLOCATION AND DEALLOCATION ROUTINES (Continued)

Routine	Coding	Meaning
SAVE (Continued)		SAVE sets the end of stack and stack level (level = internal procedure depth), sets the old stack pointer to the previous stack level, saves accumulators and Carry, and jumps to the location following the last parameter descriptor.
RETURN	JSR @RETURN	RETURN destroys the current stack frame, restoring AC0-2 and Carry. RETURN checks for top of stack (an illegal return call) and sets .SP= .FP with AC3 set to .FP for the reset previous stack. RETURN returns all parameters passed to the previous stack, following the rules described in SAVE, and returns to the location following the last CALL parameter descriptor.
ABRETN	JRS @ABRETN loc	ABRETN destroys the current stack and restores parameters as described for RETURN and then jumps to the location specified by loc. ABRETN returns normally if loc points to 0. (To be used in place of RETURN for abnormal return to a label.)
ASAV	JRS @ASAV	ASAV stores the accumulators on the stack and sets the parameter list pointer. ASAV returns to the location after the jump. It does not set AC3 to .FP and requires that it be called via a 'JSR' through a page zero word.
ARET	JMP @ARET	ARET returns from a routine which called ASAV. It restores AC0-2, sets AC3 to .FP, then jumps to the address in the parameter list pointer on the stack. Caution: the caller should have bumped the list pointer address to the end of the parameter list.
RSAV	JMP @RSAV size	RSAV is a quick form of SAVE that passes and expects n parameters.

STACK ALLOCATION AND DEALLOCATION ROUTINES (Continued)

Routine	Coding	Meaning
RSAV (Continued)		Frame size of the required number of temporaries is found in size. RSAV returns to the word after size.
RRET	JMP @RRET	RRET returns from a quick save. It restores the accumulators and the previous stack, then returns to the address in the previous stack's return.
BLKSTART	JSR @BLKSTART	When a new block is added, BLKSTART sets the two-word marker.
BLKEND	JSR @BLKEND	BLKEND removes the block marker last set by BLKSTART, which destroys the current block. Finding a 0 denotes an illegal block-end error.
GETSP	JSR @GETSP level+sublevel	GETSP determines the current .SP searching for level+sublevel (zeroeth page of the stack frame + pages of offset as defined in Appendix B) or until level 0 is found. GETSP then adds one (another page) to sublevel and places the new .SP in AC3. GETSP checks to insure that: next-level - last-level = 0 or 1
SPINIT		SPINIT initializes the run-time stacks. There is no coding; the starting location is 777_8.

In the routines above, there are two ways in which a stack can be allocated. The first uses routines CALL, SAVE, and RETURN. This method is used by ALGOL for all internal and external procedures, including the I/O procedures. CALL, SAVE, and RETURN expect a parameter count and parameter descriptors (or a 0 for no parameters.)

STACK ALLOCATION AND DEALLOCATION ROUTINES (Continued)

The CALL-SAVE-RETURN method has the following two forms:

(1)

```
        JSR   @CALL
        subr
        n
        desc1
          .
          .
          .
        descn
```

(2)

```
        JSR   @CALL
        subr
        0
```

---

(1)

```
subr:   JSR   @SAVE
        size
        level+m
        desc1
          .
          .
          .
        descm
          .
          .
          .
        JSR   @RETURN
```

(2)

```
        JSR   @SAVE
        size
        level+0
          .
          .
          .
        JSR   @RETURN
```

In form 1, if n in CALL and m in SAVE are equal, all parameters will be passed to subr's stack frame; if not equal, the shorter list count is passed. If either m or n is zero, no parameters are passed. In form 2, both m and n are zero and no parameters are passed. In either form, RETURN will return the same number of parameters that were passed by the SAVE routine.

The second method of stack allocation uses routines RSAV and RRET and expects no parameters. In this method, the routine must pick up all parameters itself and reset the return pointer in the previous stack to the first word following the parameters. The parameter list is pointed to by the return

STACK ALLOCATION AND DEALLOCATION ROUTINES (Continued)

location in the previous stack. The calling sequence for the RSAV-RRET stack allocation method is:

```
       JSR   @subrp                   ← page zero pointer to subr.
       desc1 ⎫
        .    ⎪
        .    ⎬                        ← descriptors if any
        .    ⎪
       descn ⎭
----------------------------------------------------------------
subrp: subr                           ← in page zero
----------------------------------------------------------------
subr:  JMP   @RSAV
       size
        .   }                         ← routine called
        .
       JMP   @RRET                    ← return from subr
```

The ARET and ASAV routines are used by most of the run-time routines. They do not allocate a stack but simply save the accumulators and set the stack parameter list pointer. The calling sequence is:

```
       JSR   @subrp                   ←page zero pointer to subr
       desc1
        .
        .
        .
       descn
----------------------------------------------------------------
subrp: subr                           ← in page zero
----------------------------------------------------------------
subr:  JMP   @ASAV
       LDA   3,.FP                    ← ASAV does not load AC3
        .
        .
        .
       JMP   @ARET
```

STACK ALLOCATION AND DEALLOCATION ROUTINES (Continued)

The parameter list pointer must be bumped to the return location (usually done while picking up parameters) for the return.

The remaining routines described in this appendix are usually found in procedures which use the CALL-SAVE-RETURN method of stack allocation. For example, SETCURRENT is a routine described in the "General Purpose Routines" section. This routine uses a three-word string specifier.

```
;SETCURRENT
;
;SETS THE CURRENT LENGTH FOR A STRING UNLESS
;IT EXCEEDS THE MAXIMUM (WHICH SETS CURRENT = MAXIMUM).
;
;  ** IGNORES ALL SUBSTRINGS **
;
;CALLING SEQUENCE
;
;       JSR     @CALL
;       SETCURRENT
;       2
;       STRING DESCRIPTOR
;       CURRENT DESCRIPTOR   (INTEGER)

                .TITLE  SETCURRENT
                .ENT    SETCURRENT
                .EXTU
                .NREL

S=      -167                ;STACK ARGUMENT DISPLACEMENT
SP=     1B0-S               ;STACK ARGUMENT INDICATOR
STR=    S                   ;STRING ARGUMENT
CNT=    STR+3               ;CURRENT LENGTH
CSIZE= CNT+1-S              ;STACK FRAME SIZE
STRLOC=10B11+3              ;STRING SPECIFIER
INTVAL=4B7+1B11+1           ;INTEGER VALUE SPECIFIER
```

STACK ALLOCATION AND DEALLOCATION ROUTINES (Continued)

```
SETCU:  JSR     @SAVE       ;ALLOCATE A NEW STACK FOR SETCURRENT
        CSIZE               ;ALLOCATE THIS MANY DATA WORDS
        2                   ; PASS THE FOLLOWING TWO ARGUMENTS
        SP+STR              ;   THE STRING
        STRLOC              ;     STRING LOCAL SPECIFIER
        SP+CNT              ;   CURRENT LENGTH
        INTVAL              ;     INTEGER VALUE SPECIFIER
        LDA     Ø,STR+1,3   ;GET THE SECOND WORD OF THE STRING
        MOVL#   Ø,Ø,SZC     ;IS THE STRING A SUBSTRING?
        JMP     SKIP        ;YES, IGNORE THE CALL.
        LDA     Ø,CTN,3     ;GET THE NEW CURRENT LENGTH
        LDA     1,STR+1,3   ;GET THE STRING MAXIMUM LENGTH
        SUBZ#   Ø,1,SNC     ;SKIP IF THE CURRENT <= MAXIMUM.
        MOV     1,Ø         ;NO, SET THE CURRENT TO MAXIMUM
        STA     Ø,STR+2,3   ;STORE THE NEW CURRENT LENGTH.
SKIP:   JSR     @RETURN     ;RETURN TO THE CALLER.

                .END
```

ROUTINES THAT PERFORM ALLOCATION TO THE RUN-TIME STACKS

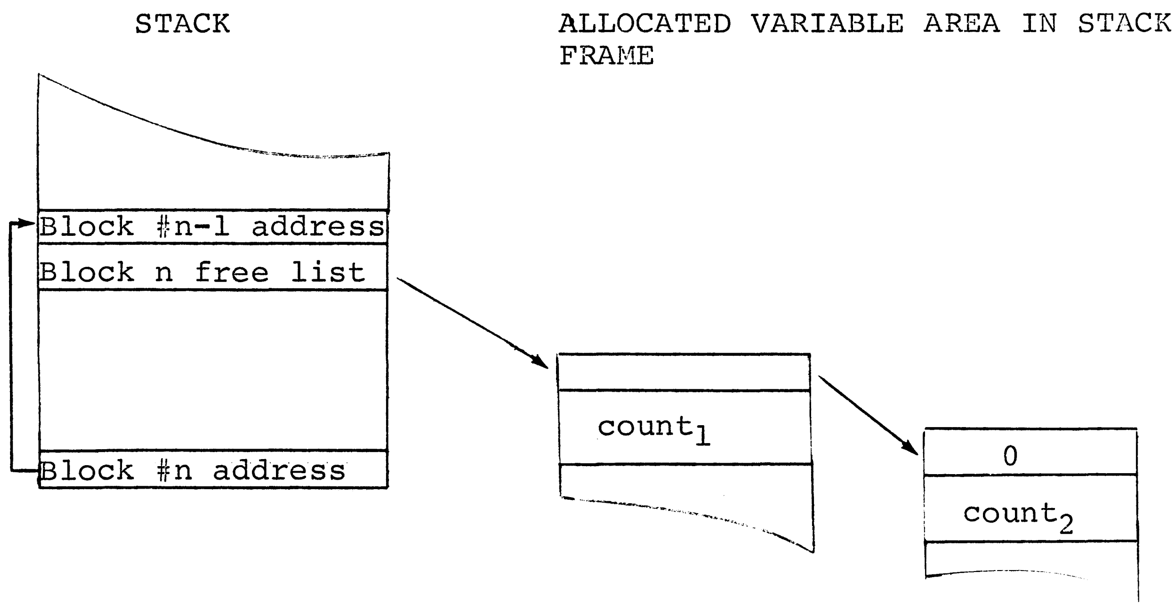
The three basic forms of allocation are for arrays, strings, and data buffer areas used with pointers (*based* data). In addition to the desc and n conventions used for a parameter descriptor and a count of items, this section uses the following convention:

loc - a single word containing an address

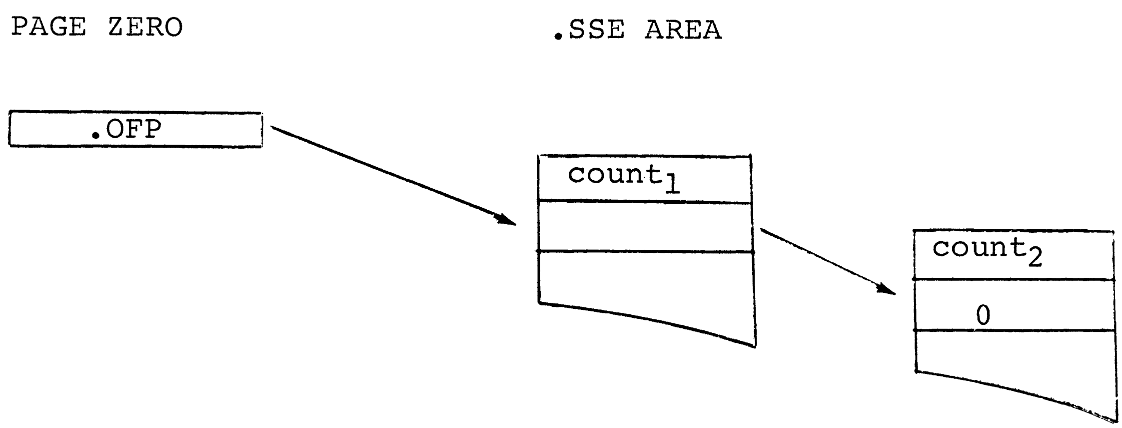
Areas allocated to strings and to *based* data can be freed when the allocated area will not be needed again. Routines used to allocate areas to strings and *based* data will check the free list to see if a previously allocated area is available for use.

The string free list for local data is a list of addresses whose second word is the word count of the data area. A zero always ends the free list pointers. For example:

ROUTINES THAT PERFORM ALLOCATION TO THE RUN-TIME STACKS (Continued)



The format of the free list for *own* data (see ALLOCATE and FREE routines in this section) is similar to the string free list for local data, except that the count is always kept in the address minus one word. Thus, count+1 words are allocated. For example:



ROUTINES THAT PERFORM ALLOCATION TO THE RUN-TIME STACKS (Continued)

Routine	Coding	Meaning
ARRAY	JSR @ARRAY n array-loc $dim\text{-}loc_1$ $dim\text{-}loc_2$ . . . $dim\text{-}loc_{n-1}$	ARRAY sets up the two-word specifier in assigned storage, builds the control table (array dope) in allocated storage, calculates the data area needed in allocated storage, and adjusts the end of the user stack. array-loc is the word pointer to the array specifier in assigned storage. If bit 0 is set to 1, bits 1-15 of array-loc may contain a displacement to be added to .FP. While adjusting the stack in accordance with the array, ARRAY performs a series of checks on dimensions, array specifier, and the stack. Note that the dimensions do not need specifiers, since they are assumed to be integer. ARRAY restores all registers, and sets AC3=.SP. See Appendix B for a graphic representation of array information stored in the user stack.
ALLOCATE	JSR @CALL ALLOCATE 2 $desc_1$ $desc_2$	ALLOCATE allocates an area of n words, pointed to by $desc_2$ (integer) in the *own* area and sets the starting address in $desc_1$ (pointer).
FREE	JSR @CALL FREE 1 desc	FREE frees an area, allocated by ALLOCATE, that has the pointer desc. desc is the address of the data area. The free list for this routine is a page zero pointer called .OFP.
SALLOC	JSR @SALLOC desc n	SALLOC builds a string specifier in assigned storage, pointed to by the address in desc and then allocates an area of n/2 words, setting maximum character count to n. Allocation is performed by the STCOM routine.

ROUTINES THAT PERFORM ALLOCATION TO THE RUN-TIME STACKS (Continued)

Routine	Coding	Meaning
SARRAY	JSR @SARRAY desc n	SARRAY builds and allocates a string, in the same way as SALLOC, for each element of the string array pointed to by desc. The maximum character count is n. Allocation is performed by the STCOM routine.
STCOM	JSR @STCOM	STCOM allocates n/2 words to the allocated storage area for all strings except *own* strings. STCOM allocates n/2 words for *own* strings by bumping back .SSE by n/2 words. Parameters are passed in the run-time temporary stack as follows: .RP+1 string address .RP+2 string parameter specifier .RP+3 byte count (n) .RP+5 stack length .RP+7 stack allocation area pointer
SFREE	JSR @SFREE desc	SFREE places the string pointed to by desc on the string free list. Only the data area is freed. If a substring is detected, the associated string is freed.

GENERAL PURPOSE ROUTINES

These routines are either called by the user to produce a special result (non-I/O) or by ALGOL source programs to facilitate run-time functions that do not involve floating point. Some of the routines, such as mod, random, rem, and seed, are not included in the reference manual as user calls since they do not follow the type-conversion conventions of general functions. However, they can be used, if desired. For example, the user can declare:

external integer procedure random;

and then call random from his program.

GENERAL PURPOSE ROUTINES (Continued)

Routine	Coding	Meaning
SUBSCRIPT	JSR @SUBSCRIPT n computed-loc array-loc subscript-loc_1 subscript-loc_2 . . . subscript-loc_{n-1}	SUBSCRIPT computes the address of the data for the array (computed-loc), using the algorithm given below.

Subscript Algorithm

If the position value of a dimension of array A is n, the upper (U) and lower (L) bounds of that dimension can be written as:

$b_{Ln}:b_{Un}$

and the maximum value that the dimension can assume is:'

$b_{MAXn}= b_{Un}-b_{(Ln)-1}$

For any given dimensionality, the subscript value of any subscript and the maximum subscript value are shown below:

DIMENSION AND FORMAT	SUBSCRIPT VALUE	MAXIMUM VALUE
1 $A[b_{L1}:b_{U}]$	b_1	b_{MAX1}
2 $A[b_{L1}:b_{U1},b_{L2}:b_{U2}]$	$b_1+b_{MAX1}\times(b_2-1)$	$b_{MAX1}\times b_{MAX2}$
3 $A[b_{L1}:b_{U1},b_{L2}:b_{U2},b_{L3}:b_{U3}]$	$b_1+b_{MAX1}\times(b_2-1)+$ $b_{MAX1}\times b_{MAX2}\times(b_3-1)$	$b_{MAX1}\times b_{MAX2}\times b_{MAX3}$
.		
.		
.		
n $A[b_{L1}:b_{U1},\ldots b_{Ln}:b_{Un}]$	$b_1+b_{MAX1}\times(b_2-1)+\ldots+$ $\ldots\times b_{MAXn-2}\times b_{MAXn-1}\times$ (b_n-1)	$b_{MAX1}\times\ldots\times b_{MAXn}$

b_1, b_2, ..., b_n are subscript expressions.

GENERAL PURPOSE ROUTINES (Continued)

Routine	Coding	Meaning
HBOUND	JSR @HBOUND desc loc_1 loc_2	HBOUND computes the upper bound for the dimension given by loc_1 of the array pointed to by desc. The result is stored at loc_2.
LBOUND	JSR @LBOUND desc loc_1 loc_2	LBOUND computes the lower bound for the dimension given by loc_1 of the array pointed to by desc. The result is stored at loc_2.
SIZE	JSR @SIZE desc loc	SIZE computes the maximum character count is a string (desc) or the element count for an array (desc) and stores the result in loc.
BSARR	JSR @BSARR desc loc_1 loc_2	BSARR builds a based array specifier in allocated storage pointed to by loc_2. desc points to the allocated data area and loc_1 to the based array from which to get the control table dope vector.
BSSTR	JSR @BSSTR desc n loc	BSSTR builds a substring specifier in allocated storage pointed to by loc. desc points to the data area and n is the character count, where: n = current = maximum character count
INDEX	JSR @CALL INDEX 3 $desc_1$ $desc_2$ $desc_3$	INDEX produces an integer index count of a string pointed to by $desc_2$ into a string pointed to by $desc_1$, and stores the result in $desc_3$. An error or non-existent index returns a zero.
SUBSTR	JSR @SUBSTR 3 [4] desc loc_1 [loc_2] loc_3	SUBSTR builds a substring specifier at loc_3 for the string pointed to by desc. The substring's first character is pointed to by loc_1 as an integer count from the start of string (desc) loc_2 points to

GENERAL PURPOSE ROUTINES (Continued)

Routine	Coding	Meaning
SUBSTR (Continued)		the final character; if $\underline{loc}_2$ is not given, it is assumed only one character is needed. Character count is: $C(\underline{loc}_2) - C(\underline{loc}_1) + 1$.
SETCURRENT	JSR @CALL SETCURRENT 2 $\underline{desc}_1$ $\underline{desc}_2$	SETCURRENT sets the current length of the string pointed to by $\underline{desc}_1$ to the integer pointed to by $\underline{desc}_2$. If the integer exceeds the maximum, the current length is set to the maximum length.
MOVSTR	JSR @MOVSTR $\underline{desc}_1$ $\underline{desc}_2$	MOVSTR moves data from the string pointed to by $\underline{desc}_1$ to the string pointed to by $\underline{desc}_2$. Both may be substrings. The smaller character count is moved. If the first character moved to $\underline{desc}_2$ is moved to a position beyond the current length of $\underline{desc}_2$, the initial positions are filled with blanks.
STREQ	JSR @STREQ $\underline{desc}_1$ $\underline{desc}_2$	STREQ compares data values for strings pointed to by $\underline{desc}_1$ and $\underline{desc}_2$ If equal in value and their current lengths are equal, Carry is set.
STRCMP	JSR @STRCMP $\underline{desc}_1$ $\underline{desc}_2$	STRCMP compares the strings pointed to by $\underline{desc}_1$ and $\underline{desc}_2$. If their current lengths are equal, their data values are compared. Indicators are set as follows: $string_1$ > $string_2$ Carry=1 $string_1$ <= $string_2$ Carry=0
ASCII	JSR @ASCII 2 [3] $\underline{desc}$ [$\underline{loc}_1$] $\underline{loc}_2$	ASCII produces the ASCII equivalent (e.g., $A=101_8$) for a character of a string pointed to by $\underline{desc}$. The result is stored in $\underline{loc}_2$. $\underline{loc}_1$ points to an integer index into the string for the character. If $\underline{loc}_1$ is 1 or does not exist, the value returned is for the first character of the string.

GENERAL PURPOSE ROUTINES (Continued)

Routine	Coding	Meaning
LENGTH	JSR @LENGTH desc loc	LENGTH calculates the current length of the string pointed to by desc and stores the result at loc.
MEMORY	JSR @CALL MEMORY 1 desc	MEMORY computes the available memory from the current .SSE (stack end) to the caller's end of stack (indicated by the stack length) minus 200_8 words and stores the result at desc.
ADDRESS	JSR @ADDRESS desc loc	ADDRESS computes the full word address of data pointed to by desc. If it is an array, ADDRESS sets the data address; if it is a string, ADDRESS gets the address of the word containing the first character. The address is stored at loc.
GETADR	JSR @GETADR desc level+sublevel loc	GETADR computes the stack address as does GETSP, using level and sub-level. Then GETADR computes the parameter address with respect to this stack. The address is stored at loc. The parameter is found at desc.
REM	JSR @CALL REM 4 $desc_1$ $desc_2$ $desc_3$ $desc_4$	REM performs an unsigned division on integers found in $desc_1$ and $desc_2$ ($desc_1$/$desc_2$) and stores the result in $desc_3$ and the remainder in $desc_4$.
MOD	JSR @CALL MOD $desc_1$ $desc_2$ $desc_3$	MOD produces an unsigned integer modulo result for C($desc_2$) MOD C($desc_3$) at $desc_1$. C(desc) means the integer found at desc.

GENERAL PURPOSE ROUTINES (Continued

Routine	Coding	Meaning
CVST	JSR @CVST	CVST converts a number pointed to by $\underline{desc_1}$ into a string pointed to by $\underline{desc_2}$. The string will have the format: [-]nnn...n[.nnn...n] [E[-]nn] where: E notation is used as in WRITE standard format. The format and length will be dependent on the type and precision of data.
STCV	JSR @STCV $\underline{desc_1}$ $\underline{desc_2}$	STCV converts a string pointed to by $\underline{desc_1}$ into a number pointed to by $\underline{desc_2}$. The string can have the form: [-]nnn...n[.nnn...n] [E[-]nn] where E notation is used as in WRITE standard format.
SDIV	JSR @SDIV desc $\underline{loc_1}$ $\underline{loc_2}$	SDIV performs a signed division of AC1 by AC2 with the result in AC1. AC0 is assumed 0. AC0 and AC2 are restored on return. Overflow is checked.
SHIFT	JSR @SHIFT desc $\underline{loc_1}$ $\underline{loc_2}$	SHIFT shifts the integer pointed to by desc the number of bits indicated by the counter at $\underline{loc_1}$ (+ = right; - = left) and stores the result in $\underline{loc_2}$. Caution: this is a logical shift.
ROTATE	JSR @ROTATE desc $\underline{loc_1}$ $\underline{loc_2}$	ROTATE rotates the integer pointed to by desc the number of bits specified by the counter at $\underline{loc_1}$. (+ = right; - = left). The result is stored at $\underline{loc_2}$.

GENERAL PURPOSE ROUTINES (Continued)

Routine	Coding	Meaning
RANDOM	JSR @CALL RANDOM 1 desc	RANDOM generates a linear congruent sequence of the form: $X(N+1) := (X(N)*A+C)_{mod2**16}$ producing a (pseudo-) random sequence of integers in the range 0<N<2**16-1, with bit 0 the most significant bit. The number is stored at desc.
SEED	JSR @CALL SEED 1 desc	SEED sets the initial pseudo-ran dom number (X(1)) to the integer found at desc. This is known as seeding the sequence.
UMUL	JSR @CALL UMUL 5 $desc_1$ $desc_2$ $desc_3$ $desc_4$ $desc_5$	UMUL performs an unsigned integer multiply of $desc_1$ and $desc_2$, then adds $desc_3$. The result is stored in $desc_5$ with overflow, if any, in $desc_4$.
EXSBSC	JSR @EXSBSC n + 1 array-desc $subscript\text{-}loc_1$ $subscript\text{-}loc_2$ . . . $subscript\text{-}loc_n$	EXSBSC calculates the address of an element in an n-dimensional globally defined array, checking each sub- script value for legality. Result- ing address is returned in AC2.
SBSCR	JSR @SBSCR array-loc subscript-loc	SBSCR calculates the address of an element in a 1-dimensional array, checking for legality. Resulting address is returned in AC2.

GENERAL PURPOSE ROUTINES (Continued)

Routine	Coding	Meaning
CLASSIFY	JSR @CLASSIFY $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$	CLASSIFY searches for integer in $\underline{desc}_1$ in range table specified by $\underline{desc}_2$ and returns the value of the range in $\underline{desc}_3$.
BYTE	JSR @BYTE $\underline{desc}_1$ [$\underline{desc}_2$] $\underline{desc}_3$	BYTE returns the ASCII equivalent of a character in the buffer pointed to by $\underline{desc}_1$, indexed by $\underline{desc}_2$ (or one) and stored in $\underline{desc}_3$.
TRACE	JSR @CALL TRACE Ø	TRACE traces the current stack and returns.
ONTRACE	JSR @CALL ONTRACE Ø	ONTRACE initiates a trace on a break and sets the break address on return.
OFFTRACE	JSR @CALL OFFTRACE Ø	OFFTRACE removes a trace on a break and break address.

RUN-TIME ERROR ROUTINES

The first run-time error routine allows the user to write messages on the console. The remaining routines are used by ALGOL run-time routines to output messages to the console when an error is encountered.

Run-time error messages may be output in long form or short form. By default, the short form of error messages is loaded, unless one of the I/O routines, read, write, or output is required. If read, write, or output is used, the long form of error messages is loaded.

The short form outputs only an error number via an error return to the system. The long form outputs a message indicating the error, and if the error was fatal, a break to the system is executed, storing the run program as a save file called BREAK.SV (see RDOS manual, #093-000075, for information on save files). If the error is not fatal, the program is resumed.

RUN-TIME ERROR ROUTINES (Continued)

The user can control loading of either the short or long form of error messages by inserting either *external procedure* LONG or *external procedure* SHORT in his source program.

The run-time error responses in both long and short form are:

Short	Long
500	"subscript out of bounds"
501	"stack overflow"
502	"integer overflow"
503	"division by zero"
504	"I/O parity error"
505	"end of file"
506	"illegal file name"
507	"illegal channel number"
510	"exponent over/underflow"
511	"out of disk space"
512	"illegal use of a file"
513	"I/O format error"
514	"illegal parameter"
515	"program not loaded"
516	"dimension error"
517	"floating point error"
520	"square root of negative number"
521	"procedure nesting error"
522	"conversion error"
>522	"unknown error"

The routines used by ALGOL run-time to output messages have the following run-time formatted error code ; where:

NUM = any number from 500_8 to 522_8

ANOP = $@10_8$ for an arithmetic NO-OP

AMES = 1B11 for the start of the number location

FATAL = 1B1 for the fatal message indicator

RUN-TIME ERROR ROUTINES (Continued)

The error has the code:

NUM.*AMES+ANOT[+FATAL]
↑
indicates a decimal number.

Routine	Coding	Meaning
ERROR	JSR @CALL ERROR 1 desc	ERROR outputs an error message given by the string at desc to the console via the system call .PCHAR and returns to the system.
.RTER	JSR @.RTER run-time formatted error code	.RTER outputs the error message as described for ERROR with the location found at .API, a page zero pointer set by ASAV.
.RTEØ	JSR @.RTEØ run-time formatted error code	.RTEØ outputs the error message as described for ERROR with the location found in ACØ.
.ARER	JSR @.ARER run-time formatted error code	.ARER outputs the error message as described for ERROR with the location found in the parameter list pointer of the current stack.

INPUT/OUTPUT RUN-TIME ROUTINES

The I/O run-time routines are described in the ALGOL manual; only a brief description is included here. All the I/O run-time routines have the following coding sequence:

```
JSR  @CALL
routine-name
n
desc1
.
.
.
descn
```

where: n is a count of parameter descriptors and
desc indicates a parameter descriptor.

INPUT/OUTPUT RUN-TIME ROUTINES (Continued)

Routine	Coding	Meaning
READ	JSR @CALL READ $\underline{n}$ $\underline{desc}_1$ . . . $\underline{desc}_n$	READ inputs data of types integer, real, boolean, pointer, and string from a previously opened file whose number is pointed to by $\underline{desc}_1$. On an end-of-file error, transfer is made to an address pointed to by $\underline{desc}_{n-1}$ if $\underline{desc}_{n-1}$ is a label. If $\underline{desc}_{n-1}$ is not a label, end-of-file is considered a normal error. Other errors transfer to $\underline{desc}_n$ if both $\underline{desc}_n$ and $\underline{desc}_{n-1}$ are labels.
WRITE	JSR @CALL WRITE $\underline{n}$ $\underline{desc}_1$ . . . $\underline{desc}_n$	WRITE outputs all arguments as described in READ in an unformatted form to a file indicated by $\underline{desc}_1$. Errors transfer to $\underline{desc}_n$'s address if $\underline{desc}_n$ is a label.
OUTPUT	JSR @CALL OUTPUT $\underline{n}$ $\underline{desc}_1$ $\underline{desc}_2$ . . . $\underline{desc}_n$	OUTPUT outputs in a format given by the string pointed to by $\underline{desc}_2$ to a file pointed to by $\underline{desc}_1$. All those arguments specified by READ errors cause transfer to the location pointed to by $\underline{desc}_n$ if $\underline{desc}_n$ is a label.

INPUT/OUTPUT RUN-TIME ROUTINES (Continued)

Routine	Coding	Meaning
OPEN	JSR @CALL OPEN 2 [3] $\underline{desc}_1$ $\underline{desc}_2$ [$\underline{desc}_3$]	OPEN opens a file pointed to by $\underline{desc}_1$ with the name in the string pointed to by $\underline{desc}_2$. If there is an error, transfer is made to the label at $\underline{desc}_3$. If $\underline{desc}_3$ does not exist, an attempt is made to create the file. If an error again occurs, an error message is output.
CLOSE	JSR @CALL CLOSE 1 desc	CLOSE closes the file pointed to by the file number at desc.
COMARG	JSR @CALL COMARG 2 [3] [4] $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ $\underline{desc}_4$	COMARG reads the command file pointed to by the file number at $\underline{desc}_1$, placing the string in $\underline{desc}_2$. If a boolean array exists at $\underline{desc}_3$, *true* is set for the first 26 elements corresponding to 26 switch letter possibilities. If end-of-file, transfer is made to the label at $\underline{desc}_4$ or a normal return is made if $\underline{desc}_4$ does not exist.
DELETE	JSR @CALL DELETE 1 desc	DELETE deletes the file name found in the string pointed to by desc.
RENAME	JSR @CALL RENAME 2 $\underline{desc}_1$ $\underline{desc}_2$	RENAME renames the file whose name is contained in the string pointed to by $\underline{desc}_1$ to the name in the string at $\underline{desc}_3$.

INPUT/OUTPUT RUN-TIME ROUTINES (Continued)

Routine	Coding	Meaning
POSITION	JSR @CALL POSITION 2 [3] $\underline{desc}_1$ $\underline{desc}_2$ [$\underline{desc}_3$]	POSITION sets the internal byte pointer for reading and writing in a disk file specified by $\underline{desc}_1$, to the byte specified by $\underline{desc}_2$. $\underline{desc}_2$ may be an integer, real, or multi-precision integer whose value is between 0 and 4,294,967,296 bytes. If an error occurs and $\underline{desc}_3$ is present, a transfer is made to the label specified by $\underline{desc}_3$.
FILESIZE	JSR @CALL FILESIZE 2 $\underline{desc}_1$ $\underline{desc}_2$	FILESIZE returns the length in bytes of a disk file specified by $\underline{desc}_1$. The length is returned in the real, integer or multi-precision integer specified by $\underline{desc}_2$.
FILEPOSITION	JSR @CALL FILEPOSITION 2 $\underline{desc}_1$ $\underline{desc}_2$	FILEPOSITION returns the position of the byte currently pointed to in the disk file specified by $\underline{desc}_1$. The position is returned in the real, integer, or multi-precision integer specified by $\underline{desc}_2$.
BYTEREAD	JSR @CALL BYTEREAD 3 [4] [5] $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ [$\underline{desc}_4$]	BYTEREAD inputs data bytes from a previously opened disk file whose number is pointed to by $\underline{desc}_1$, beginning at the byte pointed to by $\underline{desc}_2$ and continuing for the number of bytes given in the count pointed to by $\underline{desc}_3$. $\underline{desc}_4$ may be a label giving a transfer point on occurrence of end-of-file. If not present, end of file is considered a normal transfer.
LINEREAD	JSR @CALL LINEREAD 3 [4] [5] $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ [$\underline{desc}_4$]	LINEREAD inputs a line of data from a previously opened file whose number is pointed to by $\underline{desc}_1$, beginning at the byte pointed to by $\underline{desc}_2$. On return, $\underline{desc}_3$ points to the count of bytes read for the line. $\underline{desc}_4$ may be a label giving

INPUT/OUTPUT RUN-TIME ROUTINES (Continued)

Routine	Coding	Meaning
LINEREAD (continued)		a transfer point on occurrence of end-of-file. If not present, end of file is considered a normal transfer.
BYTEWRITE	JSR @CALL BYTEWRITE 3 [4] $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ [$\underline{desc}_4$]	BYTEWRITE outputs data bytes beginning at the byte pointed to by $\underline{desc}_2$ and terminating when the count of bytes reaches that given by $\underline{desc}_3$ to the file indicated by $\underline{desc}_1$. $\underline{desc}_4$ is an optional error transfer label.
LINEWRITE	JSR @CALL LINEWRITE 3 [4] $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ [$\underline{desc}_4$]	LINEWRITE outputs a line of data to the file given by $\underline{desc}_1$ beginning at the byte pointed to by $\underline{desc}_2$. On return, $\underline{desc}_3$ points to the count of bytes read. $\underline{desc}_4$ is an optional error transfer label.
OVLOD	JSR @CALL OVLOD 1 $\underline{desc}$	OVLOD loads an overlay node using the overlay number contained in $\underline{desc}$.
OVOPN	JSR @CALL OVOPN 2 $\underline{desc}_1$ $\underline{desc}_2$	OVOPN opens an overlay file on the channel in $\underline{desc}_1$ and the filename in $\underline{desc}_2$.
FORMAT	JSR @CALL FORMAT $\underline{n}$ $\underline{desc}_1$ $\underline{desc}_2$ . . . $\underline{desc}_n$	FORMAT outputs arguments $\underline{desc}_3$ through $\underline{desc}_n$ in a format given by $\underline{desc}_2$ to a file pointed to by $\underline{desc}_1$.

INPUT/OUTPUT RUN-TIME ROUTINES (Continued)

Routine	Coding	Meaning
GTIME	JSR @CALL GTIME 6 $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ $\underline{desc}_4$ $\underline{desc}_5$ $\underline{desc}_6$	GTIME returns the date and time in the arguments, as follows: $\underline{desc}_1$ - year $\underline{desc}_2$ - month $\underline{desc}_3$ - day $\underline{desc}_4$ - hour $\underline{desc}_5$ - minute $\underline{desc}_6$ - second
STIME	JSR @CALL STIME 6 $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ $\underline{desc}_4$ $\underline{desc}_5$ $\underline{desc}_6$	STIME sets the date and time to the value in the arguments, as follows: $\underline{desc}_1$ - year $\underline{desc}_2$ - month $\underline{desc}_3$ - day $\underline{desc}_4$ - hour $\underline{desc}_5$ - minute $\underline{desc}_6$ - second
PRINT	JSR @CALL PRINT $\underline{n}$ $\underline{desc}_1$ $\underline{desc}_2$ . . . $\underline{desc}_n$	PRINT outputs all arguments as described in WRITE in an unformatted form to a file indicated by $\underline{desc}_1$. Errors transfer to $\underline{desc}_n$'s address if $\underline{desc}_n$ is a label.
CHAIN	JSR @CALL CHAIN 1 $\underline{desc}$	CHAIN suspends the current program execution and invokes another save file from disk whose name is specified by a string in $\underline{desc}$ (terminated by a null).
APPEND	JSR @CALL APPEND 2 (or 3) $\underline{desc}_1$ $\underline{desc}_2$ [$\underline{desc}_3$]	APPEND appends a file using the number in $\underline{desc}_1$ with the name in the string pointed to by $\underline{desc}_2$. If $\underline{desc}_3$ does not exist, an attempt is made to create the file. If an error occurs again, an error message is output.

INPUT/OUTPUT RUN-TIME ROUTINES (Continued)

Routine	Coding	Meaning
PUTRANDOM	JSR @CALL PUTRANDOM 3 $desc_1$ $desc_2$ $desc_3$	PUTRANDOM writes a record to the file opened on channel $desc_1$ starting at the record number in $desc_2$, using the data pointed to by $desc_3$.
GETRANDOM	JSR @CALL GETRANDOM 3 $desc_1$ $desc_2$ $desc_3$	GETRANDOM reads in a record from the file opened on channel $desc_1$ from the record number in $desc_2$ into the data area pointed to by $desc_3$.

SUBROUTINES USED BY RUN-TIME ROUTINES

Each of the subroutines represents coding required for more than one run-time routine. The parameters are all passed either in the accumulators (AC0 through AC3) or on the temporary stack (.RP+n where n is an offset to the stack pointer).

Routine	Coding	Meaning
ADDRS	JSR @ADDRS	ADDRS computes the address of the present stack frame of a variable. If bit 0 is set, the pointer is to an offset to the stack; if bit 0 is not set, the pointer is to an absolute address. AC1 contains the pointer and the result is returned in AC1. Other accumulators are preserved with AC3 set to .FP.
OADDR	JSR @OADDR	OADDR computes the address of a variable exactly as ADDRS except that the address is for the previous stack frame.
SUSET	JSR @SUSET	SUSET evaluates a substring for string manipulation routines. AC1 contains a string pointer; AC2 contains a substring pointer; and Carry is set for three-word strings.

SUBROUTINES USED BY RUN-TIME ROUTINES (Continued)

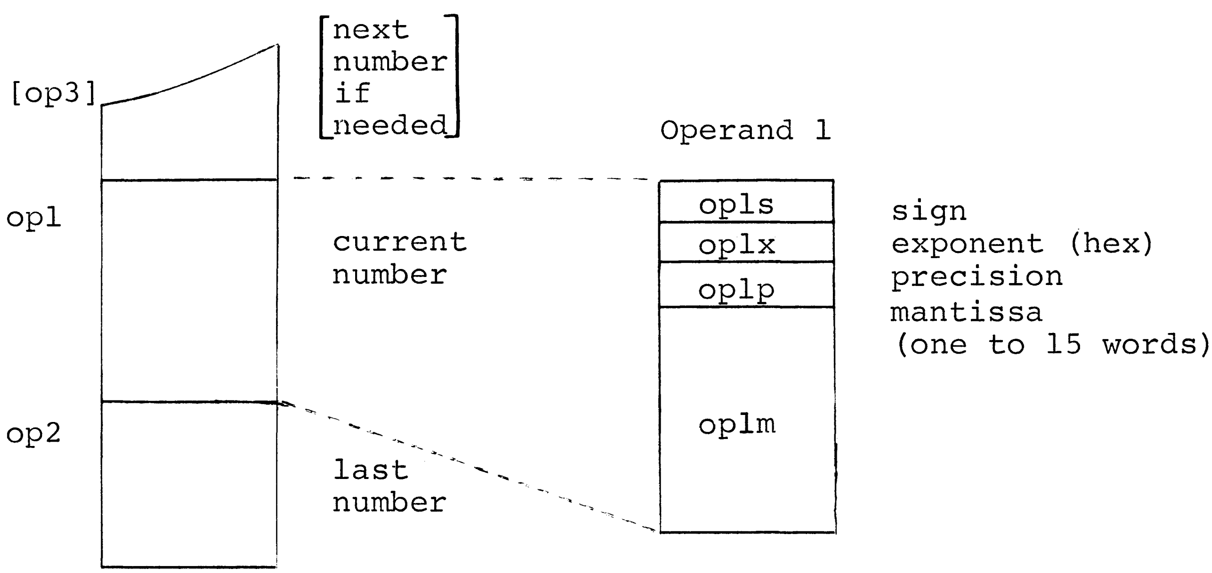
Routine	Coding	Meaning
SUSET (continued)		On return, AC1 contains a byte pointer to the first character and AC0 contains the byte count. If the substring begins beyond the current length, 0 is returned in AC0 for a null string.
SUNSET	JSR @SUNSET	SUNSET evaluates a substring for string manipulation routines. AC1 contains a substring pointer and Carry is set for three word strings. On return AC1 contains a byte pointer to the first character and AC0 contains the byte count. .RP+6 contains an end-of-data byte count (byte pointer+current length).
WRITA	JSR @WRITA	WRITA allocates a data area for an array (local) and writes the control table (via CONTR). WRITA requires: .RP+0 array dimension count+1 .RP+2 array specifier pointer .RP+3 array control table address .RP+7 temporary .RP+11 array word count (set) .RP+13 temporary
CONTR	JSR @CONTR	CONTR writes the control table (dope) for an array. It requires: .RP+0 array dimension count+1 .RP+3 array control table address .RP+13 temporary
DIMMU	JSR @DIMMU	DIMMU multiplies the dimensions of an array, checking for legal array bounds on the dimensions. It requires: AC1 array control table address .RP+0 array dimension count +1 .RP+4 temporary .RP+12 temporary .RP+13 temporary .RP+14 temporary

SUBROUTINES USED BY RUN-TIME ROUTINES (Continued)

Routine	Coding	Meaning
ASTR	JSR @ASTR	ASTR allocates a buffer of 240_8 words to the end of the current stack frame, returning a word pointer in AC0. It requires: .RP+1 temporary .RP+2 temporary
MPY	JSR @MPY	MPY is unsigned multiply of AC1* AC2 with the result in AC1, overflow in AC0. AC0 is assumed 0 to start.
DVD	JSR @DVD	DVD is unsigned divide of AC1 by AC2 with the result in AC1, remainder in AC0. AC0 is assumed 0 to start.

NUMBER ROUTINES

The run-time number routines are the routines required to do the arithmetic for ALGOL run-time, including conversion, functions, and stack manipulation. Each routine uses a number stack, allocated by the initialization routine previously described. Following is a diagram of the stack and the representation of the number.



NUMBER ROUTINES (Continued)

All the operands have the same format. Allocation is always for maximum (15 words+3) size.

The number routines are described with the notation:

op1	-	operand one - current number
op2	-	operand two - last number
op*n*s	-	operand *n* sign
op*n*x	-	operand *n* exponent (hex)
op*n*p	-	operand *n* mantissa

where: *n* is replaced by 1, 2, or 3.
.sp+*n* is the current stack variable area plus 1.

Routine	Coding	Meaning
IPTNR	JSR @CALL IPTNR 5 $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ $\underline{desc}_4$ $\underline{desc}_5$	IPTNR converts a character string into a number which is either integer, real, or pointer. It requires: $\underline{desc}_1$ - byte pointer to string $\underline{desc}_2$ - result number $\underline{desc}_3$ - number type $\underline{desc}_4$ - number precision $\underline{desc}_5$ - number radix
OPTNR	JSR @CALL OPTNR 5 $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ $\underline{desc}_4$ $\underline{desc}_5$	OPTNR converts a number to an unformatted, simplified string of either form: [-]nn...nnn[.nn...nn] or [-].nn...nnE[-]nn It requires: $\underline{desc}_1$ - string byte pointer $\underline{desc}_2$ - number address $\underline{desc}_3$ - number type $\underline{desc}_4$ - number precision $\underline{desc}_5$ - number radix

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
NROPT	JSR @CALL NROPT 11 $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ $\underline{desc}_4$ $\underline{desc}_5$ $\underline{desc}_6$ $\underline{desc}_7$ $\underline{desc}_{10}$ $\underline{desc}_{11}$	NROPT converts a number to a formatted, specified string similar to the unformatted form, but specifying which type of format will be used. It requires: $\underline{desc}_1$ - string byte pointer $\underline{desc}_2$ - number address $\underline{desc}_3$ - number type $\underline{desc}_4$ - number precision $\underline{desc}_5$ - number radix $\underline{desc}_6$ - sign indicator $\underline{desc}_7$ - integer field width $\underline{desc}_{10}$- fraction field width $\underline{desc}_{11}$- exponent field width
ASCNU	JSR @ASCNU	ASCNU converts a character string to a number. It requires: AC0 - string byte pointer (terminated by null) AC2 - radix of number The number goes to OP1, which is assumed created.
NUMASC	JSR @NUMASC	NUMASC converts a number found in OP1 to an ASCII character string. It requires: AC0 - string byte pointer AC2 - number radix NUMASC returns the digit count in AC0.
IOUT	JSR @IOUT	IOUT sets the proper sign and moves the integer part of a number character string from <u>string pointer</u> to <u>output pointer</u>. It requires:

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
IOUT (continued)		AC0 digits of precision AC1 field width AC2 sign indicator .SP+ 1 string byte pointer .SP+12 output byte pointer
FOUT	JSR @FOUT	FOUT moves the fractional part of a number character string from string pointer to output pointer. It requires: AC0 digits of precision AC1 field width .SP+ 1 string byte pointer .SP+12 output byte pointer
GETBT	JSR @GETBT	GETBT gets a character from the temporary stack (.RP) minus a two-byte pointer (i.e., .RP-2→ byte pointer).
PUTBT	JSR @PUTBT	PUTBT puts a character to the byte pointer found at .RP-1.
PUSH	JSR @PUSH desc	PUSH pushes a number to the number stack. desc points to one of the following type numbers: integer or real.
POP	JSR @POP desc	POP pops a number from the number stack to desc, pointing to one of the following type numbers: integer or real.
POWER	JSR @POWER $desc_1$ $desc_2$ $desc_3$	POWER raises a base ($desc_1$) to a power ($desc_2$) for both integer and real bases and powers and stores the result as $desc_3$ which must be real.
SQR	JSR @SQR	OP1 = SQRT(OP1) The algorithm uses constants A and B to obtain an initial approximation and iterates:

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
SQR (continued)		approx. = (number/approx.+approx.)/2
SIN COS	JSR @SIN JSR @COS	OP1 = SIN(OP1) OP1 COS(OP1) Algorithm: COS(arg) = SIN(arg+PI/2) Set A = arg*2/PI Break into integer(I) and real (R) parts. Q = 0 or 1 for SINE and COSINE QARG = SIGN(R)+SIGN(R)*Q+I If QARG is odd, set R=1.-R SIN(arg/3) = S = P(R**2)*R SIN(arg) = (3.-4.*S**2)*S
ATN	JSR @ATN	OP1 = ARCTAN(OP1) Algorithm: Set X = ABS(arg) or 1/ABS(arg) if ABS(arg) greater than or equal 1.0 Set Y = X - TAN(PI/12) Set R = X or if Y greater than or equal 0 =(X*SQRT(3)-1)/(SQRT(3)+X) Evaluate P(R**2), Q(R**2) Evaluate P*R/Q and add PI/6 if Y>= 0 Subtract PI/2 if ABS(arg)>=0 Set SIGN as that or original arg.

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
TAN	JSR @TAN	OP1 = TAN(OP1) Algorithm: arg = ABS(arg)*4/PI Break arg into integer (I) and real (R) parts. If I is odd, then R = 1.-R SIGN(R) = MOD2(SIGN(orig arg)+I/2) Set COT switch if MOD4(I) = 2,3 Evaluate R(R**2), Q(R**2) TAN(arg) = Q/P*R/8 if COT switch is set.
ALG	JSR @ALG	OP1 = LN(OP1) Algorithm: LN of exponent obtained as: LN(2)*exponent radix(2) LN of mantissa evaluated in terms of iteration: R = (arg.-0.5 [or 1.0]/(arg.+0.5 [or 1.0] according to arg <= or > 1/SQRT(2) Evaluate P(R**2, Q(R**2) Set LN(mantissa) = P*R/Q
EXP	JSR @EXP	OP1 = EXP(OP1) Algorithm: Compute X = arg*LOG2(E) Break X into integer (IX) and real (RX) parts.

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
EXP (continued)		Set R = RX-.5 (for RX>=0) = RX+.5 (for RX <0) I = IX (for RX>=0) = -(IX+1) (for RX <0) Get P(R**2), Q(R**2) Set mantissa as (Q-P*R)/(Q+P*R) or its reciprocal if RX<0. Halve as many times as (I)MOD4. Add (I) radix 4 to exponent and multiply by SQRT(2).
ABS	JSR @ABS	OP1 = ABS(OP1) Set the sign to positive. Set AC0: = +1 for OP1>0 0 for OP1=0 -1 for OP1<0 OP1 is unchanged.
VPRC	JSR @VPRC	Determine the highest order coefficient required by the function P(X) or Q(X), depending on the exponent of the argument and the exponents of the coefficients. AC0 - high order coefficients AC1 - low order coefficients OP1 - argument
PLY	JSR @PLY	Given the coefficients, evaluate P(X) to the form: (...(C(N)*arg+C(N-1))*arg+ C(N-2))...)*arg+C(0)) Result is in OP1. AC0 - high order coefficients AC1 - low order coefficients OP1 - argument

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
CMOVE	JSR @CMOVE	CMOVE pushes the constant pointed to by AC0 to the new OP1, setting the precision by OP2 (old OP1).
BREAK	JSR @BREAK	BREAK breaks the current number at OP1 into integer and real parts with the integer in AC1 and the real at OP1.
FLIP	JSR @FLIP	FLIP swaps the top two operands. OP1 becomes OP2 and OP2 becomes OP1.
FHALF	JSR @FHALF	OP1 = OP1/2 Shift the mantissa of OP1 right one bit, forcing a zero from the left, thus dividing by two.
MADD	JSR @MADD	OP1 = OP1+OP2 For multi-precision integers. The result is one word greater than the larger precision number.
MSUB	JSR @MSUB	OP1 = OP2-OP1 For multi-precision integers. The result is one word greater than the larger precision number.
MMPY	JSR @MMPY	OP1 = OP1*OP2 For multi-precision integers. The result has a precision equal to the sum of the two precisions or else the maximum precision.
MDVD	JSR @MDVD	OP1 = OP2/OP1 For multi-precision integers. The result has a precision equal to the sum of the two precisions or else the maximum precision.

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
MAND	JSR @MAND	OP1 = OP1 *and* OP2 For multi-precision integers. The resulting precision is that of the larger with excess words zeroed from the top. The rest are full word ands of the original operands.
MOR	JSR @MOR	OP1 = OP1 *or* OP2 For multi-precision integers. The resulting precision is that of the larger with excess words of the larger untouched from the top down. The remaining words are full word operations of the form: complement OP1,OP1 *and* OP1,OP2 add complement OP1,OP2
MNOT	JSR @MNOT	OP1 = *not* (OP1) The result is a full word complement of the multi-precision integer at operand 1.
FML	JSR @FML	OP1 = OP1*OP2 For multi-precision real data. Both numbers are assumed normzlized and the result has the same precision as the larger number.
FDV	JSR @FDV	OP1 = OP2/OP1 For multi-precision real data where neither real is considered normalized. The result has the same precision as the larger number.

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
FMDC	JSR @FMDC	FMDC determines the resulting characteristics for a multiplication or a division (Carry set for divide), and sets operand 3 with the following data: OP3S, OP3X, OP3P.
MMUL	JSR @MMUL	MMUL multiplies operand 1 by the single (unsigned) word in AC2. AC0 contains the addend for the first multiply. Overflow is returned in AC0.
MDIV	JSR @MDIV	MDIV divides operand 1 by a full word (unsigned) integer in AC2. The remainder is returned in AC1.
ENTIER	JSR @ENTIER $\underline{desc}_1$ $\underline{desc}_2$	ENTIER converts the real number pointed to by $\underline{desc}_1$ to an integer pointed to by $\underline{desc}_2$ (one to 15 words depending upon $\underline{desc}_2$.)
FCMP	JSR @FCMP	FCMP compares operand 1 to operand 2 and sets Carry = 0 for OP1>OP2 1 for OP1<=OP2
FEQC	JSR @FEQC	FEQC compares operand 1 and operand 2 for equality, and sets Carry if OP1 = OP2.
UPAC	JSR @UPAK	UPAK unpacks a multi-precision real number to form a new operand 1. AC0 = memory address AC2 = memory precision
XUPK	JSR @XUPK	XUPK unpacks a multi-precision integer number to form a new operand 1. AC0 = memory address AC2 = memory precision

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
PACK	JSR @PACK	Pack packs operand 1 to a multi-precision real number and deletes OP1. AC1 = memory address AC2 = memory precision
XPACK	JSR @XPAK	XPACK packs operand one to a multi-precision integer number and deletes OP1. AC1 = memory address AC2 = memory precision
FAD	JSR @FAD	OP1 = OP1+OP2 For real, multi-precision, normalized numbers. The result has the same precision as the larger number.
FSB	JSR @FSB	OP1 = OP2-OP1 For real, multi-precision normalized numbers. The result has the same precision as the larger number.
MAD	JSR @MAD	MAD adds the mantissas of two multi-precision numbers with the same precision.
MNEG	JSR @MNEG	MNEG negates the mantissa of an operand indicated by AC2 = -1 for OP2 AC2 = 0 for OP1
RONDH	JSR @RONDH	RONDH rounds a number in operand 1 to a specified hexidecimal digit indicated by AC1 and numbered from 0.
XUNM	JSR @XUNM	XUNM unnormalizes a multi-precision integer in operand 1 to a desired precision contained in AC1.

NUMBER ROUTINES (Continued)

Routine	Coding	Meaning
XFL	JSR @XFL	XFL converts a multi-precision integer in operand 1 to a multi-precision real number in operand 1 (with the smallest precision possible).
FLX	JSR @FLX	FLX converts a multi-precision real number in operand one to a multi-precision integer in operand one (with the smallest precision possible).
FLF	JSR @FLF	FLF converts operand 1 to a single precision integer in AC0; operand 1 is not destroyed. Check is made for overflow.
FXF	JSR @FXF	FXF converts a single-precision integer in AC0 to a normalized real number in operand 1. Operand 1 is assumed created.
CKOU	JSR @CKOU	CKOU checks operand 2 for overflow or underflow: overflow - exponent>=200_8 underflow - negative exponent (i.e., the exponent must have the form $0 \leq E < 200_8$ to pass the over/underflow check.
CKOU1	JSR @CKOU1	CKOU1 checks operand 1 for over/underflow as described for CKOU.
FSN SIGN	JSR @SIGN	FSN and SIGN indicate the sign of a multi-precision number and set AC0 as follows: AC0 = +1 if OP1>0 AC0 = - if OP1=0 AC0 = -1 if OP1<0

Routine	Coding	Meaning
FNOR	JSR @FNOR	FNOR normalizes the operand indicated by AC2(=0 for OP1; = -1 for OP2) to a hexadecimal digit (i.e., the top four bits), decrementing the exponent for every shift left. AC1 contains the extended carry* word to be shifted in from the left.
ROND	JSR @ROND	ROND rounds the register whose address is in AC0 (address of unpacked register or operand) if the extended carry word in AC1 has bit 0 set.
ROMDM	JSR @ROMDM	ROMDM rounds the register whose address is in AC0, as in ROND, to a precision found in AC2 (memory precision).
RST	JSR @RST	RST shifts an operand indicated by AC2(-1 for OP2, 0 for OP1) one hex digit to the right, shifts the extended carry in AC1 into the operand from the right and shifts out a new carry to AC1.
LST	JSR @LST	LST shifts an operand to the left one hex digit following the same method as described for RST, except for the left shift. AC2 - operand indicator AC1 - extended carry
IOVFL	JSR @IOVFL	IOVFL outputs an integer overflow error message and sets the operand indicator by AC2 (-1 for operand 2, 0 for operand 1) to a maximum integer with desired precision in AC1.
MOVE	JSR @MOVE	MOVE moves the operand from address in AC0 to address in AC1. Both are (and will be) unpacked operands whose data addresses are in AC0, AC1.

*Note: Carry = hardware register; carry = logical carry

COPY	JSR @COPY	COPY copies an operand whose address is in AC0 to the top of stack (i.e., it becomes the newest pushed number to stack). OP1 will be created and the number must be unpacked.

Floating Point Interpreter

ALGOL uses an interpreter to perform floating point and multi-precision integer arithmetic. The calling sequence is:

```
FENTL      (.EXTN)    (real)      XENTL          (integer)
  .                                 .
  .                                 .
  .                                 .
<instruction set>                 <instruction set>
  .                                 .
  .                                 .
  .                                 .
FEXT                              FEXT
```

with the following instruction set:

```
FPRC    n
FLDA    adr
FSTA    adr
FNEG    r,r
FMOV    r,r
FPOS    r,r
FSUB    r,r
FADD    r,r
FXFL    r,r            <to floating register>
FLFX    r,r            <to real register>
FSGH    r,r
FSEQ    r,r
FIPT    r,r            <.EXTN IPT>
FLDA    r,adr
FSTA    r,adr
FMUL    r,r            <.EXTN FM>    <.EXTN XM>
FDIV    r,r            <.EXTN FD>    <.EXTN XD>
FALG    r,r            <.EXTD ALG>
```

Floating Point Interpreter (Continued)

FATN	r,r	<.EXTD ATN>
FCOS	r,r	<.EXTD COS>
FSIN	r,r	<.EXTD SIN>
FTAN	r,r	<.EXTD TAN>
FEXP	r,r	<.EXTD EXP>
FSQR	r,r	<.EXTD SQR>

where: n is an integer from 1 to 15
r is a pseudo-register
adr is an absolute, indexed, or indirect displacement
<.EXTN name> or <.EXTN name> are external definitions for the routines required.

CACHE MEMORY MANAGEMENT ROUTINES

The following run-time routines are internal ALGOL routines used to create and maintain Cache Memory, which provides for automatic transfer between disk and core of blocks of data. They provide an alternative to the usual I/O routines and should be used when very large files are being transferred. See the description of Cache Memory in Chapter 9.

Routine	Coding	Meaning
BUFFER	JSR @CALL BUFFER 2 $desc_1$ $desc_2$	BUFFER allocates buffer of $desc_2$ words in length with $desc_1$ pointing to the first word of the buffer.
ACCESS	JSR @CALL ACCESS 3 [4] $desc_1$ $desc_2$ $desc_3$ $desc_4$	ACCESS opens a file named $desc_2$, with file number in $desc_1$ and element size $desc_4$. $desc_3$ must be the name of the buffer pointer.

CACHE MEMORY MANAGEMENT ROUTINES (Continued)

Routine	Coding	Meaning
WORDREAD	JSR @CALL WORDREAD 3 [4] $desc_1$ $desc_2$ $desc_3$ [$desc_4$]	WORDREAD reads a block from the file whose file number is in $desc_1$, beginning at file address $desc_2$, into the core area pointed to by $desc_3$. Only the number of words given in $desc_4$ are read.
WORDWRITE	JSR @CALL WORDWRITE 3 [4] $desc_1$ $desc_2$ $desc_3$ [$desc_4$]	WORDWRITE writes a block into the file whose file number is in $desc_1$, beginning at file address $desc_2$, from the area pointed to by $desc_3$. Only the number of words given in $desc_4$ are written.
FETCH	JSR @CALL FETCH 1 [2] $desc_1$ [$desc_2$]	FETCH returns the single word at file address $desc_1$ with the offset $desc_2$.
STASH	JSR @CALL STASH 2 [3] $desc_1$ $desc_2$ [$desc_3$]	STASH writes the word in $desc_1$ onto disk at file address $desc_2$ with offset $desc_3$.
NODESIZE	JSR @CALL NODESIZE 1 $desc_1$	NODESIZE returns the number of words in a node, pointed to by $desc_1$.
NODEREAD	JSR @CALL NODEREAD 2 $desc_1$ $desc_2$	NODEREAD reads a node from the file address in $desc_1$ into the core area pointed to by $desc_2$.
NODEWRITE	JSR @CALL NODEWRITE 2 $desc_1$ $desc_2$	NODEWRITE writes the node pointed to by $desc_2$ onto disk beginning at the file address in $desc_1$.

CACHE MEMORY MANAGEMENT ROUTINES (Continued)

Routine	Coding	Meaning
HASHREAD	JSR @CALL HASHREAD 4 $\underline{desc}_1$ $\underline{desc}_2$ $\underline{desc}_3$ $\underline{desc}_4$	HASHREAD returns the core address in $\underline{desc}_3$ of the block referenced in $\underline{desc}_2$ in the file opened on $\underline{desc}_1$ and returns the offset into the block in $\underline{desc}_4$.
HASHWRITE	JSR @CALL HASHWRITE 1 $\underline{desc}_1$	HASHWRITE marks the last block referenced by HASHREAD with buffer pointer $\underline{desc}_1$ as being modified.
FLUSH	JSR @CALL FLUSH 1 $\underline{desc}_1$	FLUSH clears the buffer area pointed to by $\underline{desc}_1$.

Four other run-time routines are used by the CMM routines, but are inaccessible to the user:

Routine	Function
.CAER	Performs CMM error reporting.
.CAPO	Returns the position of data within the CMM buffer.
.CARD	Reads data into the CMM buffer.
.CAWR	Writes data from the CMM buffer onto disk.

SUBROUTINES REFERENCED BY RUN-TIME ROUTINES

The following table lists the run-time routines alphabetically, together with the names of subroutines called by them. Page zero variables (e.g., .FP, .SP, etc.) are excluded and are all contained on a single program tape, entitled ZERO.

Routine		Subroutines Called
ABRETN	†	ADDRS, OADDRS, .ARER
ABS		--
ACCESS	*	SAVE, RETURN, .RTER
ADDRESS		ARET, ASAV
ADDRS		--
ALG		CMOVE,COPY,FAD,FDV,FLIP,FML,FNOR,FSB,PLY,.RTER
ALLOCATE	†	RETURN,SAVE,.ARER
APPEND	*	SAVE,RETURN,SUBSTR,ABRET,MOVSTR,SALLOC,.RTER,RTEØ
ARET	†	--
ARRAY	†	ADDRS,ARET,ASAV,CONTR,WRITA,.ARER
ASAV	†	--
ASCII		ADDRS,ARET,ASAV,.ARER
ASCNU		FLF,FNOR,GETBT,MDIV,MMUL,RONDH,RST,.RTER
ASTR	†	.ARER
ATN		CMOVE,COPY,FAD,FDV,FLIP,FML,FSB,PLY
BLKEND	†	ARET,ASAV,.ARER
BLKSTART	†	ARET,ASAV,.ARER
BREAK		FLF,FSB,FXF
BSARR		ARET,ASAV
BSSTR		ARET,ASAV
BUFFER	†	.RTEØ,CALL,DVD,SAVE,ALLOCATE,RETURN,.RTER
BYTE		ARET,ADDRS,ASAV
BYTEREAD	*	ABRETN,RETURN,SAVE
BYTEWRITE	*	ABRETN,RETURN,SAVE
CALL	†	--
CHAIN	*	SAVE
CKOU		FNOR,.RTER
CKOU1		FNOR,.RTER
CLASSIFY		ARET,ADDRS,ASAV
CLOSE	*	RETURN,SAVE,.RTER
CMOVE		--
COMARG	*	ABRETN,ASTR,MOVSTR,OADDR,RETURN,SAVE,.RTER
CONTR	†	ADDRS, DIMMU, .ARER, MPY
COPY		--

* Routines that call the operating system.
† Routines that modify stack or *own* pointers.

SUBROUTINES REFERENCED BY RUN-TIME ROUTINES (Continued)

Routine		Subroutines Called
COS		BREAK,CMOVE,FAD,FLIP,FML,FSB,MOVE,PLY,VPRC
CVST		ASTR,CALL,OADDR,RET,RSAV,.ARER,INDEX,OPTNR
DELETE	*	LENGTH,MOVSTR,RETURN,SALLOC,SAVE,SUBSTR,.RTER
DIMMU		.ARER,MPY
DVD		--
ENTIER		ADDRS,ARET,ASAV,FLF,FLX,UPAK,XPAK
ERROR	*	SAVE,.ARER
EXP		BREAK,CMOVE,COPY,FAD,FDV,FHALF,FLIP,FML,FSB, PLY,VPRC,.RTER
EXSBS		.ARER,MPY
FAD		CKOU,MAD,MNEG,MOVE,RST
FCMP		RSB
FDV		CKOU,FMDC,FNOR,MAD,MOVE,ROND,RST,DVD,MPY
FENT		RINTR
FEQC		FSB
FETCH		.CAER,.CARD
FHALF		CKOU1,FNOR
FILEPOSITION	*	OADDR,PACK,RETURN,SAVE,XFL,XPAK
FILESIZE	*	FLF,OADDR,PACK,RETURN,SAVE,XFL,XPAK
FLF		RONDM
FLIP		--
FLUSH	*	.RTEØ,CALL,DVD,SAVE,ALLOCATE,RETURN,.RTER
FLX		FNOR,IOVFL,RST
FMDC		--
FML		CKOU,FMDC,FNOR,MAD,MOVE,ROND,RST,DVD,MPY
FNOR		LST,ROND
FORMAT	*	SAVE,OADDR,RETURN,.RTER
FOUT		GETBT,PUTBT
FREE	†	RETURN,SAVE,.ARER
FSB		CKOU,MAD,MNEG,MOVE,RST
FSN		--
FXF		FNOR
GETADR		ADDRS,ARET,ASAV,.ARER
GETBT		--
GETRANDOM	*	SAVE,RETURN,.RTER
GETSP		.ARER,ARET,ASAV
GTIME	*	SAVE,RETURN,.ARER
HASHREAD		SAVE,RETURN,.CAPO

SUBROUTINES REFERENCED BY RUN-TIME ROUTINES (Continued)

Routine		Subroutines Called
HASHWRITE		SAVE,RETURN,.CAPO
HBOUND		ADDRS,ARET,ASAV
IINTR		FDV,FML,FLX,FLF,FEQS,SQR,ARET,TAN,MMPY,COPY, SIN,MOVE,MADD,FHALF,COS,MDVD,RSAV,XPAK,FUPK, PACK,UPAK,FNOR,ATN,FXF,ASAV,ALG,MSUB,FSB, RRET,FAD,EXP,SIGN,.RTEØ
IOUT		GETBT,PUTBT
IOVFL		.RTER
IPTNR		ASCNU,PACK,RETURN,SAVE,XPAK
LBOUND		ADDRS,ARET,ASAV,.ARER
LENGTH		ADDRS,ARET,ASAV
LINEREAD	*	ABRETN,RETURN,SAVE
LINEWRITE	*	ABRETN,RETURN,SAVE
LST		--
MAD		--
MADD		FAD,FLX,FNOR
MAND		FLIP,FLX
MDIV		DVD
MDVD		FDV,FLIP,FLX,FNOR
MEMORY		RETURN,SAVE
MMPY		FLIP,FLX,FML,FNOR
MMUL		MPY
MNEG		--
MNOT		FLX
MOD		RETURN,SAVE,DVD,MPY
MOR		FLIP,FLX
MOVE		--
MOVSTR		ADDRS,ARET,ASAV,SUNSET,SUSET
MPY		--
MSUB		FLX,FNOR,FSB
NODEREAD		.CAER,.CARD
NODESIZE		.CAER,.CARD
NODEWRITE		.CAER,.CAWR
NROPY		FOUT,FXF,IOUT,NUMASC,PUTBT,RETURN,SAVE,UPAK, XFL,XUPK
NUMASC		FNOR,GETBT,MDIV,MMUL,PUTBT,RONDH,RST,.RTER,DVD
OADDR		--
OFFTRACE		SAVE,RETURN
ONTRACE		SAVE,RETURN

SUBROUTINES REFERENCED BY RUN-TIME ROUTINES (Continued)

Routine		Subroutines Called
OPEN		ABRETN,LENGTH,MOVSTR,RETURN,SALLOC,SAVE,SUBSTR, .RTER
OPTNR		FOUT,FXF,IOUT,NUMASC,PUTBT,RETURN,SAVE,UPAK, XFL,XUPK
OUTPUT	*	ABRETN,ASTR,CALL,OADDR,RETURN,SAVE,.RTER,MPY, NROPT
OVLOD	*	SAVE,RETURN
OVOPN	*	SAVE,RETURN
PACK		RONDM,RST
PLY		CMOVE,FAD,FML,MOVE
POP		ARET,ASAV,FLF,PACK,XPAK
POSITION	*	ABRETN,OADDR,RETURN,SAVE,UPAK,XUNM,XUPK
POWER		ADDRS,ALG,ARET,ASAV,CMOVE,COPY,EXP,FDV,FLIP, FML,FXF,PACK,SIGN,UPAK,XFL,XUPK
PRINT	*	CALL,SAVE,OADDR,RETURN,ABRET,ASTR,.RTEØ,MPY, OPTNR
PUSH		ARET,ASAV,FXF,UPAK,XUPK
PUTBT		--
PUTRANDOM	*	SAVE,RETURN,.RTER
RANDOM		RETURN,SAVE
READ	*	ABRETN,ASTR,CALL,OADDR,RETURN,SALLOC,SAVE, .RTEØ, .RTER,IPTNR,MPY
REM		RETURN,SAVE,DVD
RENAME	*	LENGTH,MOVSTR,RETURN,SALLOC,SAVE,SUBSTR,.RTER
RETURN	†	ADDRS,OADDRS,.ARER
RINTR		FDV,FML,FLX,FLF,FEQS,SQR,ARET,TAN,MMPY,COPY,SIN, MOVE,MADD,FHALF,COS,MDVD,RSAV,XPAK,FUPK,PACK, UPAK,FNOR,ATN,FXF,ASAV,ALG,MSUB,FSB,RRET,FAD, EXP,SIGN,.RTEØ
ROND		RST
RONDH		FNOR,RST
RONDM		RST
ROTATE		ADDRS,ARET,ASAV
RRET	†	.RTER
RSAV	†	.RTER
RST		--
SALLOC		ADDRS,ARET,ASAV,STCOM
SARRAY		ADDRS,ARET,ASAV,DIMMU,STCOM
SAVE	†	ADDRS,ASAV,OADDR,.RTER
SBSCR		.ARER,MPY
SDIV		ARET,ASAV,.ARER,DVD
SEED		RETURN,SAVE

SUBROUTINES REFERENCED BY RUN-TIME ROUTINES (Continued)

Routine		Subroutines Called
SETCURRENT		RETURN,SAVE
SFREE	†	RETURN,SAVE,.ARER
SHIFT		ADDRS,ARET,ASAV
SIGN		--
SIN		BREAK,CMOVE,FAD,FLIP,FML,FSB,MOVE,PLY,VPRC
SIZE		ADDRS,ARET,ASAV,DIMMU,.ARER
SQR		CMOVE,FAD,FDV,FHALF,FML,FNOR,MOVE,.RTER
STACH		.CAER,.CAWR
STCOM	†	.ARER
STCV		ASTR,CALL,OADDR,RRET,RSAV,.ARER,IPTNR
STIME	*	SAVE,RETURN,.ARER
STRCMP		ADDRS,ARET,ASAV,SUSET
STREQ		ADDRS,ARET,ASAV,SUSET
SUBSCRIPT		.ARER,MPY
SUBSTR		ADDRS,ARET,ASAV,.ARER
SUNSET		--
SUSET		--
TAN		BREAK,CMOVE,COPY,FDV,FLIP,FML,FSB,PLY,VPRC
TRACE	*	SAVE,RETURN
UMUL		SAVE,RETURN,MPY
UPAK		LST
VPRC		--
WORDREAD		.CAER,.CARD
WORDWRITE		.CAER,.CAWR
WRITA	†	ADDRS,DIMMU,.ARER,MPY
WRITE	*	ABRETN,ASTR,CALL,OADDR,RETURN,SAVE,.RTEØ,MPY, OPTNR
XENTL		PACK,UPAK,XPAK,XUPK,MSUB,MADD,IINTR
XFL		FNOR,IOVFL,RST
XPAK		IOVFL,MNEG,XUMN
XUNM		FLX,IOVFL,RST
XUPK		FLX,MNEG
.ARER	*	--
.CAER		.RTEØ,CALL,DVD,SAVE,ALLOCATE,RETURN,.RTER
.CAPO		.RTEØ,CALL,DVD,SAVE,ALLOCATE,RETURN,.RTER
.CARD		.RTEØ,CALL,DVD,SAVE,ALLOCATE,RETURN,.RTER
.CAWR		.RTEØ,CALL,DVD,SAVE,ALLOCATE,RETURN,.RTER
.RTEØ	*	--
.RTER	*	--
.SPINIT	*†	.ARER,MAIN (ALGOL source routine)

ROUTINES THAT USE SYSTEM CALLS

The following table lists system routines alphabetically, together with the names of run-time routines that call them.

System Call	Routines that Use the System Call
.APPEND	APPEND
.BREAK	.ARER (long error), TRACE
.CHSTS	FILESIZE
.CLOSE	CLOSE
.CRAND	APPEND, OPEN, ACCESS
.DEBL	.SPINIT
.DELETE	DELETE
.EOPEN	ACCESS
.ERTN	.RTER (short error), TRACE
.EXEC	CHAIN,TRACE
.GDAY	GTIME
.GPOS	FILEPOSITION
.GTOD	GTIME
.MEM	.SPINIT
.MEMI	.SPINIT
.OPEN	OPEN
.OVLOD	OVLOD
.OVOPN	OVOPN
.PCHAR	ERROR, .ARER (long error)
.RDB	.CARD, .CAWR, .CAPO
.RDL	COMARG, LINEREAD,READ
.RDR	GETRANDOM
.RDS	COMARG, BYTEREAD
.RENAME	RENAME
.RESET	.SPINIT
.RTN	.SPINIT, .ARER (long error)
.SDAY	STIME
.SPOS	POSITION
.STOD	STIME
.SYSI	.SPINIT
.WRB	FLUSH, .CARD, .CAWR, .CAPO
.WRL	LINEWRITE, WRITE, OUTPUT, FORMAT, PRINT
.WRR	PUTRANDOM
.WRS	BYTEWRITE

* * *

APPENDIX D

OPERATING PROCEDURES

STAND-ALONE OPERATING SYSTEM

Loading the ALGOL Compiler

Extended ALGOL is a two-pass compiler for DGC computer configurations having 12K or more of memory. The tapes for ALGOL are:

Pass 1	-	Tape #1 Tape #2
Page 2	-	Tape #1 Tape #2

The user should exercise care in loading the ALGOL compiler. Loading will take some time and, if interrupted, must be restarted from the beginning.

To start the loading process, Tape #1 of Pass 1 is mounted in the input device (PTR or TTR) and loaded. When the tape is loaded, the system gives the following prompt:

```
LOAD PASS 1 TAPE #2, STRIKE ANY KEY
```

Tape #2 of Pass 1 is mounted in the reader, and the user strikes any teletypewriter key. When the tape is loaded, the system will ask what input device is being used as follows:

```
INPUT (1-TTR, 2-PTR):
```

The user responds with 1 or 2 as appropriate and follows the digit with a carriage return. The system then asks what output device is being used with the following prompt:

```
OUTPUT (1-TTP, 2-PTP):
```

The user responds with 1 or 2 as appropriate and follows the digit with a carriage return. The system then gives the following prompt:

```
INPUT: ALGOL SOURCE PROGRAM
LOAD xxx, STRIKE ANY KEY             ←xxx is TTR or PTR
```

Loading the ALGOL Compiler (Continued)

The user then mounts the ALGOL source program in the TTR or PTR. The source program is to be read from the input device, and an intermediate tape is to be punched out on the output device (TTP or PTP). The user makes sure that the output punch is turned ON, and then strikes a teletypewriter key. When the intermediate tape has been punched, the following prompt is given.

```
LOAD PASS 2, TAPE #1, STRIKE ANY KEY
```

The user loads the tape into the reader and strikes a teletypewriter key. When the tape is read in, the following prompt is given:

```
LOAD PASS 2, TAPE #2, STRIKE ANY KEY
```

The user loads the tape into the reader and strikes a teletypewriter key. when the tape is read in, the following prompt is given:

```
INPUT:  INTERMEDIATE TAPE
LOAD xxx, STRIKE ANY KEY                     ←xxx is TTR or PTR
```

The user loads the intermediate tape into the appropriate device and strikes a teletypewriter key. When the tape is loaded, results of compilation of the source code are output to the appropriate output device.

The sequence of prompts and responses for paper tape reader and punch as input and output devices would appear as follows:

```
LOAD PASS 1, TAPE #2, STRIKE ANY KEY
INPUT (1-TTR, 2-PTR): 2)                ←user responses underlined
OUTPUT (1-TTP, 2-PTP): 2)                ) is carriage return
INPUT: ALGOL SOURCE PROGRAM
LOAD PTR, STRIKE ANY KEY
LOAD PASS 2, TAPE #1, STRIKE ANY KEY
LOAD PASS 2, TAPE #2, STRIKE ANY KEY
INPUT: INTERMEDIATE TAPE
LOAD PTR, STRIKE ANY KEY
```

Assembling Source Programs

The output of compilation must be assembled with the DGC Extended Assembler. Each ALGOL-generated program is complete, with all necessary declarations and pseudo-ops to assemble using the Extended Assembler. (Although operation of the assembler is explained in the following paragraphs, the user can obtain additional information in the Extended Assembler User's Manual, document number 093-000040.)

The assembler can be loaded from paper tape, at which point it prints the prompt ASM. If the system has a cassette or magnetic tape unit, the CLI command ASM should be issued. In either case, the format of the ASM command line is:

ASM $\begin{Bmatrix} 0 \\ 1 \\ 2 \end{Bmatrix}$ filename-1 ... filename-n ↵

The ASM command line assembles one or more ASCII source files. Output can be an absolute or relocatable binary file. Files are assembled in the order specified in the command, from left to right. The same cassette or magnetic tape unit cannot be used for more than one output file or for both input and output, but can be used for more than one input file.

Action taken by the assembler is determined by the key (0, 1, or 2) specified in the ASM command line, as follows:

Key	Assembler Action
0	Perform pass one on the specified source file, then halt with the highest symbol table address (SST) in AC0.
1	Perform two passes on the specified input files, producing the specified binary and listing files. At the completion of pass two, the assembler outputs a new prompt (ASM) and awaits a new command line.
2	Perform pass two only on the specified input files, producing the specified relocatable binary and listing files. At the completion of this pass, the assembler outputs a new prompt (ASM) and awaits a new command line.

Assembling Source Programs (continued)

The following global switches can be appended to the key number.

Switch	Assembler Action
/E	Suppress assembly error messages, normally output to the $TTO. Because many errors can pass the compiler, but are detected by the assembler (especially errors in the use of reserved mnemonics), the assembly error messages should not be suppressed.
/T	Suppress the listing of the symbol table.
/U	Include local (user) symbols in the binary output file.

The following local switches can be appended to a file name:

Switch	Assembler Action
/B	Output absolute or relocatable binary file on the specified device.
/L	Output the listing file on the specified device.
/N	Do not list the specified input file on pass two.
/P	Pause before accepting input from the specified device. The message: PAUSE - NEXT FILE, devicename is printed by the assembler, which waits until any key is struck on the Teletype console before continuing assembly.
/S	Skip the specified source file during pass two.
/n	Repeat the specified source file n times, where n is a digit in the range of 2 through 9.

Loading User Programs

The Extended Relocatable Loader is used to load the binary tapes produced by the assembler. (For additional information on the loader, refer to the Extended Relocatable Loaders User's Manual, document number 093-000080.) All ALGOL relocatable binary programs must be

Loading User Programs (continued)

loaded first. If no main program is designated, the first program loaded is regarded as the main program at the start of execution. If a main program or procedure is designated, programs can be loaded in any order; a jump is made to the designated main program at the start of execution. A main program is designated by the word MAIN as the identifier of a procedure with no formal parameters or as the label of a begin block, as follows:

procedure MAIN; or MAIN: *begin*

The Relocatable Loader can be loaded from paper tape, at which point it prints the prompt RLDR. If the system has a cassette or magnetic tape unit, the CLI command RLDR should be issued. In either case, the format of the RLDR command line is:

RLDR main [subprograms] ALGOL-library-tapes trigger ↑

$\left[\begin{array}{l}\text{cassette-library}\\ \text{mag-tape-library}\end{array}\right]$ SOS-main-library)

where:

main is the name of the ALGOL main program or procedure.

subprograms are the names of one or more optional procedures to be called by main.

ALGOL-library-tapes are the names of ALGOL library tapes, to be loaded in the following order:

1. ALGOL Library Tape #1
2. ALGOL Library Tape #2
3. ALGOL Library Tape #3
4. One of the following library tapes:

   Nova Hardware Multiply/Divide Tape, if the machine configuration has the Nova multiply/divide hardware option.

   Nova 800/1200/Supernova Hardware Multiply/Divide Tape, if the machine configuration has the Nova 800, Nova 1200, or Supernova multiply/divide hardware option.

   Software Multiply/Divide Tape, if the machine configuration has no multiply/divide hardware option.

Loading User Programs (continued)

trigger — is the SOS trigger, created during SOS system generation, containing external symbols for those devices that are part of the system. (Refer to the Extended Relocatable Loaders User's Manual.)

cassette-library is the name of the tape containing the cassett library, to be loaded only when cassette units are part of the system.

mag-tape-library is the name of the tape containing the magnetic tape library, to be loaded only when magnetic tape units are part of the system.

SOS-main-library is the name of the tape containing the main library and all driver routines for SOS I/O, except cassette and magnetic tape units.

Upon completion of a successful load, the message

OK

is printed on the console and the system halts with the loaded program in core.

Executing and Restarting User Programs

A loaded program can be executed by pressing CONTINUE or by using the restart procedures:

1. Set switches to 000377.
2. Press RESET.
3. Press START.

Producing a Trigger

A trigger is produced using the SOS SYSGEN program, the binary loade or the core image loader/writer. Basically, the SYSGEN program acce a command line containing device driver entry symbols and outputs a file containing external references to the named devices. When the trigger is loaded in the RLDR command line (preceding other SOS libraries), the external normal references on the trigger load the named device drivers from the SOS libraries.

Producing a Trigger (Continued)

The format of the SYSGEN command line is:

(SYSG) $\underline{driver}_1$,...$\underline{driver}_n$ .RDSI [.CTB] output-device/O [triggername/T]

where: $\underline{driver}_1$ is one or more device driver entry symbols selected from the following chart:

Device Name	Device Driver Entry Symbol	Device
$CDR	.CDRD	card reader
CT0	.CTAD	cassette unit 0
CT0,1	.CTU1	cassette units 0 and 1
CT0,1,2	.CTU2	cassette units 0,1, and 2
⋮	⋮	⋮
CT0,1,2,3,4,5,6,7	.CTU7	cassette units 0,1,2,3,4,5, 6 and 7
$PTP	.PTPD	high-speed paper tape punch
$PTR	.PTRD	high-speed paper tape reader
$LPT	.LPTD	80-column line printer
	.L132	132-column line printer
MT0	.MTAD	magnetic tape unit 0
MT0,1	.MTU1	magnetic tape units 0 and 1
⋮	⋮	⋮
MT0,1,2,3,4,5,6,7	.MTU7	magnetic tape units 0,1,2,3, 4,5,6 and 7
$PLT	.PLTD	incremental plotter
$TTO/$TTI	.STTY	teletype printer and key-board
TTI1/TTO1	.TTI1	second teletype printer and keyboard
	.RTC1	real time clock, 10HZ
	.RTC2	real time clock, 100HZ
	.RTC3	real time clock, 1000HZ
	.RTC4	real time clock, 60HZ
	.RTC5	real time clock, 50HZ

For more detailed instructions for producing a trigger for SOS systems, refer to the Stand-alone Operating System User's Manual, 093-000062.

Error Messages

The possible error messages resulting from the ASM or RLDR command lines are:

Error Message	Meaning	ASM	RLDR
NO END	No END statement was specified in any source program.	X	
NO INPUT FILE SPECIFIED	No input file name was specified.		X
SAVE FILE IS READ/ WRITE PROTECTED	The save file device must permit both reading and writing: only cassette and magnetic tape units are permitted as save file devices.		X
I/O ERROR _n_	Input/output error _n_ where _n_ =		
	1 Illegal file name.	X	X
	7 Attempt to read a read-protected file.	X	X
	10 Attempt to write a write-protected file.	X	X
	12 Non-existent file.	X	X

RDOS OPERATING SYSTEM

Loading the ALGOL Compiler

The ALGOL compiler for the Real Time Disk Operating System configuration is supplied, as are other system tapes, as dumped tapes to be loaded using the LOAD command as described in the RDOS Manual, Document #093-000075. The tapes are:

ALGOL Dump Tape #1

ALGOL Dump Tape #2

Tape #1 contains AL1.SV, ALGOL.SV and LIBRARY.CM. Tape #2 contains AL2.SV. The ALGOL debugger, TRACE.SV, is supplied on a separate tape.

The ALGOL library tapes for RDOS are transferred to disk using the XFER command. The library tapes are input in the following order.

ALGOL Library Tape #1
ALGOL Library Tape #2
ALGOL Library Tape #3
Multiply/Divide tape, as described in Step 4, page D-5.*
Dummy SOS.LB

Compiling, Loading, and Executing ALGOL Programs under RDOS

The command invoking the ALGOL compiler to compile a main program or subroutine is described on the following page.

Each ALGOL program is compiled separately. The main program, its subprograms, and the library are then loaded (RLDR command).

To execute, give the name of the main program and a carriage return, as described in the RDOS Manual.

A sequence of commands for compilation, loading, and executing a program might be:

```
ALGOL MAIN)
ALGOL SUB1)
ALGOL SUB2)
RLDR MAIN SUB1 SUB2 @LIBRARY.CM@)
MAIN)
```

*Note: The appropriate Multiply/Divide tape can be linked by the user to SOFTMPYD.LB or LIBRARY.CM can be changed.

COMMAND FORMAT: ALGOL inputfilename [outputfilename]

PURPOSE: To compile an ALGOL source file. Output may be a relocatable binary file, intermediate source file, listing file, or combinations of all three. Asterisk and dash conventions are not permitted in the command line.

SWITCHES: By default, command execution produces an intermediate source file, inputfilename.SR (compiler output), and a relocatable binary file inputfilename.RB (assembler output). Once assembly is successfully completed, the intermediate source file is deleted. By default, no listing is produced.

GLOBAL:

- /A - Suppress assembly.
- /B - Brief listing (compiler source program input)
- /E - Suppress compiler error messages at $TTO. (Do not suppress assembler error messages.)
- /L - Produce listing to inputfilename.LS.
- /N - Do not produce relocatable binary file.
- /S - Save intermediate source output file.

LOCAL:

- /B - Direct relocatable binary output to specified file name. (Overrides global /N.)
- /E - List errors to specified file name.
- /L - Direct listing to specified file name. (Overrides global /L.)
- /S - Direct intermediate source output to specified file name.

EXTENSIONS: On input search for inputfilename.AL. If not found, search for inputfilename. On output, produce inputfilename.RB by default and other files with .LS or .SR extensions as determined by switches.

EXAMPLES:

ALGOL MAIN)

Produce relocatable binary file, MAIN.RB.

ALGOL/E/B SUBR $LPT/L)

Produce relocatable binary file SUBR.RB with a brief ALGOL source listing to the line printer. Suppress compiler error messages.

ALGOL/A RAY $PTP/S)

Do not invoke an assembly phase. Punch intermediate source output on high speed punch.

Using Disk Files to Produce Stand-Alone Files

To use RDOS to produce a stand-alone ALGOL program:

1. Compile the program as usual under RDOS.

2. Produce an appropriate Trigger using SOS SYSGEN.

3. Make sure that the actual SOS.LB (not the dummy) is on the system.

4. Load the assembled program, the trigger, and library, using the /C switch which causes the save file to start at location zero.

5. Make an absolute binary file, using the MKABS command with a /Z switch.

```
ALGOL OCTAL                    ←compile program

PROGRAM IS RELOCATABLE
R
LIST SOS.LB
SOS.LB          8578           ←size of SOS.LB shows that this is
R                               actual stand-alone operating system.
RLDR/C OCTAL TRIG @LIBRARY.CM@←load program with library begin-
 .MAIN                          ning at 0.

    NMAX Ø16227
    ZMAX ØØØ15Ø
    CSZE
     EST
     SST

R
MKABS/Z OCTAL $PTP             ←Make absolute binary for SOS.
R
```

If SOS.LB is not on disk,,read it in, e.g.,

```
    XFER $PTR SOS.LB )
```

To restart the loaded program, examine the contents of location 405 through the console switches. Location 405 contains the starting address of the program.

★ ★ ★

APPENDIX E

TIPS FOR EFFICIENT CODING AND REDUCED EXECUTION TIME

1. GENERAL

Any ALGOL expression or statement that maps directly into an assembler instruction will provide maximum efficiency, e.g., adding and subtracting integer 1 or multiplying by integer 2 when values are single-precision

2. NUMERICS - TYPE AND PRECISION

A. Default one-word (single-precision) integers and pointers are fast and efficient. They are not checked for overflow and may, in case of overflow, produce erroneous results.

Multi-precision integers and floating-point values take more space, since interpreters must be brought in.

It is possible, if overflow checking is desired on one-word integers, to force checking by declaring the precision to be one; this forces use of the multi-precision interpreter.

```
integer (1) array  A[3,4];  ←array elements checked for
                             overflow
```

It is also possible to declare a floating-point number with one-word precision. However, only a two or three digit decimal value can be stored in a one-word mantissa.

B. Unnecessary type conversion should be avoided.

```
real x,y; integer  i,j;

x:=y+2.0;                   ←not x:=y+2;
j:=i+2;                     ←not j:=i+2;
```

2. <u>NUMERICS - TYPE AND PRECISION</u> (Continued)

C. Unnecessary resetting of precision should be avoided. Less code is generated when the precision of variables is kept the same.

D. When raising a number to a power, the most efficient code is generated when a number is raised to an integer literal. A power >5 will require the use of the floating-point interpreter.

E. When floating-point precision is greater then default, care should be taken to insure that literals used with the variable in expressions have a like precision if the literal is a repeating fraction in binary. To define the precision of a literal, the number is followed by the letter P which in turn is followed by the precision in words.

For example, suppose x has a precision of 4 words. The result of evaluation when 1/3 is added to x will differ depending upon the precision of the literal.

```
real (4) x, y;

y:= x+1.0/3.0;           ←1/3 has only default precision

y:= x+1.0P4/3.0P4;       ←1/3 has 4-word precision
```

F. When formatting floating-point numbers using the I/O procedure, output, round the output to the number of digits desired.

```
output (1, "#.##", x+.005); ←.005 provides rounding to
                              2 decimal places
```

The procedures write and print do not require rounding--only the format specification of the output procedure requires this.

G. When formal and actual parameter types do not agree, both specifiers are generally kept. If the precisions differ, then the precision of results cannot be defined. Be careful about matching precision in passing parameters.

2. NUMERICS - TYPE AND PRECISION (Continued)

H. The precision of run-time routines is limited to 15 words on multi-precision integers and floating-point (60 digits).

I. The precision of ALGOL library mathematical functions is limited to about 25-30 digits.

J. Some type conversion errors are caught on the first compiler pass, but many do not show up until the second pass. As a general rule, the compiler initially accepts any type, whether or not it exists or is legal in the given expression, and later on in Pass 2 gives the error message:

illegal operand

3. EXPRESSIONS

If an expression is used several times, it is more efficient to do the arithmetic once only. This includes pointer expressions and subscripting.

```
ip := p+i;
 .
 .
 .
ip→J...
 .
 .
 .
ip→k
```

```
m := z[n];
 .
 .
 .
R1 := m↑2+m;
```

4. SUBSCRIPTING

It is more efficient to use based variables than subscripts of arrays.

4. SUBSCRIPTING (Continued)

```
begin based integer j; pointer p;
allocate (p,30);
(p+i)→j:=expression;

   is more efficient than:

begin integer array A[0:100];
a[i] := expression;
```

5. BIT HANDLING AND MASKING

Whenever possible, keep masking literals within the default one-word integer limit. Multi-precision integers can be used with *and*, *or*, and *not* operations, but they will bring in the multi-precision interpreter, requiring additional code. Any literal $>2^{16}-1$ forces multi-precision arithmetic unless the user specifies single-precision by appending Pl precision.

Use of binary and octal literals is more efficient than use of the shift and rotate functions.

6. COMPARISON OF REAL VALUES

Be careful about equality comparisons involving real variables. Real variables are very seldom equal.

7. LITERALS

Declaring a literal is more efficient than assigning a value to a variable.

```
literal pi(3.14159);

    is more efficient than:

pi := 3.14159;
```

8. STATEMENTS

A. In a *for* statement, a *step* clause is more efficient than a list.

8. STATEMENTS (Continued)

B. The control variable in a *for* statement must be declared with the precision needed for the range of the variable.

```
for i := -32000 step 1 until 32000 ←be sure i is de-
do ...                                clared: integer (2)
```

C. In many instances, multiple assignment generates better code than individual assignment.

```
X := Y := 0;        is more efficient than:

X := 0;
Y := 0;
```

D. The null statement is defined by a semicolon. An extra semicolon appearing in the declaration section of a block or procedure will cause termination of declarations since it will be interpreted as the beginning of the statement section.

```
begin; integer a,b;          ←no declarations will be inter-
                              preted because of the semi-
                              colon after begin.

begin real c;; integer a; ←the integer declaration will
                              not be interpreted because
                              of the null statement follow-
                              ing the real declaration.
```

E. Programmers who are used to BASIC must be careful about the order of clauses in the *for* statement.

```
for...step...until...do     ←step and until are both re-
                              quired and attempts to use
                              a FOR-TO or FOR-UNTIL for-
                              mat as in BASIC will cause
                              errors.
```

9. STRINGS

A. To make a string shorter, assign a shorter string to the string variable function.

```
x := substr (x,1,3);       ←if x was originally 5
    .                        characters, the result of
    .                        the first assignment is a
    .                        3-character string and of
x := "";                     the second is a null string.
```

B. To make a substring of a string, using the same string identifier, first copy the string.

```
a:="XYZYZX";
b:=a;
a:=substr(b,2,4);          ←avoids attempting to copy
                             string a to itself.
```

D. Although a *based* string looks like a string, the programmer should remember that the specifiers are different and that the current length is always equal to the maximum length for a *based* string. For example, if p is a pointer and s is a *based* string:

```
p→s := "";
```

The statement in the example does nothing. No adjustment of string length can be made when there is no current length.

10. SCOPE AND STACK HANDLING

A. Strings and arrays require more space than scalars. For such quantities, setting up a number of blocks so that space is allocated and released as blocks are entered and exited is efficient. (The BLKSTART and BLKEND run-time routines are quite efficient.)

B. Temporaries (assigned storage) can be reused so that the limit of a page need not be exceeded, and temporaries should be limited to a single page whenever possible. With each additional page of assigned storage, the loss

10. <u>SCOPE AND STACK HANDLING</u> (Continued)

B. in space and time becomes greater. .SP must be reset many times. Temporaries usually exceed the page limit only when there are a great many large-precision temporaries and long strings.

C. Page 0 contains one word for .FP, one for .SP, and one for each run-time routine called. In addition, assigned *own* and assigned *external* storage is in page 0, which can cause an overflow of page 0. Avoid too many *own* and *external* variables.

11. <u>LABELS AND TRANSFERS</u>

A. Declarations cannot be labeled.

B. Coding between two labels is optimized. Therefore, it is efficient to keep transfer points down to a minimum and to create, where possible, a single body of code into which to transfer.

C. The only way to transfer to (*go to*) another procedure is through a parameter label.

12. <u>IDENTIFIERS</u>

A. External variable and procedure names must not conflict with ALGOL Library routine identifiers. For a complete list of these identifiers, print a core map during loading.

B. External identifiers must be unique within the first five characters for assembler compatibility.

C. Every variable must be declared even if it appears only on the lefthand side of an assignment or as the controlled variable in a *for* statement.

13. <u>FUNCTIONS AND PROCEDURES</u>

A. Use of built-in functions is relatively inexpensive in space.

B. Parameters of built-in functions are converted as required.

C. It is not possible to pass built-in functions by name.

13. <u>FUNCTIONS AND PROCEDURES</u> (Continued)

D. Care must be taken when passing procedures by name when three or more levels of procedures are involved. When stack frames are created, the outermost procedure is at level one, the next at level two, etc. When an attempt is made to reference a global variable from a procedure at a lower level, the search is conducted for the next higher level, then the next, etc. However, it is possible that the search will actually encounter a lower level:

Stack Frames

X	level 1	Stack frame of X; level 1.
Y	level 2	Stack frame of Y, a procedure internal to X at level 2.
Z	level 3	Z is a procedure internal to Y at level 3
Z	level 3	Z calls itself; level 3.
Y2	level 2	Y2 is a procedure internal to X at level 2 and passed as a parameter to Y and Z.

Suppose X passes Y2 to Z. Then later there is a reference within Y2 to a variable that is global to Y2 and local to X. When a search is attempted up the stack frames from a level 2 procedure, the search algorithm expects either to encounter the same level procedure (2) or a higher level procedure (1). When level 3 is encountered instead, results of referencing the global variable are undefined.

E. Taking the address of a function value will produce an undefined result.

F. Some built-in functions, such as the mathematical functions, allow expressions to be used as parameters. Other built-in functions, such as the address function, allow only variables to be used as parameters. For such built-in functions, be sure to assign the desired expression to a variable and then use the variable as the function parameter.

14. RUN-TIME OVERHEAD

For any ALGOL programming, the following programs are always required:

SPINIT	89 words		Stack initialization.
CALL SAVE RETURN	17 239 166		Standard ALGOL call/save/ return.
ASAV ARET	27		Runtime save and return.
ARER ARER	119 453	(short form) (long form)	Runtime error.
ADDRS OADDR	13 14		Used by runtime to interpret parameter descriptors.
BLKSTART BLKEND	25		Block start and end.

The basic package, as defined above, does not include floating point, the string package, or the I/O package supplied with ALGOL. It requires 15 words of page zero and approximately .6K additional words. (Under RDOS, page 1 is always reserved by the loader; this is not included in the stated requirements.)

The basic package plus array allocation requires 1.1 K.

The basic package plus floating point requires 1.2 K. The use of floating point functions is not included.

The basic package plus generalized floating point and multi-precision integer requires 2.5 K.

The basic package plus string package requires 1.4 K.

The basic package plus formatted I/O requires 3.4 K.

If all the above features are included, the package requires 4.3 K plus 55 words in page zero.

For the stand-alone operating system, add 1.2 K to the overhead given above.

15. COMPILER ERRORS

The up arrow does not necessarily point to the error itself. If no error is found where the arrow points, check to the left and to the right of the arrow in the statement. If there still appears to be no error, check previous

15. COMPILER ERRORS (Continued)

statements that would affect the statement in which the error was found.

16. STRING SPECIFIERS

In allocated storage, the data area for a string scalar contains the string data. However, the data area for an array of strings contains the string specifier for each string of the array. To manipulate specifiers rather than the data, use a based string array consisting of one element.

* * *

APPENDIX F

SAMPLE PROGRAMS

The programs following show some of the features of DGC ALGOL. They range from very simple programs, such as FACTORIAL, to a sophisticated program, HELP.

FACTORIAL, defined in the main program shown below, is a recursive procedure. The precision of values returned is set at 15 to allow large multi-precision integers. Output in the main program is directed to the teletype by the open call. Note also the output call, where the format permits a variable string of digits, immediately followed by a carriage return. The first three number symbols allow up to three digits for the value of N. That value is followed by a triple space and then by the value of factorial N and the carriage return.

Output from the program is shown on the following page.

```
BEGIN

INTEGER (15) PROCEDURE FACTORIAL (N); INTEGER (15) N;
FACTORIAL := IF N>1 THEN N*FACTORIAL(N-1) ELSE 1;

INTEGER (15) N;

OPEN (1, "$TTO");
FOR N := 1 STEP 1 UNTIL 50 DO
   OUTPUT (1, "###   #<15>", N, FACTORIAL(N));

END
```

1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600
13	6227020800
14	87178291200
15	1307674368000
16	20922789888000
17	355687428096000
18	6402373705728000
19	121645100408832000
20	2432902008176640000
21	51090942171709440000
22	1124000727777607680000
23	25852016738884976640000
24	620448401733239439360000
25	15511210043330985984000000
26	403291461126605635584000000
27	10888869450418352160768000000
28	304888344611713860501504000000
29	8841761993739701954543616000000
30	265252859812191058636308480000000
31	8222838654177922817725562880000000
32	263130836933693530167218012160000000
33	8683317618811886495518194401280000000
34	295232799039604140847618609643520000000
35	10333147966386144929666651337523200000000
36	371993326789901217467999448150835200000000
37	13763753091226345046315979581580902400000000
38	523022617466601111760007224100074291200000000
39	20397882081197443358640281739902897356800000000
40	815915283247897734345611269596115894272000000000
41	33452526613163807108170062053440751665152000000000
42	1405006117752879898543142606244511569936384000000000
43	60415263063373835637355132068513997507264512000000000
44	2658271574788448768043625811014615890319638528000000000
45	119622220865480194561963161495657715064383733760000000000
46	5502622159812088949850305428800254892961651752960000000000
47	258623241511168180642964355153611979969197632389120000000000
48	12413915592536072670862289047373375038521486354677760000000000
49	608281864034267560872252163321295376887552831379210240000000000
50	30414093201713378043612608166064768844377641568960512000000000002

The program on the page following uses a *for* loop to compute a series of X-Y coordinates for an earth satellite at equal time intervals.

Note that a large part of the header information for output and the format for real values to be output are written as literals, referenced in output statements. Shorter formatting information appears directly in the output call format.

The values of the coordinates and the information on which the coordinates are based is declared with 8-word precision to provide greater accuracy.

```
BEGIN

COMMENT:          ****************************
                  *                          *
                  *  SATELLITE ORBIT PROBLEM  *
                  *                          *
                  ****************************

THIS PROGRAM COMPUTES THE PATH OF A SATELLITE IN AN XY
COORDINATE SYSTEM.  THE POINTS SPECIFY THE POSITION OF THE
SATELLITE AT EQUAL TIME INTERVALS, I. E., THE SATELLITE
REQUIRES THE SAME AMOUNT OF TIME TO MOVE FROM POSITION 2
TO POSITION 3 AS FROM POSITION 1 TO POSITION 2.  C IS A
CONSTANT DETERMINED BY THE GRAVITATIONAL ATTRACTION OF THE
EARTH AND THE TIME INTERVAL;

INTEGER K, N;
REAL (8) C, X1, Y1, X2, Y2, X3, Y3;

LITERAL
TITLE ("<15>COORDINATES OF SATELLITE ORBIT<15><15>"),
HEADER ("POINT       X COORDINATE           Y COORDINATE<15><15>"),
FORMAT ("##      ###.###############  ###.###############<15>");

OPEN (1, "$TTI");
OPEN (2, "$TTO");
OUTPUT (2, "NUMBER OF POINTS: ");
READ (1, N);
OUTPUT (2, "GRAVITATIONAL CONSTANT: ");
READ (1, C);
OUTPUT (2, "COORDINATES: ");
READ (1, X1, Y1, X2, Y2);

OUTPUT (2, TITLE);
OUTPUT (2, HEADER);

OUTPUT (2, FORMAT, 1, X1, Y1);
OUTPUT (2, FORMAT, 2, X2, Y2);

FOR K := 1 STEP 1 UNTIL N DO BEGIN

        X3 := 2 * X2 + X1 * (C/((X1↑2+Y1↑2) ↑ 1.5) -1);
        Y3 := 2 * Y2 + Y1 * (C/((X1↑2+Y1↑2) ↑ 1.5) -1);
        OUTPUT (2, FORMAT, K+2, X3, Y3);
        X1 := X2;
        X2 := X3;
        Y1 := Y2;
        Y2 := Y3;
        END;
END
R
ALGOL SATELLITE
```

```
SATELLITE
NUMBER OF POINTS: 10
GRAVITATIONAL CONSTANT: 1000
COORDINATES: 103, 64, 94, 81

COORDINATES OF SATELLITE ORBIT

POINT        X COORDINATE           Y COORDINATE

  1      103.0000000000000000   64.0000000000000000
  2       94.0000000000000000   81.0000000000000000
  3       85.0577616620752599   98.0358907414836566
  4       76.1647242968633792  115.1141780658369049
  5       67.3105891368010237  132.2373033010300066
  6       58.4854161679414929  149.4042015409320096
  7       49.6808463553870941  166.6115764110970910
  8       40.8904370928790893  183.8551251825812089
  9       32.1094811295371042  201.1303768977927635
 10       23.3346450477221041  218.4331453544294743
 11       14.5636091560343964  235.7597177981035999
 12        5.7947744332426462  253.1068951511053371
R
SATELLITE
NUMBER OF POINTS: 4
GRAVITATIONAL CONSTANT: 1000
COORDINATES: 103, 100.5, 94, 81

COORDINATES OF SATELLITE ORBIT

POINT        X COORDINATE           Y COORDINATE

  1      103.0000000000000000  100.5000000000000000
  2       94.0000000000000000   81.0000000000000000
  3       85.0345613476326838   61.5337224799716963
  4       76.1183236679782270   42.1098415428129842
  5       67.2756193048082811   22.7391716602875985
  6       58.5485482974201304    3.4324719501779192
R
SATELLITE
NUMBER OF POINTS: 4
GRAVITATIONAL CONSTANT: 1400
COORDINATES: 103, 100.5, 94, 81

COORDINATES OF SATELLITE ORBIT

POINT        X COORDINATE           Y COORDINATE

  1      103.0000000000000000  100.5000000000000000
  2       94.0000000000000000   81.0000000000000000
  3       85.0483858866857573   61.5472114719603748
  4       76.1656531351695178   42.1537781599381779
  5       67.3858275531441648   22.8348159769868502
  6       58.7676396500004157    3.6053119497447690
```

The program following produces a plot of a two-dimensional function:

$$Z = F(X,Y)$$

using strings. The program demonstrates the way in which the substr function can be used to plot topological problems such as the earth's magnetic field or land contours, using different letters within a string to represent different values.

```
BEGIN   REAL X, Y;
        STRING (80) LINE;
        LITERAL SYMBOL ("A B C D E F G H I J K "), CR ("<15>");
        EXTERNAL REAL PROCEDURE FNC;

OPEN(0, "$TTO");

FOR Y := 30 STEP -1 UNTIL -30 DO BEGIN

   FOR X := -35 STEP 1 UNTIL 35 DO

        SUBSTR(LINE, X+36) :=  SUBSTR(SYMBOL, FNC(X, Y));

   WRITE(0, LINE, CR);
   END

END


REAL PROCEDURE FNC (X, Y); VALUE X, Y; REAL X, Y;
        FNC := (COS(X/7) + COS(Y/7) + 2) * 2.5;
```

```
  BBB    AAAAAAAAA    BBB  CCCC         CCCC  BBB    AAAAAAAAA    BBB
  BB    AAAAAAAAAAA    BB   CCCCC     CCCCC   BB    AAAAAAAAAAA    BB
 BBB   AAAAAAAAAAAAA   BBB   CCCCCCCCCCCCC   BBB   AAAAAAAAAAAAA   BBB
BBB   AAAAAAAAAAAAAAA   BBB   CCCCCCCCCCC   BBB   AAAAAAAAAAAAAAA   BBB
BB   AAAAAAAAAAAAAAAAA   BBB   CCCCCCCCC   BBB   AAAAAAAAAAAAAAAAA   BB
BB   AAAAAAAAAAAAAAAAA   BBB    CCCCCCC    BBB   AAAAAAAAAAAAAAAAA   BB
B   AAAAAAAAAAAAAAAAAAA   BBB    CCCCC    BBB   AAAAAAAAAAAAAAAAAAA   B
B   AAAAAAAAAAAAAAAAAAA   BBB     CCC     BBB   AAAAAAAAAAAAAAAAAAA   B
B   AAAAAAAAAAAAAAAAAAA   BBB      C      BBB   AAAAAAAAAAAAAAAAAAA   B
B   AAAAAAAAAAAAAAAAAAA   BBB      C      BBB   AAAAAAAAAAAAAAAAAAA   B
B   AAAAAAAAAAAAAAAAAAA   BBB     CCC     BBB   AAAAAAAAAAAAAAAAAAA   B
BB   AAAAAAAAAAAAAAAAA   BBB     CCCCC     BBB   AAAAAAAAAAAAAAAAA   BB
BB   AAAAAAAAAAAAAAAAA   BBB    CCCCCCC    BBB   AAAAAAAAAAAAAAAAA   BB
BBB   AAAAAAAAAAAAAAA   BBB    CCCCCCCCC    BBB   AAAAAAAAAAAAAAA   BBB
 BBB   AAAAAAAAAAAAA   BBB    CCCCCCCCCCC    BBB   AAAAAAAAAAAAA   BBB
  BB    AAAAAAAAAAA    BB    CCCC     CCCC    BB    AAAAAAAAAAA    BB
  BBB    AAAAAAAAA    BBB   CCC           CCC   BBB    AAAAAAAAA    BBB
C  BBB     AAAAA     BBB   CCC           CCC   BBB     AAAAA     BBB  C
CC  BBBB           BBBB   CCC    DDDDD    CCC   BBBB           BBBB  CC
CCC  BBBB         BBBB   CC    DDDDDDDDD    CC   BBBB         BBBB  CCC
 CCC   BBBB     BBBB    CC    DDDDDDDDDDD    CC    BBBB     BBBB   CCC
  CCC   BBBBBBBBBBB    CC   DDDDD     DDDDD   CC    BBBBBBBBBBB   CCC
   CCC   BBBBBBBBB    CC   DDDD         DDDD   CC    BBBBBBBBB   CCC
D  CCCC    BBBBB     CCC  DDDD           DDDD  CCC     BBBBB    CCCC  D
D   CCCC            CCC   DDD    EEEEE    DDD   CCC            CCCC   D
DD   CCCC          CCC   DDD    EEEEEEE    DDD   CCC          CCCC   DD
DDD   CCCC        CCC   DDD    EEEEEEEEE    DDD   CCC        CCCC   DDD
DDD   CCCCC      CCCC   DDD   EEEEEEEEEEE   DDD   CCCC      CCCCC   DDD
 DDD   CCCCC    CCCC   DDD   EEEEEEEEEEEEE   DDD   CCCC    CCCCC   DDD
 DDD   CCCCCC  CCCCC   DDD   EEEEEEEEEEEEE   DDD   CCCCC  CCCCCC   DDD
 DDD   CCCCCCCCCCCCC   DDD   EEEEEE EEEEEE   DDD   CCCCCCCCCCCCC   DDD
 DDD   CCCCCC  CCCCC   DDD   EEEEEEEEEEEEE   DDD   CCCCC  CCCCCC   DDD
 DDD   CCCCC    CCCC   DDD   EEEEEEEEEEEEE   DDD   CCCC    CCCCC   DDD
DDD   CCCCC      CCCC   DDD   EEEEEEEEEEE   DDD   CCCC      CCCCC   DDD
DDD   CCCC        CCC   DDD    EEEEEEEEE    DDD   CCC        CCCC   DDD
DD   CCCC          CCC   DDD    EEEEEEE    DDD   CCC          CCCC   DD
D   CCCC            CCC   DDD    EEEEE    DDD   CCC            CCCC   D
D  CCCC    BBBBB     CCC  DDDD           DDDD  CCC     BBBBB    CCCC  D
   CCC   BBBBBBBBB    CC   DDDD         DDDD   CC    BBBBBBBBB   CCC
  CCC   BBBBBBBBBBB    CC   DDDDD     DDDDD   CC    BBBBBBBBBBB   CCC
 CCC   BBBB     BBBB    CC    DDDDDDDDDDD    CC    BBBB     BBBB   CCC
CCC  BBBB         BBBB   CC    DDDDDDDDD    CC   BBBB         BBBB  CCC
CC  BBBB           BBBB   CCC    DDDDD    CCC   BBBB           BBBB  CC
C  BBB     AAAAA     BBB   CCC           CCC   BBB     AAAAA     BBB  C
  BBB    AAAAAAAAA    BBB   CCC           CCC   BBB    AAAAAAAAA    BBB
  BB    AAAAAAAAAAA    BB    CCCC     CCCC    BB    AAAAAAAAAAA    BB
 BBB   AAAAAAAAAAAAA   BBB    CCCCCCCCCCC    BBB   AAAAAAAAAAAAA   BBB
BBB   AAAAAAAAAAAAAAA   BBB    CCCCCCCCC    BBB   AAAAAAAAAAAAAAA   BBB
BB   AAAAAAAAAAAAAAAAA   BBB    CCCCCCC    BBB   AAAAAAAAAAAAAAAAA   BB
BB   AAAAAAAAAAAAAAAAA   BBB     CCCCC     BBB   AAAAAAAAAAAAAAAAA   BB
B   AAAAAAAAAAAAAAAAAAA   BBB     CCC     BBB   AAAAAAAAAAAAAAAAAAA   B
B   AAAAAAAAAAAAAAAAAAA   BBB      C      BBB   AAAAAAAAAAAAAAAAAAA   B
B   AAAAAAAAAAAAAAAAAAA   BBB      C      BBB   AAAAAAAAAAAAAAAAAAA   B
B   AAAAAAAAAAAAAAAAAAA   BBB     CCC     BBB   AAAAAAAAAAAAAAAAAAA   B
B   AAAAAAAAAAAAAAAAAAA   BBB    CCCCC    BBB   AAAAAAAAAAAAAAAAAAA   B
BB   AAAAAAAAAAAAAAAAA   BBB    CCCCCCC    BBB   AAAAAAAAAAAAAAAAA   BB
BB   AAAAAAAAAAAAAAAAA   BBB   CCCCCCCCC   BBB   AAAAAAAAAAAAAAAAA   BB
BBB   AAAAAAAAAAAAAAA   BBB   CCCCCCCCCCC   BBB   AAAAAAAAAAAAAAA   BBB
 BBB   AAAAAAAAAAAAA   BBB   CCCCCCCCCCCCC   BBB   AAAAAAAAAAAAA   BBB
  BB    AAAAAAAAAAA    BB   CCCCC     CCCCC   BB    AAAAAAAAAAA    BB
  BBB    AAAAAAAAA    BBB  CCCC         CCCC  BBB    AAAAAAAAA    BBB
R
```

The following four pages contain the source code and some output for a program that produces character representation of integers input by the user. Procedures THOUSANDSSTRING and HUNDREDSSTRING provide, respectively, character representations of integers 100 or greater and integers less than 100. Note, in particular, the use of subscripted labels in HUNDREDSSTRING.

The program shows how the length function is used to keep track of current string length, providing the next location to be filled, while the substr function is used in filling the appropriate locations in the string.

```
BEGIN

INTEGER (1) I; STRING S; REAL X;
EXTERNAL STRING (20) PROCEDURE HUNDREDSSTRING;
LITERAL FORMAT ("# DOLLARS AND # CENTS <15><15>");


COMMENT:        ***********************************
                *                                 *
                *  THOUSANDSSTRING PRODUCES THE   *
                *  CHARACTER REPRESENTATION FOR   *
                *  AN INTEGER LESS THAN 32768.    *
                *  HUNDREDSSTRING IS CALLED FOR   *
                *  INTEGER PARTS LESS THAN 100.   *
                *                                 *
                ***********************************
                                                        ;


STRING (60) PROCEDURE THOUSANDSSTRING (NUMBER);
        VALUE NUMBER; INTEGER NUMBER;

BEGIN   STRING (60) OUT;
        INTEGER THOUSANDS, HUNDREDS, UNITS;

        PROCEDURE PUT (S); STRING S;
        BEGIN   INTEGER N;

                N := LENGTH(OUT);
                IF N>0 THEN BEGIN
                        SUBSTR(OUT,N+1) := " ";
                        N := N+1;
                        END;

                SUBSTR(OUT,N+1,N+LENGTH(S)) := S;
                END;

        THOUSANDS := NUMBER/1000;
        HUNDREDS := (NUMBER-THOUSANDS*1000)/100;
        UNITS := NUMBER-THOUSANDS*1000-HUNDREDS*100;

        IF THOUSANDS>0 THEN BEGIN
                PUT(HUNDREDSSTRING(THOUSANDS));
                PUT("THOUSAND");
                END;

        IF HUNDREDS>0 THEN BEGIN
                PUT(HUNDREDSSTRING(HUNDREDS));
                PUT("HUNDRED");
                END;

        IF (UNITS>0) OR (LENGTH(OUT)=0) THEN
                PUT(HUNDREDSSTRING(UNITS));

        THOUSANDSSTRING := OUT;
        END;
```

```
COMMENT:            ********************************
                    *                              *
                    *  THE MAIN PROGRAM IS EXECUTED *
                    *  AS A COMMAND FROM DOS WITH A *
                    *  SINGLE ARGUMENT TO BE PRINTED *
                    *  IN CHARACTER FORM AS DOLLARS *
                    *  AND CENTS.                   *
                    *                              *
                    ********************************
                                                          ;

        OPEN (1,"COM.CM");
        OPEN (2,"$TTO");
        COMARG (1, S);
        COMARG (1, S);
        I := X := S;
        OUTPUT (2, FORMAT, THOUSANDSSTRING(I), (X-I)*100);

        END



STRING (20) PROCEDURE HUNDREDSSTRING (NUMBER);
        VALUE NUMBER; INTEGER NUMBER;

BEGIN   INTEGER N, M; STRING (20) S1, S2, OUT;
        S1 := S2 := OUT := "";
        M := NUMBER/10; N := NUMBER-M*10;

TENS:   GOTO TEN[M];                COMMENT: DISPATCH ON TENS DIGIT;

TEN[0]:           GOTO ONES;
TEN[1]:           GOTO TEEN[N];     COMMENT: TEENS ARE SPECIAL;

TEEN[0]:          OUT := "TEN"; GOTO DONE;
TEEN[1]:          OUT := "ELEVEN"; GOTO DONE;
TEEN[2]:          OUT := "TWELVE"; GOTO DONE;
TEEN[3]:          OUT := "THIRTEEN"; GOTO DONE;
TEEN[4]:          OUT := "FOURTEEN"; GOTO DONE;
TEEN[5]:          OUT := "FIFTEEN"; GOTO DONE;
TEEN[6]:          OUT := "SIXTEEN"; GOTO DONE;
TEEN[7]:          OUT := "SEVENTEEN"; GOTO DONE;
TEEN[8]:          OUT := "EIGHTEEN"; GOTO DONE;
TEEN[9]:          OUT := "NINETEEN"; GOTO DONE;

TEN[2]:           S1 := "TWENTY"; GOTO ONES;
TEN[3]:           S1 := "THIRTY"; GOTO ONES;
TEN[4]:           S1 := "FORTY"; GOTO ONES;
TEN[5]:           S1 := "FIFTY"; GOTO ONES;
TEN[6]:           S1 := "SIXTY"; GOTO ONES;
TEN[7]:           S1 := "SEVENTY"; GOTO ONES;
TEN[8]:           S1 := "EIGHTY"; GOTO ONES;
TEN[9]:           S1 := "NINETY"; GOTO ONES;
```

```
ONES:    GOTO ONE[N];                  COMMENT: DISPATCH ON ONES DIGIT;

ONE[Ø]:          IF LENGTH(S1)=Ø THEN OUT:="ZERO"
                          ELSE OUT := S1; GOTO DONE;
ONE[1]:          S2 := "ONE"; GOTO PACK;
ONE[2]:          S2 := "TWO"; GOTO PACK;
ONE[3]:          S2 := "THREE"; GOTO PACK;
ONE[4]:          S2 := "FOUR"; GOTO PACK;
ONE[5]:          S2 := "FIVE"; GOTO PACK;
ONE[6]:          S2 := "SIX"; GOTO PACK;
ONE[7]:          S2 := "SEVEN"; GOTO PACK;
ONE[8]:          S2 := "EIGHT"; GOTO PACK;
ONE[9]:          S2 := "NINE"; GOTO PACK;

PACK:    IF LENGTH(S1)>Ø THEN BEGIN

                 N := LENGTH(OUT)+1;
                 SUBSTR(OUT,N,N+LENGTH(S1)-1) := S1;
                 N := LENGTH(OUT)+1;
                 SUBSTR(OUT,N) := "-";
                 N := N+1;
                 SUBSTR(OUT,N,N+LENGTH(S2)-1) := S2;
                 END

         ELSE    OUT := S2;

DONE:    HUNDREDSSTRING := OUT; END
```

```
R
CHECK 29.98
TWENTY-NINE DOLLARS AND 98 CENTS

R
CHECK 100
ONE HUNDRED DOLLARS AND 0 CENTS

R
CHECK 1971
ONE THOUSAND NINE HUNDRED SEVENTY-ONE DOLLARS AND 0 CENTS

R
CHECK 34000
INTEGER OVERFLOW: LOCATION 11015
THIRTY-TWO THOUSAND SEVEN HUNDRED SIXTY-SEVEN DOLLARS AND 0 CENTS

R
CHECK 19.27
NINETEEN DOLLARS AND 27 CENTS

R
CHECK 12.07
TWELVE DOLLARS AND 7 CENTS

R
CHECK 100.02
ONE HUNDRED DOLLARS AND 2 CENTS

R
CHECK 1000.02
ONE THOUSAND DOLLARS AND 2 CENTS

R
CHECK 10000.02
TEN THOUSAND DOLLARS AND 2 CENTS

R
CHECK 2999
TWO THOUSAND NINE HUNDRED NINETY-NINE DOLLARS AND 0 CENTS

R
CHECK 2.19
TWO DOLLARS AND 19 CENTS

R
```

To understand the following program, refer to "How to Use the Nova Computers", Chapter 6 on Analog Conversion.

Procedure SAMPLE is an assembly language program called from an ALGOL main program. SAMPLE collects data from an A/D converter with Extended Interface, Type 4033 (page 6-6 of "How to Use the Nova Computers"). SAMPLE initiates a request for data from multiple channels. The sample rate is controlled by an internal clock in the A/D converter. The clock rate is variable from 100 KHz to 10 Hz.

The number of channels to be sampled is determined by the upper bound of the first dimension of an array passed to SAMPLE. Multiple samples for each channel may be specified by dimensioning the array for two dimensions, where the upper bound of the second dimension specifies the number of samples per channel.

After I/O is initiated from the converter, SAMPLE will immediately return to the caller. The caller may choose to wait for I/O completion at any time by calling WAIT or SAMPLE for another array. This allows sampling to be buffered.

In the example, a 12-bit converter is used. The least significant bit is .0024 volts. The program double buffers its samples, averaging and outputting one set of samples while another set is being collected.

Procedure SAMPLE examines interrupts for ADCV interrupt. Other interrupts are passed to the operating system.

The ALGOL main program declares SAMPLE as an external procedure and contains the array information to be passed as a parameter to SAMPLE.

Note in SAMPLE the equivalence statements for the array specifier, conversion word count, highest channel scanned, and frame size. Unlike normal ALGOL conventions as given in Appendix B, the stack argument displacement S is included as part of the definition of the variable name in the assembly language program. This provides for more conventional use of the variables as displacements in machine instructions.

```
BEGIN   EXTERNAL PROCEDURE SAMPLE;
        LITERAL HIGHCH (5), TIMES (2);
        INTEGER ARRAY A, B[Ø:HIGHCH, 1:TIMES];

        PROCEDURE AVGOUT (A); INTEGER ARRAY A;

        BEGIN REAL AVERAGE; INTEGER I, J;
        FOR I := Ø STEP 1 UNTIL HIGHCH DO BEGIN
        AVERAGE := Ø;
        FOR J := 1 STEP 1 UNTIL TIMES DO
           AVERAGE := AVERAGE + (A[I,J]/HIGHCH)*.ØØ24;
        OUTPUT (1, "AVERAGE: #.###<15>", AVERAGE);
        END;
        END;

        OPEN (1, "$TTO");
        SAMPLE (B);
LOOP:   SAMPLE (A);
        AVGOUT (B);
        SAMPLE (B);
        AVGOUT (A);
        GO TO LOOP;

        END
```

```
;        ***********************************
;        *  A/D CONVERSION PROGRAM FOR THE  *
;        *   EXTENDED INTERFACE TYPE 4033   *
;        ***********************************

; THIS PROCEDURE INITIATES AN I/O REQUEST TO
; THE A/D CONVERTER TO READ SAMPLES FROM A
; NUMBER OF CHANNELS INTO A TWO-DIMENSIONAL
; ARRAY.  THE ARRAYS (MULTIPLE ARRAYS MAY BE
; USED FOR BUFFERING) SHOULD BE DECLARED IN
; THE CALLING PROGRAM AS:
;
;        INTEGER ARRAY A, B[Ø:HIGHCH, 1:TIMES];
;
;        WHERE A AND B ARE ARRAYS, HIGHCH IS
;        THE HIGHEST CHANNEL SAMPLED AND TIMES
;        IS THE NUMBER OF SAMPLES PER CHANNEL.
;
; TWO ENTRIES ARE PROVIDED TO THE PROGRAM -
; SAMPLE, WHICH WAITS FOR I/O TO BE COMPLETED
; AND INITIATES A NEW CONVERSION REQUEST AND
; WAIT, WHICH SIMPLY WAITS FOR I/O TO FINISH.
; I/O MAY BE BUFFERED BY CALLING SAMPLE FOR
; ONE ARRAY AND THEN PROCESSING DATA FOR A
; SECOND ARRAY WHILE THE I/O IS IN PROGRESS.

; THE CALLING SEQUENCES ARE:
;
;        SAMPLE (A);
;        WAIT;

        .TITL   ADCONV
        .ENT    SAMPLE
        .ENT    WAIT
        .EXTU

        T.INTEGER=1B11+1        ;TYPE SPECIFIER FOR INTEGER
        C.PARAMETER=2B7         ;STORAGE CLASS FOR PARAMETER
        S.ARRAY=1B3             ;SHAPE SPECIFIER FOR ARRAY
        ASPCF=T.INTEGER+C.PARAMETER+S.ARRAY

        S=-167                  ;INITIAL VARIABLE DISPLACEMENT
        SP=1BØ-S                ;PARAMETER DESCRIPTOR ADJUSTMENT

        ADATA=S                 ;TWO WORD ARRAY SPECIFIER
        WDCNT=ADATA+2           ;WORD COUNT FOR CONVERSION
        HIGHCH=WDCNT+1          ;HIGHEST CHANNEL SCANNED
        FSIZE=HIGHCH-S+1        ;ASSIGNED STACK FRAME SIZE
```

```
        .ZREL

SYSINT: .BLK    1                ;SAVED SYSTEM INTERRUPT ADDRESS
SAVACØ: .BLK    1                ;SAVED AC FOR INTERRUPT HANDLER
PENDFL: .BLK    1                ;INTERRUPT PENDING FLAG FOR WAIT

        .NREL


; INITIATE I/O REQUEST

SAMPLE: JSR     @SAVE            ;SAVE REGISTERS, ALLOCATE FRAME
        FSIZE                    ;STACK FRAME SIZE
        1                        ;ONE PARAMETER
        SP+ADATA                 ;THE ARRAY SPECIFIER (TWO WORDS)
        ASPCF                    ;INTEGER PARAMETER ARRAY
        JSR     @SIZE            ;THE SIZE OF THE ARRAY (TOTAL NUMBER
        SP+ADATA                 ;OF WORDS ALLOCATED) IS THE NUMBER
        ASPCF                    ;OF WORDS TO TRANSFER, FILL THE
        SP+WDCNT                 ;ARRAY.  WORD COUNT SET.
        JSR     @HBOUND          ;HIGH BOUND OF THE FIRST DIMENSION
        SP+ADATA                 ;IS THE HIGHEST CHANNEL NUMBER
        ASPCF                    ;TO BE SCANNED.
        C1                       ;DIMENSION 1.
        SP+HIGHCH                ;SET HIGHEST CHANNEL NUMBER.
        LDA     Ø,PENDFL         ;IS THERE I/O IN PROGRESS?
        MOV     Ø,Ø,SZR          ;IF YES THEN WAIT FOR IT TO FINISH.
        JMP     .-2              ;YES, TRY AGAIN.
        INTDS                    ;DISABLE INTERRUPTS WHILE WE MESS
        LDA     Ø,1              ;AROUND WITH THE INTERRUPT LOCATION.
        STA     Ø,SYSINT         ;SAVE SYSTEM INTERRUPT ADDRESS.
        LDA     Ø,IADR           ;PROCEDURES'S INTERRUPT HANDLER.
        STA     Ø,1              ;INTERCEPT ALL INTERRUPTS.
        LDA     Ø,ADATA,3        ;POINTER TO ARRAY DATA.
        DOB     Ø,ADCV           ;LOAD ADCV CURRENT ADDRESS.
        LDA     Ø,WDCNT,3        ;NUMBER OF WORDS TO TRANSFER.
        DOC     Ø,ADCV           ;LOAD ADCV WORD COUNT.
        LDA     Ø,HIGHCH,3       ;HIGHEST CHANNEL TO SCAN.
        DOAP    Ø,ADCV           ;LOAD FINAL ANALOG CHANNEL AND START.
        ISZ     PENDFL           ;INTERRUPT IS NOW OUTSTANDING.
        INTEN                    ;WE BETTER LET THEM IN NOW.
        JSR     @RETURN          ;RETURN TO CALLER.
```

```
; WAIT FOR I/O TO COMPLETE

WAIT:   JSR     @SAVE                  ;SAVE REGISTERS, ALLOCATE FRAME
        FSIZE                          ;WE DON'T REALLY NEED THIS MUCH.
        0                              ;NO PARAMETERS, JUST WAIT.
        LDA     0,PENDFL               ;ARE THERE INTERRUPTS PENDING?
        MOV     0,0,SZR                ;IF I/O NOT FINISH THEN LOOP.
        JMP     .-2                    ;PICK UP FLAG AND TRY AGAIN.
        JSR     @RETURN                ;I/O IS DONE, RETURN TO CALLER.

; INTERRUPT HANDLER

INTRP:  SKPDN   ADCV                   ;IF NOT OUR INTERRUPT THEN LET
        JMP     @SYSINT                ;THE SYSTEM PROCESS IT.
        STA     0,SAVAC0               ;IT'S OURS, SAVE AN AC.
        LDA     0,SYSINT               ;RESTORE SYSTEM INTERRUPT HANDLER
        STA     0,1                    ;WE MAY NOT PASS THIS WAY AGAIN.
        SUBC    0,0                    ;SET INTERRUPT DONE INDICATOR.
        STA     0,PENDFL
        NIOC    ADCV                   ;CLEAR THE A/D INTERRUPT.
        LDA     0,SAVAC0               ;RESTORE THE SAVED AC.
        INTEN                          ;LET THE OTHERS IN.
        JMP     @0                     ;RETURN TO INTERRUPTED PROGRAM.

IADR:   INTRP                          ;ADDRESS OF INTERRUPT HANDLER
C1:     1                              ;CONSTANT FOR HBOUND

        .END
```

```
R
ADCON
AVERAGE: 3.607
AVERAGE: 3.609
AVERAGE: 3.611
AVERAGE: 3.604
AVERAGE: 3.607
AVERAGE: 3.616
AVERAGE: 3.609
AVERAGE: 3.614
AVERAGE: 3.607
AVERAGE: 3.609
AVERAGE: 3.611
AVERAGE: 3.604
AVERAGE: 3.607
AVERAGE: 3.616
AVERAGE: 3.609
AVERAGE: 3.614
AVERAGE: 3.607
AVERAGE: 3.609
AVERAGE: 3.611
AVERAGE: 3.604
AVERAGE: 3.607
AVERAGE: 3.616
AVERAGE: 3.609
AVERAGE: 3.614
AVERAGE: 3.607
AVERAGE: 3.609
AVERAGE: 3.611
AVERAGE: 3.604
AVERAGE: 3.607
```

The page following shows an alternative program for reading in single words from an A-D converter.

```
; BASIC A/D CONVERSION

; THIS PROCEDURE READS A SINGLE WORD FROM
; THE SPECIFIED CHANNEL RETURNING THE
; WORD AS THE VALUE OF THE FUNCTION.
;
; SEE 'HOW TO USE THE NOVA COMPUTERS',
; EDITION 4, PP. 6-4 TO 6-6.

; CALLING SEQUENCE:
;
;       I := ADCONV(CHANNEL);
;
;       WHERE I IS AN INTEGER AND ADCONV IS
;       DECLARED AS AN 'EXTERNAL INTEGER PROCEDURE'

        .TITL   ADCONV
        .ENT    ADCONV
        .EXTU

        .NREL

        T.INTEGER=1B11+1        ;TYPE SPECIFIER FOR INTEGER
        C.PARAMETER=2B7         ;STORAGE CLASS FOR PARAMETER
        C.VALUE=4B7             ;STORAGE CLASS FOR VALUE

        S=-167                  ;INITIAL VARIABLE DISPLACEMENT
        SP=1B0-S                ;PARAMETER DESCRIPTOR ADJUSTMENT

        CHANNEL=S               ;DISPLACEMENT FOR CHANNEL NUMBER
        RESULT=CHANNEL+1        ;DISPLACEMENT FOR FUNCTION VALUE
        FSIZE=RESULT-S+1        ;ASSIGNED FRAME SIZE

ADCONV: JSR     @SAVE           ;SAVE REGISTERS, ALLOCATE FRAME
        FSIZE                   ;NUMBER OF WORDS TO SAVE
        2                       ;TWO PARAMETERS INCLUDING RESULT
        SP+RESULT               ;RETURNED FUNCTION VALUE
        T.INTEGER+C.PARAMETER   ;INTEGER PARAMETER
        SP+CHANNEL              ;CHANNEL NUMBER FOR CONVERSION
        T.INTEGER+C.VALUE       ;INTEGER VALUE

        INTDS                   ;DISABLE INTERRUPTS DURING I/O
        LDA     0,CHANNEL,3     ;CHANNEL NUMBER => AC0
        DOAS    0,ADCV          ;LOAD CHANNEL SELECT AND START
        SKPDN   ADCV            ;WAIT FOR END OF CONVERSION
        JMP     .-1
        DICC    0,ADCV          ;READ DATA INTO AC0, CLEAR DONE
        INTEN                   ;INTERRUPTS ARE OK NOW
        STA     0,RESULT,3      ;STORE FUNCTION VALUE FOR RETURN
        JSR     @RETURN         ;RETURN TO CALLER

        .END
```

HELP is a natural language question-answering program designed to provide a computerized library reference service. The program is based on the HELP/QAS system developed at Project Genie, University of California at Berkeley. The program supplies interactive responses to requests for information in the form of keywords or sentences containing keywords. The program scans an input line for keywords and responds to a keyword with limited or more detailed information as desired.

Cross-referencing between keywords is implemented, so that additional information can be supplied. When information on a subject is exhausted, the program prints out a message to that effect.

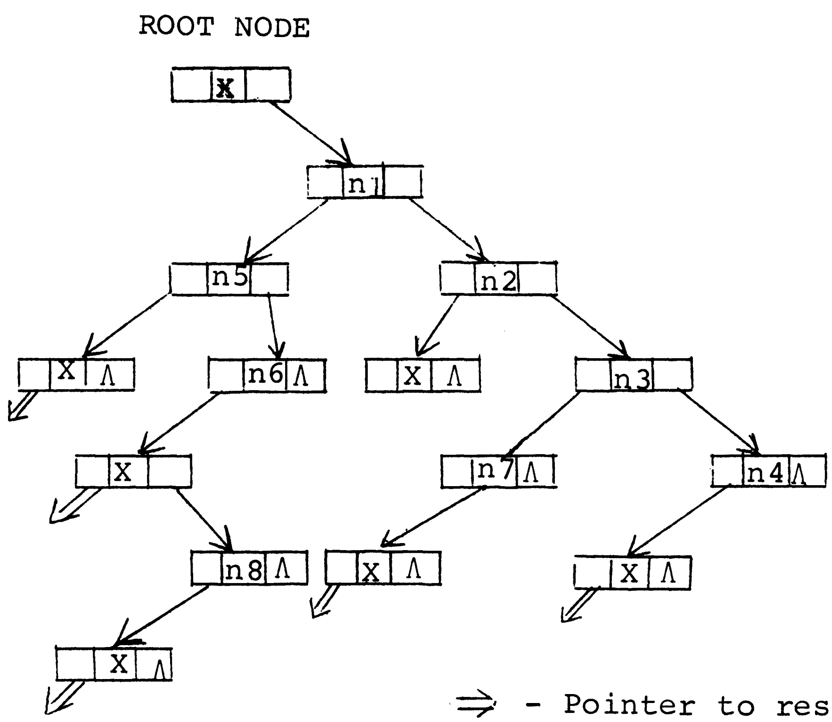
HELP was implemented in Data General ALGOL to show relatively sophisticated uses of several of the language features. Note, for example, how the hashing procedure HASH uses the functions length, ascii, and shift, together with logical operations, to develop in a relatively succinct code an integer hash value based on the original keyword string.

Use of string procedures and string manipulating facilities, in particular the substr function, are shown in some detail in the string procedure FETCH, which parses the question line. Note also how FETCH uses two boolean procedures, DELIMITER and TERMINATOR, to test for delimiters and terminators.

Essentially, when the user asks a question, the question line is parsed, and each word of the question is hash-encoded by forming a 16-bit value from the number of characters in the word and the first, middle, and last character. If this value is located in the hash table, the word is assumed to be a keyword and is added to the keyword list for the question. When all the words have been processed, redundant keywords are eliminated from the list and the list is sorted.

This sorted list of encoded keywords is passed to COMBINATIONS, a recursive procedure which builds a new keyword list for each possible combination of the original keyword list. Each of these new lists is passed to the procedure TREEMAINTAINER.

TREEMAINTAINER is used to locate and append branches to a binary tree structure representing the set of all known answers. Each node of the answer tree has three fields, a left link, a right link, and a value. The value is either an index into the keyword hash table or a flag indicating the end of the answer list. To obtain an answer, TREEMAINTAINER follows a path through the tree structure until an end node is encountered. The path is such that the first key in the answer may be reached by following right links. When that node is reached, its left link points to the chain of right links that leads to the second key in the answer, etc. The figure following shows how the binary tree structure is implemented.



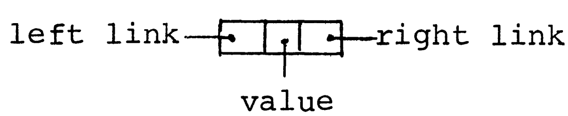
⇒ - Pointer to response

X - "End" flag set

Λ - End of thread

Paths: n1<n2<n3<n4
n1<n5<n6<n8
n3<n7

Defined Keyword Lists: {n1,n5},{n1,n6},{n1,n6,n8},
{n2},{n3,n7},{n4}



When an answer is found, the REPLY procedure positions the script file to the byte indicated for a response. A reference in this re- sponse of the form [name] will cause the procedure ANSWERPOINTER to be called to locate the byte position for answer "name" in its an- swer list, and REPLY will then be called recursively. The assembly language source code for ANSWERPOINTER is included at the end of th source code for HELP.

The HELP program is provided with a script which associates a re- sponse with one or more sets of keywords. Each entry in the scrip has as its first line a name or number for the response followed by zero or more keywords. Lines of the entry following the first line are responses for this set of keywords.

After a response has been used, it will be deleted, with the excep- tion of the last response in the entry, which will be reused. Each response will be either one line or several lines surrounded by quotes. The text will be formatted into 60-character lines on out- put. Formatting may be suppressed by enclosing part of a response in percent signs (%).

An entry may refer to other entries by enclosing the entry name in square brackets. An entire entry may be made a synonym for a pre- viously defined entry by providing the same entry name on the first line. In this case, the entry would be only one line long.

A sample script for HELP, containing information on Data General's software, is shown on following pages. Note cross-referencing; for example, a reference to LANGUAGES also references FORTRAN and ALGOL.

The user of HELP need not phrase his questions precisely. For ex- ample the queries:

OPEN FILE?

and

HOW DO I OPEN A FILE?

will produce equivalent responses.

To avoid repetitive responses, the script includes messages indica- ting the end of available information such as:

THE ARGUMENTS TO OPEN HAVE BEEN DESCRIBED EARLIER.

The ALGOL source tapes for HELP are available from Data General.

Sample HELP Teletype Session

```
R
HELP

> WHAT LANGUAGES ARE AVAILABLE
ALGOL AND FORTRAN ARE AVAILABLE FOR THE NOVA LINE OF MINICOMPUTERS.

> IS STAND ALONE OPERATION POSSIBLE?
STAND ALONE OPERATION IS SUPPORTED AS WELL AS DISK OPERATION.

> WHAT LANGUAGES ARE AVAILABLE FOR STAND ALONE?
BOTH FORTRAN AND ALGOL ARE AVAILABLE FOR STAND-ALONE USE.

> IS THE BASIC LANGUAGE AVAILABLE?
TIME SHARED BASIC IS AVAILABLE FOR STAND ALONE USE.

> ARE FILES ALLOWED IN BASIC?
FILES ARE NOT PRESENTLY ALLOWED IN BASIC

> CAN FILES BE USED AT ALL?
FILES CAN BE USED UNDER THE DISK OPERATING SYSTEM.

> HOW DO I OPEN A FILE UNDER DOS?
TO OPEN A FILE FROM DOS USE THE SYSTEM COMMAND '.OPEN'.  THE ARGUMENTS
ARE:

        AC0 - BYTE POINTER TO FILE NAME
        AC1 - FILE ATTRIBUTES TO BE OVERRIDDEN
        AC2 - ERROR NUMBER RETURNED

THE CHANNEL NUMBER IS IN THE COMMAND OR MAY BE 77 SIGNIFYING THAT
THE CHANNEL NUMBER IS IN AC2.

> HOW DO I OPEN A FILE FROM ALGOL?
TO OPEN A FILE FROM AN ALGOL PROGRAM USE THE CALL:

        OPEN (CHANNEL, FILENAME);


> HOW DO YOU OPEN A FILE?
TO OPEN A FILE FROM DOS USE THE SYSTEM COMMAND '.OPEN'.  THE ARGUMENTS
WERE DESCRIBED BEFORE.  FROM ALGOL USE THE 'OPEN' CALL.

> HOW DO I OPEN A FILE FROM ALO←GOL?
FROM ALGOL USE THE 'OPEN' CALL.
```

Sample HELP Script

```
12 OPEN FILE
[14][13]

13 OPEN FILE ALGOL
"TO OPEN A FILE FROM AN ALGOL PROGRAM USE THE CALL:%

        OPEN (CHANNEL, FILENAME);
%"
"FROM ALGOL USE THE 'OPEN' CALL.  "

14 OPEN FILE DOS
"TO OPEN A FILE FROM DOS USE THE SYSTEM COMMAND '.OPEN'.  [OA]"

OA
"THE ARGUMENTS ARE:%

        AC0 - BYTE POINTER TO FILE NAME
        AC1 - FILE ATTRIBUTES TO BE OVERRIDDEN
        AC2 - ERROR NUMBER RETURNED

%THE CHANNEL NUMBER IS IN THE COMMAND OR MAY BE 77 SIGNIFYING
THAT THE CHANNEL NUMBER IS IN AC2.  "
"THE ARGUMENTS WERE DESCRIBED BEFORE.  "

8 FILES
FILES CAN BE USED UNDER THE DISK OPERATING SYSTEM.

9 PLOTTER
"ALGOL AND FORTRAN PROVIDE A COMPREHENSIVE COLLECTION
OF PLOTTER ROUTINES."

1 FORTRAN
[DG] FORTRAN COMPILER IS A SUPERSET OF ANSI FORTRAN.
FORTRAN HAS BEEN MENTIONED EARLIER.

2 ALGOL
[DG] EXTENDED ALGOL COMPILER PROVIDES A SUPERSET OF THE ALGOL 60 LANGUA

3 LANGUAGES AVAILABLE
ALGOL AND FORTRAN ARE AVAILABLE FOR THE NOVA LINE OF MINICOMPUTERS.

6 LANGUAGES
[3] [1] [2]
```

```
4 STAND ALONE LANGUAGES
BOTH FORTRAN AND ALGOL ARE AVAILABLE FOR STAND-ALONE USE.

5 STAND ALONE
STAND ALONE OPERATION IS SUPPORTED AS WELL AS DISK OPERATION.

DG
DATA GENERAL'S

11 BASIC
"BASIC IS A FULL IMPLEMENTATION OF THE BASIC LANGUAGE DEVELOPED
AT DARTMOUTH COLLEGE"

7 BASIC FILES
FILES ARE NOT PRESENTLY ALLOWED IN BASIC

10 BASIC LANGUAGE
TIME SHARED BASIC IS AVAILABLE FOR STAND ALONE USE.
```

```
BEGIN

COMMENT:          ********************************
                  *                              *
                  *      *  * **** *    ***      *
                  *      *  * *    *    *  *     *
                  *      **** ***  *    ***      *
                  *      *  * *    *    *        *
                  *      *  * **** **** *        *
                  *                              *
                  *  A NATURAL LANGUAGE QUESTION *
                  *  ANSWERING PROGRAM BASED ON  *
                  *  THE HELP SYSTEM DEVELOPED AT *
                  *  PROJECT GENIE, UNIVERSITY OF *
                  *  CALIFORNIA, BERKELEY.       *
                  *                              *
                  ********************************
                                                    ;

LITERAL TABLESIZE (459);
INTEGER ARRAY TABLE[0:TABLESIZE];


INTEGER PROCEDURE HASH (S);

COMMENT:  THE HASH VALUE FOR A KEY WORD IS DEVELOPED
        BY PACKING THE LENGTH AND THE RADIX 16 VALUE
        OF THE FIRST, MIDDLE, AND LAST CHARACTERS OF
        THE WORD INTO AN UNSIGNED INTEGER;

        STRING S;

BEGIN   INTEGER I, J, K;

        J := K := LENGTH(S);

        FOR I := 1, (K+1)/2, K DO
           J := SHIFT(J, -4) OR ASCII(S,I) AND 17R8;

        HASH := ABS(J);
        END;


INTEGER PROCEDURE HASHINDEX (N);

COMMENT:  AN INDEX INTO THE HASH TABLE IS DETERMINED
        BY THE HASH VALUE FOR THE WORD MODULO THE
        SIZE OF THE HASH TABLE;

        INTEGER N;

        HASHINDEX := N-(N/TABLESIZE)*TABLESIZE;
```

```
EXTERNAL POINTER PROCEDURE ANSWERPOINTER;
EXTERNAL PROCEDURE TREEMAINTAINER;
EXTERNAL STRING PROCEDURE FETCH;
EXTERNAL PROCEDURE COMBINATIONS;
EXTERNAL PROCEDURE SORT;

INTEGER ARRAY KEYLIST[1:20];
INTEGER I, J, N, ANSWER, KEYS;
BASED INTEGER BI;
LITERAL CR ("<15>");
STRING (512) S;



COMMENT:        ********************************
                *                              *
                *      ANSWER DEFINITION       *
                *                              *
                ********************************
                                                    ;



        OPEN (1, "HELP.$$");
        FOR I := Ø STEP 1 UNTIL TABLESIZE DO
           TABLE[I] := Ø;

DEFINE: S := FETCH;
        IF S = CR THEN GO TO DEFINE;
        ANSWER := ANSWERPOINTER(S);
        KEYS := Ø;

1:      S := FETCH;
        IF S = CR THEN GO TO 2;
        IF S = "" THEN GO TO EOF;
        KEYS := KEYS+1;
        N := HASH(S);
        TABLE[HASHINDEX(N)] := N;
        KEYLIST[KEYS] := N;
        GO TO 1;

2:      IF KEYS = Ø THEN GO TO 3;
        SORT (KEYLIST, 1, KEYS);
        TREEMAINTAINER (KEYLIST, KEYS, ANSWER, TRUE);

3:      READ (1, S, EOF);
        IF S = "" THEN GO TO DEFINE;
        ANSWER->BI := ANSWER->BI + 1;
        GO TO 3;

EOF:    CLOSE (1);
        OPEN (1, "$TTI");
        OPEN (2, "$TTO");
        OPEN (3, "HELP.$$");
```

```
COMMENT:          *********************************
                  *                               *
                  *       MAIN PROGRAM LOOP       *
                  *                               *
                  *********************************
                                                     ;


PARSE:  WRITE (2, "<15>> ");
        S := FETCH;
        KEYS := 0;

4:      IF S = "?" THEN GO TO 6;
        IF S = CR THEN GO TO 7;
        N := HASH(S);
        IF TABLE[HASHINDEX(N)] = N THEN BEGIN

           FOR I := 1 STEP 1 UNTIL KEYS DO
           IF KEYLIST[I] = N THEN GO TO 5;
           KEYS := KEYS+1;
           KEYLIST[KEYS] := N;
           END;

5:      S := FETCH;
        GO TO 4;

6:      IF FETCH <> CR THEN GO TO 6;

7:      SORT (KEYLIST, 1, KEYS);
        FOR N := 0 STEP 1 UNTIL KEYS-1 DO BEGIN

           COMBINATIONS(KEYLIST, KEYS, KEYS-N, J);
           IF J > 0 THEN GO TO PARSE;
           END;

        WRITE (2, "I DON'T KNOW.<15>");
        GO TO PARSE;

        END
```

```
PROCEDURE TREEMAINTAINER (KEYLIST, KEYS, ANSWER, DEFINITION);

        INTEGER ARRAY KEYLIST;
        INTEGER KEYS, ANSWER;
        BOOLEAN DEFINITION;

COMMENT:        ***************************************
                *                                     *
                *  TREEMAINTAINER MAINTAINS A BINARY  *
                *  TREE STRUCTURE WITH EACH NODE      *
                *  CONSISTING OF THREE FIELDS, A LEFT *
                *  LINK, A RIGHT LINK, AND A VALUE.   *
                *  THE VALUE FIELD CAN BE EITHER AN   *
                *  INDEX INTO THE KEY WORD HASH TABLE *
                *  OR A FLAG INDICATING THE END OF AN *
                *  ANSWER LIST.  A COMPLETE PATH FROM *
                *  THE ROOT NODE TO AN END NODE IN    *
                *  THE STRUCTURE DESCRIBES THE SET OF *
                *  KEY WORDS WHICH WILL ELICIT A      *
                *  PARTICULAR ANSWER.                 *
                *                                     *
                ***************************************
                                                        ;

BEGIN   POINTER P, LINK, NEW;
        OWN POINTER ROOT;
        INTEGER KEY, NEXT;
        BASED POINTER BP;
        BASED INTEGER BI;
        LITERAL NULL(0), LEFT(1), RIGHT(2);

        LINK := ADDRESS(ROOT);
        P := ROOT;
        NEXT := 1;
        KEY := KEYLIST[NEXT];

COMMENT:  FOR AN ORDERED SET OF KEYS COMPRISING A VALID
        ANSWER LIST THERE EXISTS A PATH THROUGH THE TREE
        SUCH THAT THE FIRST KEY IN THE ANSWER MAY BE
        REACHED BY FOLLOWING RIGHT LINKS.  WHEN THIS NODE
        IS REACHED ITS LEFT LINK POINTS TO THE CHAIN OF
        RIGHT LINKS LEADING TO THE SECOND KEY IN THE
        ANSWER AND SO FORTH FOR EACH KEY IN THE LIST;

RIGHTSEARCH:
        IF P = NULL THEN GO TO NOTFOUND;
        IF P->BI = KEY THEN GO TO LEFTSEARCH;
        IF P->BI > KEY THEN GO TO NOTFOUND;
        LINK := P+RIGHT;
        P := LINK->BP;
        GO TO RIGHTSEARCH;
```

```
LEFTSEARCH:
        IF KEY = NULL THEN GO TO FOUND;
        LINK := P+LEFT;
        P := LINK->BP;
        NEXT := NEXT+1;
        KEY := IF NEXT =< KEYS THEN KEYLIST[NEXT] ELSE NULL;
        GO TO RIGHTSEARCH;

NOTFOUND:
        IF DEFINITION THEN BEGIN

           ALLOCATE (NEW, 3);
           NEW->BI := KEY;
           (NEW+RIGHT)->BP := P;
           LINK->BP := P := NEW;
           GO TO RIGHTSEARCH;
           END

        ELSE BEGIN

           ANSWER := Ø;
           GO TO DONE;
           END;

FOUND:  IF DEFINITION THEN (P+LEFT)->BI := ANSWER
        ELSE ANSWER := (P+LEFT)->BI;

DONE:   END


STRING PROCEDURE FETCH;

COMMENT:        ***********************************
                *                                 *
                *  FETCH PARSES THE INPUT STREAM  *
                *  INTO ENGLISH WORDS, NUMBERS,   *
                *  AND CERTAIN SPECIAL SYMBOLS.   *
                *                                 *
                ***********************************
                                                      ;


BEGIN   BOOLEAN PROCEDURE DELIMITER (C);
        STRING (1) C;
        DELIMITER := NOT((C >= "A") AND (C =< "Z")
           OR (C >= "Ø") AND (C =< "9") OR (C = "$"));

        BOOLEAN PROCEDURE TERMINATOR (C);
        STRING (1) C;
        TERMINATOR := (C = "?") OR (C = "<15>");
```

```
        STRING (1) C;
        STRING (12) S;
        OWN STRING (127) LINE;
        OWN INTEGER LINEINDEX;

1:      IF (LINEINDEX = 0) OR
           (LINEINDEX > LENGTH(LINE)) THEN BEGIN
           READ (1, LINE, EOF);
           SUBSTR(LINE, LENGTH(LINE)+1) := "<15>";
           LINEINDEX := 1;
           END;

        C := SUBSTR(LINE, LINEINDEX);
        IF NOT(DELIMITER(C)) THEN GO TO 2;
        LINEINDEX := LINEINDEX+1;
        IF TERMINATOR(C) THEN GO TO 3;
        GO TO 1;

2:      SUBSTR(S, LENGTH(S)+1) := C;
        LINEINDEX := LINEINDEX+1;
        C := SUBSTR(LINE, LINEINDEX);
        IF LINEINDEX > LENGTH(LINE) THEN GO TO 4;
        IF DELIMITER(C) THEN GO TO 4;
        GO TO 2;

3:      S := C;

4:      FETCH := S;
EOF:    END;
```

```
POINTER PROCEDURE ANSWERPOINTER (ANSWER);

        STRING ANSWER;

COMMENT:        ************************************
                *                                  *
                *  ANSWERPOINTER MAINTAINS A LIST  *
                *  OF NAMES OF ANSWER STRINGS AND  *
                *  THEIR ASSOCIATED POSITION IN    *
                *  THE SCRIPT FILE.  IF GIVEN THE  *
                *  NAME OF AN ANSWER THE PROGRAM   *
                *  WILL RETURN A POINTER TO A      *
                *  STRUCTURE ASSOCIATED WITH THE   *
                *  ANSWER.  IF THE NAME IS NOT     *
                *  FOUND A NEW ANSWER STRUCTURE    *
                *  WILL BE CREATED.                *
                *                                  *
                ************************************
                                                       ;

BEGIN   OWN POINTER ANSWERLIST;
        POINTER P;
        LITERAL NULL (0), ELMTSZ (6);
        LITERAL THREAD (1), FPOS (2), LGTH (3), NAME (4);
        EXTERNAL PROCEDURE FILEPOSITION;
        BASED INTEGER BI;
        BASED POINTER BP;
        BASED STRING (4) BS;

        P := ADDRESS(ANSWERLIST) - THREAD;

        FOR P := (P+THREAD)->BP WHILE P <> NULL DO

           IF SUBSTR((P+NAME)->BS, 1, (P+LGTH)->BI) = ANSWER
           THEN GO TO FOUND;

        ALLOCATE (P, ELMTSZ);
        (P+THREAD)->BP := ANSWERLIST;
        ANSWERLIST := P;
        (P+LGTH)->BI := LENGTH(ANSWER);
        (P+NAME)->BS := ANSWER;
        FILEPOSITION (1, (P+FPOS)->BI);

FOUND:  ANSWERPOINTER := P;
        END
```

```
PROCEDURE COMBINATIONS (A, N, M, ANSWER);

        VALUE N;
        INTEGER ARRAY A;
        INTEGER N, M, ANSWER;

COMMENT:        ******************************************
                *                                        *
                *   SEARCH THE ANSWER LIST HIERARCHY FOR *
                *   THE LONGEST SUB-LIST OF THE KEY WORD *
                *   LIST.  RETURN THE RECORD NUMBER FOR  *
                *   AN ANSWER OR ZERO IF NONE IS FOUND.  *
                *   EACH CALL FROM THE MAIN PROGRAM WILL *
                *   PRODUCE REPLIES FOR ALL N-ELEMENT    *
                *   SUB-LISTS FOUND IN THE HIERARCHY.    *
                *                                        *
                ******************************************


BEGIN   INTEGER ARRAY B[1:N];
        INTEGER I, J, K;
        EXTERNAL PROCEDURE REPLY, TREEMAINTAINER;

        IF N = M THEN BEGIN

           TREEMAINTAINER (A, N, ANSWER, FALSE);
           IF ANSWER > Ø THEN REPLY (ANSWER);
           END

        ELSE BEGIN

           ANSWER := Ø;
           FOR J := 1 STEP 1 UNTIL N DO BEGIN

              FOR I := 1 STEP 1 UNTIL N DO
                 B[IF I>J THEN I-1 ELSE I] := A[I];

              COMBINATIONS (B, N-1, M, K);
              IF K <> Ø THEN ANSWER := K,
              END;

           END;

        END;
```

```
PROCEDURE REPLY (AP);

        POINTER AP;

COMMENT:        ****************************************
                *                                      *
                *  REPLY FORMATS AND PRINTS A SPECIFIED *
                *  ANSWER.  REFERENCES IN THE ANSWER OF *
                *  THE FORM [NAME] WILL CAUSE REPLY TO  *
                *  BE CALLED RECURSIVELY FOR "NAME".    *
                *                                      *
                ****************************************
                                                          ;

BEGIN   EXTERNAL POINTER PROCEDURE ANSWERPOINTER;
        EXTERNAL PROCEDURE FILEPOSITION;
        LITERAL CR ("<15>"), FPOS (2);
        OWN INTEGER LEVEL;
        OWN STRING (80) LOUT;
        BASED INTEGER BI;
        STRING (512) S;
        STRING (4) T;
        STRING (1) C;
        INTEGER I;

        POSITION (3, (AP+FPOS)->BI);
        READ (3, S);
        IF AP->BI > 1 THEN BEGIN

           AP->BI := AP->BI - 1;
           FILEPOSITION (3, (AP+FPOS)->BI);
           END;

        LEVEL := LEVEL+1;
        FOR I := 1, I+1 WHILE I <= LENGTH(S) DO BEGIN

           T := "";
           C := SUBSTR(S, I);
           IF C = "[" THEN BEGIN

              S := SUBSTR(S, I+1, LENGTH(S));
              T := SUBSTR(S, 1, INDEX(S, "]")-1);
              I := LENGTH(T)+1;
              REPLY (ANSWERPOINTER(T));
              END
```

```
    ELSE IF C = "%" THEN BEGIN

       WRITE (2, LOUT);
       LOUT := "";
       S := SUBSTR(S, I+1, LENGTH(S));
       I := INDEX(S, "%");
       WRITE (2, SUBSTR(S, 1, I-1));
       END

    ELSE IF (C = " ") AND LENGTH(LOUT) > 60
       THEN BEGIN

       WRITE (2, LOUT, CR);
       LOUT := "";
       END

    ELSE IF C = CR THEN
       SUBSTR(LOUT, LENGTH(LOUT)+1) := " "

    ELSE SUBSTR(LOUT, LENGTH(LOUT)+1) := C;

    END;

 LEVEL := LEVEL-1;
 IF LEVEL = 0 THEN BEGIN

    WRITE (2, LOUT, CR);
    LOUT := "";
    END;

 END
```

```
PROCEDURE SORT (A, N, M);

        INTEGER ARRAY A;
        INTEGER N, M;

COMMENT:        ****************************
                *                          *
                *  A SIMPLE BUBBLE SORT    *
                *  TO ORDER THE RELATIVELY *
                *  SHORT KEY WORD LISTS.   *
                *                          *
                ****************************
                                              ;

BEGIN   INTEGER I, T;
        BOOLEAN DONE;

BUBBLE: DONE := TRUE;

        FOR I := N STEP 1 UNTIL M-1 DO
           IF A[I] > A[I+1] THEN BEGIN

              T := A[I];
              A[I] := A[I+1];
              A[I+1] := T;
              DONE := FALSE;
              END;

        IF NOT DONE THEN GO TO BUBBLE;

        END;
```

```
              ; POINTER PROCEDURE ANSWERPOINTER (ANSWER);

                      .TITL   ANSWERPOINTER

                      .EXTU
                      .ENT    ANSWERPOINTER
                      .EXTN   ALLOCATE
                      .EXTN   FILEPOSITION

                      .ZREL
00000-000401'.LP:     LP+200
      000001^OP:      .BLK    1

                      .NREL
      000001          .TXTM   1

              ANSWERPOINTER:
00000'006011$         JSR     @SAVE
00001'000014          FS0
00002'000402          1B7+2
00003'100000          SP+0                    ;ANSWERPOINTER
00004'003541          003541                  ;POINTER PARAMETER
00005'100001          SP+1                    ;ANSWER
00006'001203          001203                  ;STRING PARAMETER

              ;
              ;       STRING ANSWER;
              ;
              ; COMMENT:      ************************************
              ;               *                                  *
              ;               *  ANSWERPOINTER MAINTAINS A LIST  *
              ;               *  OF NAMES OF ANSWER STRINGS AND  *
              ;               *  THEIR ASSOCIATED POSITION IN    *
              ;               *  THE SCRIPT FILE.  IF GIVEN THE  *
              ;               *  NAME OF AN ANSWER THE PROGRAM   *
              ;               *  WILL RETURN A POINTER TO A      *
              ;               *  STRUCTURE ASSOCIATED WITH THE   *
              ;               *  ANSWER.  IF THE NAME IS NOT     *
              ;               *  FOUND A NEW ANSWER STRUCTURE    *
              ;               *  WILL BE CREATED.                *
              ;               *                                  *
              ;               ************************************
              ;                                                     ;
              ;
              ; BEGIN OWN POINTER ANSWERLIST;
              ;       POINTER P;
              ;       LITERAL NULL (0), ELMTSZ (6);
              ;       LITERAL THREAD (1), FPOS (2), LGTH (3),
              ;               NAME (4);
              ;       EXTERNAL PROCEDURE FILEPOSITION;
              ;       BASED INTEGER BI;
              ;       BASED POINTER BP;
              ;       BASED STRING (4) BS;
              ;
              ;       P := ADDRESS(ANSWERLIST) - THREAD;

00007'006003$         JSR     @BLKSTART
```

```
00010'006001$          JSR      @ADDRESS
00011'000001-          OP+0                        ;ANSWERLIST
00012'000541           000541                      ;POINTER OWN
00013'100007           SP+7                        ;TEMPORARY
00014'021620           LDA      0,S+7,3            ;TEMPORARY
00015'100400           NEG      0,0
00016'100000           COM      0,0
00017'041615           STA      0,S+4,3            ;P

            ;
            ;          FOR P := (P+THREAD)->BP WHILE P <> NULL DO

00020'021615 .G3:      LDA      0,S+4,3            ;P
00021'101400           INC      0,0
00022'111000           MOV      0,2
00023'021000           LDA      0,0,2
00024'041615           STA      0,S+4,3            ;P
00025'126400           SUB      1,1                ;(0)
00026'106404           SUB      0,1,SZR
00027'126520           SUBZL    1,1
00030'125005           MOV      1,1,SNR
00031'000406           JMP      .G4
00032'006401           JSR      @.+1
00033'000041'          .G2
00034'034014$          LDA      3,.SP
00035'002401           JMP      @.+1
00036'000020'          .G3
00037'002401 .G4:      JMP      @.+1
00040'000111'          .G1
00041'165400 .G2:      INC      3,1                ;SAVE

            ;
            ;             IF SUBSTR((P+NAME)->BS, 1, (P+LGTH)->BI)
            ;                   = ANSWER

00042'034014$          LDA      3,.SP
00043'021615           LDA      0,S+4,3            ;P
00044'034000-          LDA      3,.LP
00045'031605           LDA      2,L+5,3            ;LITERAL
00046'113000           ADD      0,2
00047'034014$          LDA      3,.SP
00050'051620           STA      2,S+7,3            ;TEMPORARY
00051'006004$          JSR      @BSSTR
00052'100007           SP+7                        ;TEMPORARY
00053'000021           000021                      ;INTEGER
00054'000004           4
00055'100010           SP+10                       ;TEMPORARY
00056'034000-          LDA      3,.LP
00057'031604           LDA      2,L+4,3            ;LITERAL
00060'113000           ADD      0,2
00061'031000           LDA      2,0,2
00062'034014$          LDA      3,.SP
00063'051620           STA      2,S+7,3            ;TEMPORARY
00064'006013$          JSR      @SUBSTR
00065'000004           4
00066'100010           SP+10                       ;TEMPORARY
00067'000202           000202                      ;STRING
```

```
00070'000203'          LP+2                        ;LITERAL
00071'100007           SP+7                        ;TEMPORARY
00072'100012           SP+12                       ;TEMPORARY

             ;            THEN GO TO FOUND;

00073'006012$          JSR      @STREQ
00074'100012           SP+12                       ;TEMPORARY
00075'000202           000202                      ;STRING
00076'100001           SP+1                        ;ANSWER
00077'001203           001203                      ;STRING PARAMETER
00100'152560           SUBCL    2,2
00101'045620           STA      1,S+7,3            ;TEMPORARY
00102'151015           MOV#     2,2,SNR
00103'002402           JMP      @.+2
00104'000402           JMP      .+2
00105'000110'          .G5
00106'002401           JMP      @.+1
00107'000175'          .L1
00110'003620 .G5:      JMP      @S+7,3

             ;
             ;         ALLOCATE (P, ELMTSZ);

00111'006005$.G1:      JSR      @CALL
00112'177777           ALLOCATE
00113'000002           2
00114'100004           SP+4                        ;P
00115'000141           000141                      ;POINTER LOCAL
00116'000202'          LP+1                        ;LITERAL
00117'000021           000021                      ;INTEGER

             ;         (P+THREAD)->BP := ANSWERLIST;

00120'021615           LDA      0,S+4,3            ;P
00121'101400           INC      0,0
00122'111000           MOV      0,2
00123'020001-          LDA      0,OP+0             ;ANSWERLIST
00124'041000           STA      0,0,2

             ;         ANSWERLIST := P;

00125'021615           LDA      0,S+4,3            ;P
00126'040001-          STA      0,OP+0             ;ANSWERLIST

             ;         (P+LGTH)->BI := LENGTH(ANSWER);

00127'025615           LDA      1,S+4,3            ;P
00130'034000-          LDA      3,.LP
00131'031604           LDA      2,L+4,3            ;LITERAL
00132'133000           ADD      1,2
00133'006006$          JSR      @LENGTH
00134'100001           SP+1                        ;ANSWER
00135'001203           001203                      ;STRING PARAMETER
00136'100007           SP+7                        ;TEMPORARY
00137'021620           LDA      0,S+7,3            ;TEMPORARY
```

```
00140'041000            STA     0,0,2

             ;          (P+NAME)->BS := ANSWER;

00141'034000-           LDA     3,.LP
00142'021605            LDA     0,L+5,3            ;LITERAL
00143'123000            ADD     1,0
00144'034014$           LDA     3,.SP
00145'041620            STA     0,S+7,3            ;TEMPORARY
00146'006004$           JSR     @BSSTR
00147'100007            SP+7                       ;TEMPORARY
00150'000021            000021                     ;INTEGER
00151'000004            4
00152'100012            SP+12                      ;TEMPORARY
00153'006007$           JSR     @MOVSTR
00154'100001            SP+1                       ;ANSWER
00155'001203            001203                     ;STRING PARAMETER
00156'100012            SP+12                      ;TEMPORARY
00157'000202            000202                     ;STRING

             ;          FILEPOSITION (1, (P+FPOS)->BI);

00160'034000-           LDA     3,.LP
00161'021603            LDA     0,L+3,3            ;LITERAL
00162'123000            ADD     1,0
00163'111000            MOV     0,2
00164'034014$           LDA     3,.SP
00165'051620            STA     2,S+7,3            ;TEMPORARY
00166'006005$           JSR     @CALL
00167'177777            FILEPOSITION
00170'000002            2
00171'000203'           LP+2                       ;LITERAL
00172'000021            000021                     ;INTEGER
00173'100007            SP+7                       ;TEMPORARY
00174'100021            @000021                    ;INTEGER

             ;
             ; FOUND:           ANSWERPOINTER := P;

00175'021615 .L1:       LDA     0,S+4,3            ;P
00176'041611            STA     0,S+0,3            ;ANSWERPOINTER

             ;          END

00177'006002$           JSR     @BLKEND
00200'006010$           JSR     @RETURN
      000014 FS0=       14
```

```
        177600 L=        -200
        177611  S=       -167
        100000  SP=      1B0

 00201'000000 LP:        000000
 00202'000006            000006
 00203'000001            000001
 00204'000002            000002
 00205'000003            000003
 00206'000004            000004

                         .END



  0007   ANSWE
ADDRE    000001$X
ALLOC    000112'X
ANSWE    000000'
BLKEN    000002$X
BLKST    000003$X
BSSTR    000004$X
CALL     000005$X
FILEP    000167'X
FS0      000014
L        177600
LENGT    000006$X
LP       000201'
MOVST    000007$X
OP       000001-
RETUR    000010$X
S        177611
SAVE     000011$X
SP       100000
STREQ    000012$X
SUBST    000013$X
.G1      000111'
.G2      000041'
.G3      000020'
.G4      000037'
.G5      000110'
.L1      000175'
.LP      000000-
.SP      000014$X
R
```

* * *

APPENDIX G

DEBUGGING ALGOL PROGRAMS

This appendix describes, through the use of examples, correcting compilation errors and debugging ALGOL programs with the Symbolic Debugger and the TRACE program. The operating environment used for all examples is RDOS.

CORRECTING COMPILATION ERRORS

Compilation errors are detected by the ALGOL compiler, which gives explicit error messages and the source statement in which the error was detected. (Refer to Chapter 10 for illustrations of compiler error messages.) These errors can be corrected using the Text Editor. The following example shows errors detected at compilation and illustrates the use of the Text Editor for correcting these errors. For details on the operation of the Text Editor, refer to Text Editor Manual, document number 093-000018.

The procedures shown in this example are general in nature and can be used to correct any source program. The procedures are:

1. Obtain a copy of the source programs (TEST.AL and SORT.AL) on the program console using the RDOS command TYPE. This command is issued here to give the programmer an accurate copy of the source programs before compilation.

2. Compile the programs with the ALGOL command. Source errors in TEST and SORT are printed on the console.

3. Call the Text Editor with the EDIT command to correct these errors. The Text Editor commands

   GWTEST1.AL$

   and

   GWSORT1.AL$

   write the corrected files into TEST1 and SORT1, respectively. These file names must be used when the programs are recompiled.

4. Use the ALGOL command again to compile the corrected programs, TEST1 and SORT1. The /U global switch includes user symbols, required for debugging, in the output. Both programs compile without error.

CORRECTING COMPILATION ERRORS (Continued)

```
TYPE TEST.AL                                ←RDOS command to type TEST.

BEGIN   INTEGER I;
        INTEGER ARRAY A[0:10];
        EXTERNAL PROCEDURE SORT;

        OPEN (1, "$TTI");
        OPEN (2, "$TTO");

        FOR I :=  0 UNTIL 10 STEP 1 DO BEGIN
           WRITE (2, "> ");
           READ (1, A[I]);
           END;

        WRITE (2, "SORT<15>");
        SORT (A);
        WRITE (2, "END SORT<15>", A, "<15>");

        END
```

Source program for TEST as written.

```
R
TYPE SORT.AL                                ←RDOS command to type SORT.

PROCEDURE SORT (A); INTEGER ARRAY A;

BEGIN   INTEGER I;
        BOOLEAN DONE;

BUBBLE:

DONE := TRUE;
FOR I := LBOUND(A,1) STEP 1 UNTIL HBOUND(A,1)-1 DO

   IF A[I] > A[I+1] THEN DO BEGIN

        T := A[I];
        A[I] := A[I+1];
        A[I+1] := T;
        DONE := FALSE;
        END;

IF NOT DONE THEN GO TO BUBBLE;

END
```

Source program for SORT as written.

CORRECTING COMPILATION ERRORS (continued)

```
ALGOL/U TEST; ALGOL/U SORT                        ← Compile TEST and SORT
        FOR I :=  Ø UNTIL 1Ø STEP 1 DO BEGIN
                  ↑
*** 'UNTIL' MUST FOLLOW 'STEP' ***                ← Source error in TEST

PROGRAM IS RELOCATABLE
   IF A[I] > A[I+1] THEN DO BEGIN
                          ↑
*** STATEMENT DOES NOT END PROPERLY ***
        A[I+1] := T;                              ← Source errors in SORT
                  ↑
*** UNDEFINED VARIABLE ***

PROGRAM IS RELOCATABLE
                          .TITL   SORT
R
EDIT                                              ← User enters Editor.

*GRTEST.AL$GWTEST1.AL$YS$
*CUNTIL 1Ø STEP 1 $STEP 1 UNTIL 1Ø $$
*PGC$$
*GRSORT.AL$GWSORT1.AL$YS$                           User edits source
*SINTEGER I$I, T$$                                  program, correcting
*L1T$$                                              errors and exits the
BEGIN   INTEGER I, T;                               Editor.
*CTHEN DO$THEN$$
*PGCH$$
R
ALGOL TEST1;ALGOL/U SORT1
                  ↑
PROGRAM IS RELOCATABLE                            U means "compile with
                                                  symbol output". Sym-
PROGRAM IS RELOCATABLE                            bols needed for debug-
                          .TITL   SORT            ging.
```

DEBUGGING USING THE SYMBOLIC DEBUGGER

Once compilation errors have been corrected in an ALGOL program, the Symbolic Debugger can be used to detect run-time errors.

Loading the Symbolic Debugger

To debug an ALGOL program, the following programs must be loaded together with the RLDR command:

1. The Symbolic Debugger, which is loaded automatically when the command line contains the /D global switch.

2. The ALGOL programs to be debugged.

Loading the Symbolic Debugger (Continued)

3. User symbols, output during compilation, which are loaded with the /U local switch.

4. The contents of the ALGOL library, LIBRARY.CM, loaded with the @ indirect convention, which brings in all necessary ALGOL run-time routines.

5. The file to contain the output save file, which is loaded using the /S local switch.

The format of the RLDR command, then, is:

```
RLDR/D inputfilename-1/U [...inputfilename-n] @LIBRARY.CM@
       savefilename/S )
```

RLDR/D inputfilename-1/U [...inputfilename-n] @LIBRARY.CM@ savefilen

The sample programs used in the explanation of correcting compilation errors and the debugger can be loaded with the command:

```
R                                            /D loads the debugger.
RLDR/D TEST1 SORT1/U @LIBRARY.CM@ SORT/S
 .MAIN                                       Output save file with
 SORT                                                     symbols.
                                             Loads the library.
R
```

The save file is named SORT. This name must be used in the DEB command, which starts the debugger.

Operating the Symbolic Debugger

General debugging procedures are described below; for additional information on the debugger, the user is referred to the Symbolic Debugger User's Manual, document number 093-000044. The sample programs described previously are again used to illustrate the use of the Symbolic Debugger. A listing of the source file, obtained at compilation with the /L switch, is used as an aid in debugging.

To understand the use of the debugger in this example, the user must first know the functions to be performed by the procedures. Basically, the TEST procedure performs all I/O. TEST reads values input at the console into array A. It first types a > and waits for the user to type a value, followed by a carriage return. When eleven values have been requested and input, TEST types the word SORT, followed by a carriage return. It then calls in the SORT procedure which performs a bubble sort of the eleven values in array A. Values are sorted so that the smallest is the first value in the array. When sorting is complete, control returns to TEST, which prints END SORT, followed by a carriage return and the contents of the array, each value of which is terminated by a carriage return.

The following command begins debugger execution:

DEB savefilename)

The user must then enable all local and global symbols for a procedure with the debugger command:

procedure-name %

where procedure-name is the name of a procedure. This procedure must have been compiled separately.

User commands to the debugger at this point will depend on the program being debugged. (To follow this example of debugging, the user should refer to the actual debugging session and the partial listing of SORT, shown on the next pages.) For this example, the next two commands open the location .L1, print its contents, and set a breakpoint at that location. A listing of SORT indicates the locations where breakpoints should be set to halt the debugger at critical points in execution. .L1 is the first location in SORT. Note that if a breakpoint is set on a JSR instruction, the breakpoint must be deleted before execution proceeds.

The main program is started with the $R command. When eleven values have been requested and input, control is passed to SORT. Because a breakpoint is set on the first instruction in SORT, execution stops and the location of the breakpoint and the contents of the accumulators are printed by the debugger.

The $= command instructs the debugger to print all output in numeric (octal) format. The next two commands open the locations .FP and .SP, the frame and stack pointers, and print their contents.

To understand the next command, refer to the listing of SORT. The location immediately following the JSR call to the run-time routine LBOUND contains a pointer to the first data in the array. (This can be verified by checking the calling sequence for LBOUND, described in Appendix C under the heading "General Purpose Routines.") The command S+Ø+16146/ opens the location containing the address of array A.

Operating the Symbolic Debugger (Continued)

This command illustrates a useful format for determining the address of data on the stack:

S+offset+C(.SP)

where: S is the initial stack offset.

offset is an integer offset from S into the stack.

C(.SP) is the contents of the stack pointer. (In the listing, the stack plus offset are expressed as SP+Ø.)

The command S+Ø+16146/ shows the first value of the array is in location 15731. That location is then opened with "/" to reveal a 7, which is the first data input to the array. Successive line feeds open the next five locations and print their contents.

Satisfied that the array contains the correct data, the user sets a breakpoint at location .G4. This location contains a JSR indirect instruction to the run-time routine HBOUND. By setting a breakpoint here, the user can examine the lower bound of array A before the upper (high) bound is computed. The Proceed command ($P) continues execution until this breakpoint is encountered. The location of the breakpoint and the contents of the four accumulators are printed automatically.

The next command opens the location containing I, the subscripting index, and shows its value to be Ø. Because I is an integer, the contents of the location S+offset+C(.SP) is the actual data in the location, not a pointer to the data. (Refer to Appendix B for a description of storage of the various data types on the stack.) The command S+5+16146/ opens the location for temporary storage, which is the result from LBOUND, and prints its contents, also zero.

Before execution can proceed, the breakpoint on the JSR @HBOUND instruction must be deleted. This is accomplished with the command 6$D.

The next command to the debugger examines the contents of .G4+5. The contents of this location shows the assembled data value and the instruction LDA Ø -162 3. (The value of the initial stack offset S is -167; S+5, as displayed in the listing, is -167+5.) To check that the high bound of array A is correct, the user sets a breakpoint at location .G4+5. Execution proceeds until the breakpoint is encountered. The location S+5+16146 is examined for the value of the high bound. It is shown to be 12; thus, the bounds of the array are correct (0 to 11).

Operating the Symbolic Debugger (Continued)

$P continues execution until breakpoint 7 is encountered. At this time, some data values should have been sorted. The user checks the locations 15731 through 15734 to verify that the sort is proceeding. The program is then run two more times (3$P) before the breakpoint stops execution. A spot-check of the same four locations shows sorting continues correctly. All breakpoints are deleted with $D and the program is allowed to run normally. The outputted data confirms that SORT and TEST have executed without error.

```
R
DEB SORT                   ← Command to start in debugger.

SORT%                      ← User symbols made known to debugger.
.L1/LDA 3 .LP              ← Open location .L1 and print its contents.
.$B                        ← Set breakpoint at .L1 in SORT.
$R                         ← Start program.
> 7         ⎫
> -2        ⎪
> 1Ø        ⎪
> 7777R8    ⎪
> 3.5       ⎬              ← Data read by main program.
> -4        ⎪
> 274       ⎪
> 2         ⎪
> Ø         ⎪
> 11        ⎪
> 34ØØ      ⎭
SORT                       ← Printed by main program.

7B .L1                     ← Breakpoint 7 encountered.
Ø ØØØØ13  1 177776  2 177777  3 Ø16146   ← Contents of accumulators
$= ←——————Numeric (octal) mode.            printed by debugger.
.FP/Ø16146 ⎫ ←———————————————————— Check stack pointers.
.SP/Ø16146 ⎭ ←————————————————————1st data word (S+0 is array A)
S+Ø+16146/Ø15731 /ØØØØØ7 ←———————      7
15732    177776 ←———————————————      -2
15733    ØØØØ12 ←———————————————      10
15734    ØØ7777 ←———————————————    7777R8
15735    ØØØØØ3 ←———————————————      3 (truncated 3.5)
15736    177774 ←———————————————      -4
.G4$B ←———————————————— Set breakpoint on  JSR  @HBOUND (at .G4)
$P
6B .G4
Ø ØØØØØØ  1 177776  2 ØØ6411  3 Ø16146
S+2+16146/ØØØØØØ   ←———————————————— S+2 is I
S+5+16146/ØØØØØØ   ←———————————————— S+5 is temporary
6$D ←————————————————— Must delete breakpoint on  JSR  @HBOUND
```

Operating the Symbolic Debugger (continued)

```
.G4+5/Ø21616 ;LDA Ø -162 3
.$B              <------------   Set breakpoint at .G4+5
$P
6B .G4+5
Ø ØØØØØØ  1 177776  2 ØØ6335  3 Ø16146
6$D
S+5+16146/ØØØØ12
$P
7B .L1
Ø ØØØØØ1  1 ØØØØ12  2 ØØ6335  3 Ø16146
15731/177776     <------------   -2
15732   ØØØØØ7
15733   ØØØØ12
15734   ØØØØØ3
3$P              <------------   Break before 3rd time executing .L1
7B .L1
Ø ØØØØØ1  1 ØØØØ12  2 ØØ6335  3 Ø16146
15731/177776     <------------   -2
15732   177774   <------------   -4
15733   ØØØØØ3
15734   ØØØØØ2
$D               <------------   Delete all breakpoints.
$PEND SORT
-4
-2
Ø
2
3
7
1Ø
11
274
34ØØ
4Ø95
R
```

Operating the Symbolic Debugger (continued)

The following portions of the SORT listing show coding where breakpoints were set on .L1, .G4, and .G4+5.

```
                    ; DONE := TRUE;

 00006'034000- .L1:     LDA      3,.LP
 00007'021600           LDA      0,L+0,3               ;LITERAL
 00010'034010$          LDA      3,.SP
 00011'041615           STA      0,S+4,3               ;DONE
```

```
          ; FOR I := LBOUND(A,1) STEP 1 UNTIL HBOUND(A,1)-1 DO

00012'006004$ .G3:    JSR       @LBOUND
00013'100000          SP+0                         ;A
00014'011021          011021                       ;INTEGER PARAMETER ARRAY
00015'000157'         LP+1                         ;LITERAL
00016'100005          SP+5                         ;TEMPORARY
00017'021616          LDA       0,S+5,3            ;TEMPORARY
00020'041613          STA       0,S+2,3            ;I
```

```
 SORT

 00021'006003$ .G4:    JSR       @HBOUND
 00022'100000          SP+0                         ;A
 00023'011021          011021                       ;INTEGER PARAMETER ARRAY
 00024'000157'         LP+1                         ;LITERAL
 00025'100005          SP+5                         ;TEMPORARY
 00026'021616          LDA       0,S+5,3            ;TEMPORARY
 00027'100400          NEG       0,0
 00030'100000          COM       0,0
 00031'025613          LDA       1,S+2,3            ;I
 00032'122500          SUBL      1,0
 00033'101002          MOV       0,0,SZC
```

DEBUGGING USING THE TRACE PROGRAM

Program debugging can use the TRACE procedure rather than the Symbolic Debugger. TRACE gives the user a complete picture of user stack frames created at run-time.

TRACE is supplied as an RDOS dump tape. It can be called either at the console or in a user program.

Calling TRACE at the Console

TRACE can be called at the console to be used only with a break file. The break file must be created before TRACE is called. During execution of an ALGOL program, the break file, BREAK.SV, can be created in either of two ways:

- When a fatal run-time error occurs, or
- When the user generates a console break by issuing the CTRL C combination.

The user can then trace the execution in the break file by issuing the command:

```
TRACE [filename] )
```

where: filename is the name of the break file, if the break file has been renamed. If the break file has not been renamed, this argument should be omitted.

The /L global switch can be appended to TRACE to indicate output is directed to the line printer. If this switch is omitted, output is directed to the console.

Calling TRACE in an ALGOL Program

An ALGOL program can call the TRACE procedure in either of two ways:

1. By declaring TRACE an external procedure and calling TRACE when an error is encountered. In this case, TRACE begins execution when it is called. The following example includes a call to TRACE in an ALGOL program.

Calling TRACE in an ALGOL Program (Continued)

```
PROG:   BEGIN INTEGER I;
        REAL (3) X;
        STRING (1Ø) S;
        INTEGER ARRAY IA[1:1Ø];
        EXTERNAL PROCEDURE TRACE;      ←declare an external pro-
                                        cedure.
        X := .333333R8;
        FOR I := 1 STEP 1 UNTIL 1Ø DO
                IA[I] := -1;


        OPEN (1, "$TTI");
        READ (1, S);
        OPEN (1, S, ERR1);             ←label to transfer to on
                                        occurrence of error.
        FOR I := 1 STEP 1 UNTIL 1Ø DO
                READ (1, IA[I]);

ERR1:   TRACE;                         ←on error, call

        END
```

2. By declaring ONTRACE and OFFTRACE external procedures and calling these procedures to make TRACE available or unavailable to the procedure. At any time after a call to ONTRACE is executed, the user can bring in the TRACE program by causing a break (CTRL C) at the console. OFFTRACE can be called later in the program to turn off the availability of the TRACE program. The following example illustrates the use of ONTRACE and OFFTRACE to trace parts of an ALGOL program.

Calling TRACE in an ALGOL Program (continued)

```
prog:  begin integer i;
       real (3) x;
       string (10) s;
       integer array  A[1:10];
       external procedure ontrace, offtrace;

       x := .333333r8;
       for i := 1 step 1 until 10 do
                A[i] := -1;

       ontrace;             ← turn on TRACE
       open (1, "$TTI");
       read (1,S);
       open (1,S);
       offtrace;            ← turn off TRACE while reading.
       for i := 1 step 1 until  10 do
               read(1, A[i]);
       ontrace;             ← turn TRACE on again.
          .
          .
          .
```

Debugging Aids for Use with TRACE

To understand the stack information printed by TRACE, the user should obtain the following:

- A full listing of the input source files, output at compilation with the /L switch.
- A load map, containing a list of symbols in numeric order, output at load time with the /L switch.

In addition, the user can load the Symbolic Debugger (with the /D global switch) with the programs to be traced. The debugger can be used in conjunction with TRACE to aid in debugging.

Loading Programs for Use with TRACE

To debug an ALGOL program with TRACE, the following programs should be loaded together with the RLDR command:

1. Symbolic Debugger, optionally loaded if the command line contains the /D global switch.
2. The ALGOL programs to be debugged.

Loading Programs for Use with TRACE (Continued)

3. User symbols output during compilation, which are loaded with the /U local switch.

4. The contents of the ALGOL library, LIBRARY.CM, loaded with the @ indirect convention, which brings in all necessary ALGOL run-time routines.

5. The file to contain the output save file (if it is to be different from the first input file name), which is loaded using the /S local switch.

6. A listing of symbols output with the /L switch.

The format of the RLDR command, then, is:

```
RLDR/D inputfilename-1/U [...inputname-n/U] @LIBRARY.CM@
     [savefilename/S] outputfilename/L
```

Using TRACE Information

Regardless of how it is called, when TRACE begins it types the following information for each procedure being traced.

PROGRAM NAME: procname	←The name of the procedure being traced. The last procedure called is the first procedure traced. If the procedure is not loaded with user symbols (/U local switch), this line is omitted.
RETURN LOCATION: xxxxx	←The address, xxxxx, to which control returns when the procedure completes execution. If the traced procedure is called by another program, it is a location in the calling procedure.
CALLED LOCATION: xxxxx	←The address, xxxxx, at which the procedure begins execution.

Using TRACE Information (Continued)

STACK POINTER: xxxxx	←The contents of .SP when the procedure was called.
ACØ xxxxx ACl xxxxx AC2 xxxxx	←The contents of the accumulators when the procedure was called
CARRY x STACK LEVEL x	←The state of Carry and the stack level of the procedure.

Following this header information is a table of five columns of stack information. This table displays the contents of each location in the stack, printed four locations per line. The first column in the table is of the form:

xxxxx/S+n

where xxxxx is the octal core address of the data in the first column and S+n is its octal address relative to the beginning of the stack. S+0 is the first location on the stack; S+4 is the fifth location; S+n is the n+1th location. The next four columns give the actual data in the four consecutive stack addresses. For example:

```
          data at location 15544,          data at location 15547,
        stack-relative address S+Ø       stack-relative address S+3
                      ↓                                   ↓
15544/S+Ø          15475        15471      4              4
1555Ø/S+4          4            3          33354          40
↑       ↑          ↑                                      ↑
 core   |   data at location 1555Ø,          data at location 15553,
address |   stack-relative address S+4       stack-relative address
        |                                    S+7

 stack-relative
    address
```

TRACE then prints the number of parameters, if any, passed to the procedure and the stack-relative address of the parameters. The actual data on the stack is explained in the following example.

TRACE Example

The following example illustrates use of TRACE and its output. This example consists of a main program, TEST, and a procedure SHELLSORT. TEST merely reads data input at the Teletype into array AR. It then calls SHELLSORT, passing the values in the array to it, which performs a string sort on the data. As shown in the source code, the bounds of array AR are dimensioned Ø to 6 in TEST, but the *for* statement of SHELLSORT indicates a lower bound of 1 and an upper bound of size(a), which will cause an error at run time. A listing of TEST and SHELLSORT follows.

```
TEST:    BEGIN STRING (3) ARRAY AR[Ø:6];
         EXTERNAL PROCEDURE SHELLSORT;

         OPEN (Ø, "$TTI");
         OPEN (1, "$TTO");
         READ (Ø, AR);
         SHELLSORT (AR);
         END
```

```
 PROCEDURE SHELLSORT (A);

STRING (3) ARRAY A;

    BEGIN INTEGER I, J, K, M;
          STRING W;

      FOR I := 1 STEP I UNTIL SIZE(A) DO M := 2*I-1;
      FOR M := M/2 WHILE M <> Ø DO

        BEGIN K := SIZE(A)-M;

          FOR J := 1 STEP 1 UNTIL K DO

            BEGIN FOR I := J STEP -M UNTIL 1 DO

              BEGIN IF A[I+M] >= A[I] THEN GO TO 1;
                W := A[I]; A[I] := A[I+M]; A[I+M] := W;
              END I;

        1:  END J

        END M

    END SHELLSORT;
```

TRACE Example (Continued)

The programs are then loaded together with the debugger and the contents of the ALGOL library. (The file name for the SHELL-SORT procedure is SORT.)

```
RLDR/D TEST/U SORT/U @LIBRARY.CM@     ←load TEST and SHELLSORT
 TEST                                  with symbols and debugger

 SHELL

R
```

Execution of the loaded programs begins when the user calls TEST. TEST waits for the user to type sevel input strings at the Teletype. The strings are read into array AR. When SHELL-SORT is called by TEST, a run-time error is reported, execution ceases, and a break file (BREAK.SV) is created.

```
TEST                                 ← run loaded program
ZZZ
AAA
BBC
REX                                  ← strings read into array AR
YU
AAA
B11
SUBSCRIPT OUT OF BOUNDS: LOCATION 1275  ← error message
BREAK                                   ← core image is saved
R                                         in "BREAK.SV"
```

The user then calls TRACE at the console with the TRACE command. Output of TRACE is shown on the following page. Words in italics in the trace are comments, inserted here only to aid the reader in understanding the data in the stack.

TRACE Example (continued)

```
TRACE                                        ← user brings in TRACE.SV
                                               to trace "BREAK.SV"
PROGRAM NAME:  SHELL                         ← called by TEST; see next frame.

RETURN LOCATION:    1Ø45                     ← return address in TEST
CALLED LOCATION:    1Ø71                     ← starting address for SHELL
STACK POINTER:     15733
ACØ        4        AC1    1252         AC2   155Ø3
CARRY Ø                STACK LEVEL    1
                     array data. ptr    array  dope  ptr
 15544/S+Ø         15475  ←A1    15471  ←A2    4   ← I    4   ← J
 1555Ø/S+4         4      ← K    3      ← M2   33354 ← W1 4Ø  ← W2
 15554/S+1Ø        Ø             116Ø          12Ø7         7  (I+M)
 1556Ø/S+14        1252          3336Ø         Ø            Ø
 15564/S+2Ø        S+16          Ø             S+2Ø         3Ø415
 1557Ø/S+24        64ØØ          Ø             Ø            Ø
 15574/S+3Ø        Ø             Ø             Ø            Ø
 156ØØ/S+34        S+12          Ø             Ø            Ø
 156Ø4/S+4Ø        Ø             Ø             S+2Ø
                                                    end of block ptr
PARAMETER   1: ARRAY AT CALLERS S+Ø
```

- -

```
PROGRAM NAME: TEST

RETURN LOCATION:    5625
CALLED LOCATION:    1ØØØ                ← starting address for TEST
STACK POINTER:     15652
ACØ        Ø        AC1       Ø         AC2      -1
CARRY Ø                STACK LEVEL    1
                     array  data ptr  array  dope ptr
 15463/S+Ø         S+12            S+6            Ø               Ø
 15467/S+4         S+2             Ø              3  dope size    1Ø2Ø2  array specifier
 15473/S+1Ø        Ø  lbound AR    6  hbound AR   33226 AR[Ø]     14Ø3   max/cur
 15477/S+14        33232           14Ø3           33236 AR[2]     14Ø3
 155Ø3/S+2Ø        33242           14Ø3           33246           14Ø2
 155Ø7/S+24        33252           14Ø3           33256           14Ø3
 15513/S+3Ø        55132  ZZ       55ØØØ  Z       4Ø5Ø1  AA       4Ø4ØØ  A
 15517/S+34        411Ø2  BB       414ØØ  C       511Ø5  RE       54ØØØ  X
 15523/S+4Ø        54525  YU       Ø              4Ø5Ø1  AA       4Ø4ØØ  A
 15527/S+44        41Ø61  B1       3Ø4ØØ  1       S+4
                                                    end of block ptr
NO PARAMETERS PASSED
```

- -

TRACE Example (continued)

The run-time error message

SUBSCRIPT OUT OF BOUNDS: LOCATION 1275

indicates the location of the error. Using the load map, a portion of which is shown, the user can locate the procedure containing the error.

```
SUSET 000140
ASTR  000141
SUNSE 000142
TEST  001000
SHELL 001071
READ  001403
LONG  002506
RTER  002507
ARER  002517
RTE0  002530
OPEN  003403
      003505
      ..4121
```

Portion of the load map, showing that the location of the subscript error (1275) is in SHELLSORT.

The load map gives the starting location of each run-time routine and procedure. Since 1275 is between the starting locations of SHELL and READ, the error must have occurred somewhere in SHELLSORT. To check the exact coding for the error, the user subtracts the starting location of SHELL from the error location:

1275 - 1071 = 204

and checks location 204 in the coding. The portion of the assembly code showing that location follows.

TRACE Example (continued)

```
               ;             BEGIN IF A[I+M] >= A[I] THEN GO TO 1;

00174'034013$        LDA     3,.SP
00175'021613         LDA     0,S+2,3            ;I
00176'031616         LDA     2,S+5,3            ;M
00177'113000         ADD     0,2
00200'051624         STA     2,S+13,3           ;TEMPORARY
00201'006012$        JSR     @SUBSCRIPT
00202'000002         2
00203'100000         SP+0                       ;A
00204'100013         SP+13                      ;TEMPORARY
00205'051624         STA     2,S+13,3           ;TEMPORARY
00206'006012$        JSR     @SUBSCRIPT
00207'000002         2
00210'100000         SP+0                       ;A
00211'100002         SP+2                       ;I
00212'051625         STA     2,S+14,3           ;TEMPORARY
00213'006011$        JSR     @STRCMP
00214'100014         SP+14                      ;TEMPORARY
00215'100203         @000203                    ;STRING
002                  SP+13                      ;TEMPORARY
                                                ;STRING

                                                TEMPORARY
```

Location 204 contains SP+13. This is the octal address of the error, relative to the start of the stack. The stack information output by TRACE shows that this location contains a 7. This value, according to the calling sequence for SUBSCRIPT, is the value of the subscript. Since TEST declared an upper bound of 6 for the subscript, the value is out of bounds.

This example also illustrates the kind of information output during a trace. For example, because SHELLSORT was the last procedure called, its trace is printed first. The header information for the trace indicates the return and called locations, the contents of the stack pointer and accumulators, the state of Carry, and the stack level. The actual stack information following this header shows that the first word of data on the stack is at location 15544. This word contains the pointer to array A (the first variable declared in SHELLSORT). This pointer is a location (15475) in the run-time stack for TEST. The trace for TEST shows a 33226 at that location. 33226 is, in turn, a byte pointer to the first word of data in array AR. When 33226 is converted to a data address and pointer (bits 0 through 14 is the data address; bit 15 indicates the right or left byte), it yields 15513. Location 15513 contains 55132, the ASCII code for ZZ, which is the first two characters input to the array. The remaining data in the array is stored in consecutive locations in TEST's stack.

TRACE Example (continued)

Back in the trace for SHELLSORT, the second location (15545) contains an array dope pointer to the array control table. The pointer is an address (15471) in the stack for TEST. Address 15471 contains the dope size (3); the next location in the stack is the array specifier, followed by the dimensions of the array (the low bound is Ø; the high bound, 6).

The third through sixth locations in the stack for SHELLSORT are the values of the integers I, J, K, and M. Note that these variables are pushed on the stack in the order in which they are given in the procedure. At the time of the run-time error, I, J and K were 4; M was 3. (The subscript that caused the error was, in fact, I+M or 7.) The next three words in the stack constitute the string specifie for W. Additional words on the stack are internal data used by ALGOL

Finally, the last word of data, S+20, is an end-of-block pointer, whi is a list of bounds of data areas used by the run-time routines. S+2 points to a stack-relative address. This address, in turn, contains a pointer to stack-relative address S+16,which contains a zero.

* * *

INDEX

Changes from Revision 4 to Revision 5 of the ALGOL User's Manuals, 093-000052.

Page	Nature of Change
34	a and b are string variables
5-4,6-2	Conditional designational expressions never follow *then* or *go to* keywords.
6-3	Under note 3, only the lefthand side of the assignment statement is now given in the format.
7-20	The statement tagged LOOP has been corrected in the example.
8-8	Array elements of SET have been corrected.
9-9,9-10	The format of the classify function has been corrected.
9-10	The classify example uses the ascii function, not the substr function.
9-16,9-17	The output examples required a fourth list item for the output given.
9-27	Null bytes are required after arguments in the COM.CM file.
9-28	The array is correctly identified as B2.
9-29	Correction was made to the manner in which chaining operates.
9-34ff	Cache Memory Management procedures (formerly called Software Virtual Memory procedures) have been incorporated into the manual, obsoleting application note 017-000016.
11-1	Reference to condition signalling has been removed. Reference to cache memory has been added.
B-7	Complex data type has been removed.
C-22	Descriptors of byteread and lineread are limited to four, instead of five.
C-41ff	Software Virtual Memory has been changed to Cache Memory Management.

Changes from Revision 4 to Revision 5 of the ALGOL User's Manuals, 093-000052 (Continued)

Page	Nature of Change
C-49	The section on ALGOL routines that use system calls now reflects the calls that have been added to the RDOS system.
D-7	SYSGENing under SOS has been updated to reflect SOS Rev. 9.
D-9	The dummy SOS.LB is included in the list of library tapes. A note on linking the multiply/divide library or changing LIBRARY.CM has been added.
D-11	Loading must include the SOS trigger.
App.D, App.G	References to the ALGOL library file in the RLDR command line have been corrected to read @LIBRARY.CM@.

DataGeneral

Software Documentation Remarks Form

How Do You Like This Manual?

Title ______________________ No. __________

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

Who Are You?

- ☐ EDP Manager
- ☐ Senior System Analyst
- ☐ Analyst/Programmer
- ☐ Operator
- ☐ Other ______________

What programming language(s) do you use? ______________

How Do You Use This Manual?

(List in order: 1 = Primary use)

- _____ Introduction to the product
- _____ Reference
- _____ Tutorial Text
- _____ Operating Guide
- _____ ______________

Do You Like The Manual?

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the topic order easy to follow?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you everything you need to know?

Comments?

(Please note page number and paragraph where applicable.)

From:

Name ______________ Title ______________ Company ______________

Address ______________ Date ______________

SD-00742

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP