ALGOL 60 VERSION 4.8C RELEASE NOTE

CP/M Version 2

When used with CP/M version 2 the Algol system will accept disk drive names extending from A: to P:

ASSEMBLY CODE ROUTINES

The program CODE.ALG and CODE.ASC is an aid to users who wish to add their own code routines to the runtime system. The program produces an output file which contains a reconstruction of the INPUT, OUTPUT, IOC and ERROR tables described in the manual. This file forms the basis of an assembly code program to which users may edit in their routines. Parts not required may be removed as desired. The resulting program should then be assembled and overlaid onto the runtime system using DDT, the resulting code being SAVEd as a new runtime system. It is important that the CODE program be run using the version of the runtime system to which the code routines are to be linked else the tables produced may be incompatible.

Another example program supplied is QSORT, the sorting procedure described in the manual.

LONG INTEGER ALGOL

Arun-L is a version of the RML Z80 Algol system in which real variables are represented not in the normal mantissa/exponent form but rather as 32 bit 2's complement integers. This runtime system is useful for those applications where greater precision is desirable but without the need to extend the number range to the extent allowed by the floating point representation, e.g. business programs. The number range allowed is from (2^31)-1 to -(2^31), (about ±2.15*10^9). The compiler itself remains unchanged. Variables declared as integers will still be represented as 16 bit 2's complement numbers. This document outlines the differences from the Algol system described in the manual.

STANDARD FUNCTIONS

The following functions have been removed:

sin, cos, sqrt, arctan, ln, exp

The function entier exists but is equivalent to a real to integer assignment. For example, the statements

```
i:=entier(x);
i:=x;
```

have the same effect.

LIBRARY PROCEDURES

The standard library file 'ALIB.ALG' can be used with Arun-L with the following exception

random has been removed.

Two additional library procedures can be found in file 'ARUNL.ALG'

```
pow10(n)
lrem(t,b)
```

DIVISION

Real division (/) always truncates the result towards zero in the same way as with integer division (%). A procedure has been added to the library (lrem) to give the remainder term lost by the division.

```
z:=lrem(t,b);
```

gives the remainder lost by the division

```
u:=t/b;
```

The result of lrem will always have the same sign as the quotient (or zero) in the same way as the MOD operator does for the integer case e.g.

t	b	t/b	lrem(t,b)
35	8	4	3
-35	8	-4	-3
35	-8	-4	-3
-35	-8	4	3

INPUT/OUTPUT

The decima[illegible] and [illegible]
for the addition

p[illegible]w!0(n,

where 'n' i[illegible]mall integer causes all subsequent calls to read to be scaled by [illegible]actor of 10^n. The digit string representing the number (including [illegible]onal and exponent fields) is read and the result scaled by 10^n. fractional part is then disregarded before returning the result. Fo[illegible]ple with n=2, on reading the number 123.4567 the result would be 1[illegible] On output the converse scaling is performed. The value to be outpu[illegible]irst converted internally into a digit string; the decimal po[illegible]s then effectively shifted left by 'n' digits before printing th[illegible]lt in the required format. The meaning of the format parameters [illegible]n unchanged. This scaling on output applies only to rwrite; the [illegible]er print routine (write) is unchanged.

RUNTIME ERR[illegible]ESSAGES

The following changes to the runtime error numbers given in the manual have been made.

8 Real [illegible]g integer) division by zero or lrem(t,0).
9 Over[illegible]in real multiply.
16 Over[illegible]In real addition.
17 Over[illegible]in real subtraction.
18 Illegal standard function called e.g. sin, cos etc.
19 Largest negative number -(2^31) with no corresponding positive representation. This error can occur from abs, *, /, rem, rwrite etc.

STRING HANDLING

The following string handling routines were written by Clare Tagg of the London School of Economics. The source code can be found in the file STRING.ALG.

General Philosophy

In RML Algol variable strings are stored in BYTE ARRAYs whereas literal strings are passed as STRINGs. This means that routines which may have variable or literal strings as arguments must have addresses as parameters. In the Algol library supplied in ALIB.ALG there are four such functions: **atext, tlen, smatch** and **blmove** as well as the literal string specific function **text**. As these functions require the frequent use of the functions **location** or **sloc** to find the relevant address, they are rather unwieldy to use. In designing new string functions it was decided to continue this distinction between variable and literal strings and have special versions for routines commonly having literal arguments. These routines are distinguishable by ending in an 's' and in general the rightmost string parameter is of type STRING.

In the following descriptions b is a single dimension BYTE ARRAY containing a string (ie. terminated by 0), i is an index in the byte array which indicates the start of the string and s is a literal string. Most of the routines use some of the string handling library routines (ie. **location, tlen, smatch, sloc, blmove, atext**). The routines provided are:-

```
BOOLEAN PROCEDURE seq(b1,i1,b2,i2);
```

Gives equality of two strings starting at b1[i1] and b2[i2].

```
BOOLEAN PROCEDURE seqs(b,i,s);
```

Gives equality of variable and literal string.

```
BOOLEAN PROCEDURE subeq(b1,i1,b2,i2,len);
```

Gives equality of substrings starting at b1[i1] and b2[i2] of length len. If len is negative it defaults to the length of the string starting at b2[i2].

```
BOOLEAN PROCEDURE subeqs(b,i,s);
```

Gives equality of substring starting at b[i] and literal string s of length of s.

INTEGER PROCEDURE sfind(b1,i1,b2,i2);

Finds start of string starting at b2[i2] in string starting at b1[i1]. Result is index in b1 or -1 if not found.

INTEGER PROCEDURE sfinds(b,i,s);

Same as sfind except 2nd operand is a literal string.

PROCEDURE scopy(b1,i1,b2,i2,len);

Copies string from b2 into b1 starting at b2[i2] for len characters starting at b1[i1]. If len is negative entire string is copied.

PROCEDURE scopys(b,i,s);

Same as scopy but 2nd operand is a literal.

PROCEDURE stext(dev,b,i,len);

Outputs string to device dev starting at b[i] for len characters or whole string if len is negative.

PROCEDURE sread(dev,b,i,len);

Reads a string from device dev into b starting at i. Reads len characters or until cr is found. On return len is set to the length of the string which is also terminated with a zero.

PROCEDURE sinit(b,i,ch,len);

Initialises integer or byte array b to character ch starting at i for len elements.

ALGOL 60 RELEASE NOTE
Compiler Version4.1T, Runtime System 4.7T

TSR-80 MEMORY MAP

The TRS-80 RAM starts at location 4200H instead of the normal zero for CP/M computers. The absolute addresses described on page 69 of the manual should be offset by this amount. Users adding assembly code to the runtime system are strongly advised to make use of the skeleton generated by running the program CODE described below.

TRS-80 CHARACTER SET

The RML Algol 60 compiler accepts the full ASCII character set described in the manual. Note that on the TRS-80 VDU a number of characters do not appear as their ASCII representation e.g.

"["	appears as	"→"
"]"		"←"
"{"		"↑"
"}"		"→"

None of these characters (or ^) can be input via the TRS-80 keyboard. The compiler has been modified to accept the following alternative symbols.

"<<"	as alternative to	"["
">>"		"]"
"@"		"^"

For example the statements

```
A<<I>>:=B@3; CRTL C:=&@C;
```

are the same as

```
A[I]:=b^3; CTRL C:=&^C;
```

The program TANCON on the distribution disk can be used to convert programs written using the above representations into the standard form.

Switch options attached to filenames may be enclosed by '<' and '>' instead of '[' and ']' (see page 39), for example:

```
DEV:=FINDINPUT("DATA.DAT<B>");
```

TRS80 GRAPHICS

The graphics routines described in the manual as "for the RML380Z only" (see pages 43,44,71 and 73) are available with the following differences:

1. The origin is at the bottom left hand corner, not 4 lines up.
2. For CHPOS the limits are $0 \leq x < 64$ and $0 \leq y < 16$.
3. For POINT and LINE the limits are $0 \leq x < 128$ and $0 \leq y < 48$
4. The EMT call is not available.
5. In POINT and LINE there is no difference in brightness between z=1 and z=2.
6. The x resolution is about twice that of the y resolution instead of being equal.

For programs using graphics it is often convenient to obtain characters from the keyboard without echo. This may be achieved with the call BIOS(3,0). (See page 36). Note also that all the control codes listed in C/1 of the LEVEL II handbook work when sent to the VDU via stream 1. For example, CHOUT(1,28) sends the cursor home to the top left corner. The source of the graphics routines is in ARUNG.ALG.

The distribution disc contains several simple examples of the use of graphics (TEST1 to TEST5). Some of these require a key to be struck before moving on to the next display.

The program DUCK is a game of "duck shooting"; respond with an angle in the range of about 45 to 90.

ASSEMBLY CODE ROUTINES

The program CODE.ALG and CODE.ASC is an aid to users who wish to add their own code routines to the runtime system. The program produces an output file which contains a reconstruction of the INPUT, OUTPUT, IOC and ERROR tables described in the manual. This file forms the basis of an assembly code program to which users may edit in their routines. Parts not required may be removed as desired. The resulting program should then be assembled and overlaid onto the runtime system using DDT, the resulting code being SAVEd as a new runtime system. It is important that the CODE program be run using the version of the runtime system to which the code routines are to be linked else the tables produced may be incompatible.

Other example programs supplied are QSORT, the sorting procedure described in the manual, and STARTREK, the well known game.

ALGOL 60

RELEASE NOTE

Compiler	Version	4.1C	
Runtime system	Version	4.1R	(380Z only)
		4.1C	(pure CP/M)

Users linking their own code procedures into the runtime system should note that the addresses of the necessary tables are contained in the list of pointers located near the start of the code. The examples given in the manual describe the mechanism; the exact locations of these tables will vary between the various versions of the runtime system.

380Z USERS ONLY.

If you wish to use a VDU as the outout device rather than the normal memory mapped VT screen then the pure CP/M version of the runtime system should be used (file ARUNC.COM on the disk). List the file README.DOC for more information.

A number of additional test programs can be found on the distribution disk which make use of the graphics facilities available.

A number of characters on the VT ROM do not print as the desired ASCII code. In particular, square brackets appear as left and right arrows, and braces (curly brackets) appear as one quarter and three quarters.

INTRODUCTION

RML Algol is an implementation of the Algol 60 language designed especially for small computers. Almost all the features of Algol 60 are implemented together with a significant number of extensions. The system is essentially portable, the compiler, which is itself written in Algol, having been "burned in" over a period of several years on PDP8 and PDP11 computers. The RML Algol for the Z80 running under the CP/M operating system consists of a one-pass compiler and a runtime program. The compiler translates the Algol source program into a machine independent intermediate code which specifies the sequence in which a number of subroutines is to be obeyed and which contains arguments for these subroutines.

The intermediate codes each occupy only one byte of memory resulting in a very compact object code. Roughly 10 bytes are required to store an average statement. The runtime program contains a loader for the compiler output and all the routines required to run the Algol program.

The compiler determines the minimum memory requirements of the system. The compiler and its runtime system together occupy about 12K bytes. Program work space and the CP/M operating system bring the total memory requirement to about 21K.

The runtime system requires about 8K bytes of memory, which together with program, data, and CP/M requirements allows sizable programs to run in as little as 16K bytes.

The availability of virtually the same compiler for Z80, PDP11 and PDP8 computers makes it possible to develop programs on larger computers and run high level programs on microcomputers which do not in themselves have the I/O or memory capability to support CP/M or the compiler.

The modular construction of the programs allows the user to optimise the runtime system to meet individual requirements. In particular users can add I/O handlers, assembly code routines and error handling without needing to become involved in the inner workings of the system.

RML Algol 60 provides an ideal medium for the distribution of precompiled software avoiding the need to supply program sources. Licences may be obtained from RML to redistribute software linked to the runtime system in this way.

THE RML ALGOL LANGUAGE

The RML Algol language is Algol 60 with a few restrictions. Some of these are a result of the one-pass nature of the compiler. For example variables must be declared textually before use. In other possible ambiguous situations that a multi-pass compiler could resolve, this compiler may require a "clue" as to the type of object being translated. These differences are described in the following sections. A number of extensions to the language have been introduced. These include the data type BYTE ARRAY, logical and MOD operators, and a significant number of functions. Appendix 3 gives a summary of the differences from the Algol 60 Report.

This manual describes the syntax of the language. The distribution disk includes a number of example programs which are described in appendix 4. Users new to structured programming may find it beneficial to refer to an introductory manual on the language and also to the Algol 60 Report (see Bibliography).

LANGUAGE KEY WORDS AND IDENTIFIERS

Some convention is required in Algol to distinguish between the language key words, e.g. BEGIN, END, ELSE, etc., and identifiers. In textbooks this is usually done by printing the key words in bold type. In actual compilers some other convention is needed depending upon the I/O hardware available. This compiler accepts two possible conventions. The first must be used where the I/O is restricted to upper case characters only. The second may be used where both upper and lower case are available. Two small utilities are provided to convert between these conventions (see appendix 4).

CONVENTION 1. Upper case only. All basic language words must be enclosed within single quote (') marks e.g.

```
SUM:=0;
'FOR' I:=1 'STEP' 1 'UNTIL' MAX 'DO'
SUM:=SUM+(X[I]-XM[I])^2;
RMS:=SQRT(SUM/MAX);
```

CONVENTION 2. Upper and lower case. All basic language words must be in upper case and all identfiers must use only lower case. The above example now becomes:

```
sum:=0;
FOR i:=1 STEP 1 UNTIL max DO
sum:=sum+(x[i]-xm[i])^2;
rms:=sqrt(sum/max);
```

The compiler decides which of the the two conventions is being used from the first word of the program, which must be BEGIN. Once selected the compiler expects the

upper case (which is how they appear if the identifier tables are printed). The compiler only checks the first 2 letters of the language key words, then skips until a suitable terminator is found (a closing quote or a non upper case letter). This implies that in the second convention adjacent language key words must be separated e.g.

THEN GOTO label; and not THENGOTO label;

One small difference to note is the representation of real numbers within the program. In the first convention decimal exponentiation is represented by "E" e.g. X:=1.234E-5 whereas in the second case a small "e" must be used e.g. x:=1.234e-5. Note that when data is being read by the program then a capital "E" is always used.

Users, if they so wish, may use any of the language key words for the names of identifiers. As an example the following declaration is perfectly acceptable.

BEGIN INTEGER real,begin,end;

For the sake of readability the examples given in this manual will for the most part use the upper/lower case convention.

PRE-DECLARED IDENTIFIERS

Certain procedure names are recognised by the compiler without declaration. Such identifiers may be regarded as having been declared in a fictitious outer block enclosing the entire program and thus in scope everywhere except where masked out by a local redeclaration. These procedures include the input/output routines which are discussed later and the standard functions as defined in the Algol 60 Report. In addition to these there is the procedure "ioc" which takes a single integer parameter, e.g.

ioc(n);

This routine serves a variety of purposes depending upon the value of n. These include input/output selection, format control and so on. These uses will be discussed in the following sections to which they apply. Another use of this procedure is as a simple way of linking to code routines within the runtime system. An inspection of the file "ALIB.ALG" on the distribution disk shows that the majority of the procedures consist simply of a formal definition of their parameters with a single call to "ioc" as the body of the procedure. Users may link in their own code routines by this same method. The details are discussed later.

STANDARD FUNCTIONS

sin(x)	x is in radians
cos(x)	x is in radians
arctan(x)	the result is in radians in the range -pi/2 to +pi/2.
ln(x)	natural logarithm
exp(x)	e to the power x
sqrt(x)	square root of x
abs(x)	absolute value of x
sign(x)	delivers -1, 0, or +1 according to whether x is negative, zero or positive.
entier(x)	returns the largest integer less or equal to x. Thus if x = 3.3 the result is 3. if x = -3.3 the result is -4. Note: as the result is integer the value of x must lie within the valid integer range of -32768 to 32767.

In each of the above procedures x is called by VALUE and thus the actual parameter may be an expression.

THE STRUCTURE OF AN ALGOL PROGRAM

One of the most important features of the Algol language is that it is structured. Just as round brackets define sub-expressions within expressions, so the brackets BEGIN and END enclose a set of statements which are treated syntactically as a single statement. These bracketed statements are used in controlling the order of execution of the program and largely replace the use of labels and GOTOs since they are obeyed as a whole or not at all. Such a statement is known as a compound statement or, if it contains any declarations apart from labels, a block. Statements within a compound statement are separated by semicolons. Strictly speaking, an Algol program is one statement, because it must start with BEGIN and end with END. A complete RML Algol program can be represented thus

BEGIN s1; s2; s3;.....sn END FINISH

The statements s1, s2 etc. may be of any type, including compound statements and blocks. The closing FINISH must be present. It is used by the compiler to check that there are the same number of BEGINs and ENDs.

is worth explaining why the "converted" consider this such an important feature. An examination of any Algol program reveals the modular nature of its structure, each block containing specific declarations of the variables, arrays and procedures relevant to its operation. Many users new to such languages are at first irritated by the need to declare explicitly every identifier in a program. While there may be some justification for such criticism when considering trivial program examples, the advantages become obvious as the programs become larger and the declarations become a small percentage of the program. The block in which an identifier is declared defines the "scope" of that identifier. Outside that block the identifier has no meaning and occupies no memory resource. The system thus takes on the role of resource management. The allocation of memory to variables is done dynamically as the program is being executed. On entering a block the system makes available those resources defined in the declarations and upon exit from that block these resources are reclaimed and made available for other uses; thus the memory required is always minimised. The penalty in terms of runtime speed is negligible as most of the organisation is done by the compiler. The total declarations throughout the program may in fact be in excess of the memory of the computer provided it is not all in scope at once. In languages such as Fortran, on the other hand, the nearest one can get to this feature is to work out some cumbersome EQUIVALENCE statements which probably take longer to define than the corresponding Algol declarations.

Economy of memory can also be achieved using "dynamic bounds" on array declarations. The declared bounds can be defined in terms of arithmetic expressions evaluated at runtime on entering a block. These bounds may be different each time the block is entered. The size of the arrays can be chosen to be the smallest that will do the required task.

Because memory is allocated dynamically in this way it is important that programs make no assumptions on entering a block about the initial values of variables declared therein. Between leaving a block and re-entering it the memory used by such variables may have been re-used for other purposes and indeed, in the case of procedures, the variables need not even occupy the same memory addresses each time the block is entered. If it is important that a particular variable should preserve its value between leaving and entering a block then its declaration should be removed to an outer block such that it remains "in scope".

The localisation of the scope of variables to that of the block in which they are declared also economises on the use of identifiers. The same names can be declared within several blocks without ambiguity as to which is being referred to. This also helps when program segments from various sources are combined without leading to major problems with conflicting identifiers.

BLOCKS AND DECLARATIONS

It has been mentioned already that variables must be declared before being used and that the presence of such declarations makes a compound statement into a block. All declarations must be placed immediately after a BEGIN or another declaration. This does not apply to labels, which are set by placing an identifier terminated by a colon just in front of the statement to be labelled. The form of a declaration of a set of unsubscripted variables of the same type is:

Type ident1, ident2,..... identn;

"Type" may be REAL, INTEGER, or BOOLEAN. ident1 etc. represent the names of the identifiers. Variables may be used within the block in which they are declared from the point of their declaration up until the corresponding END. Outside this range they do not exist and are said to be out of scope. Procedures and labels may be used before declaration, but the declaration must still be such that they are in scope at the point they are used. Variables and switches must be declared before they are used. This is so that the compiler can distinguish them from functions, which may be used before declaration. This restriction is not made in full Algol, in which variables may be referenced before declaration as long as they are in scope. To each identifier used in a given block there must be a corresponding declaration. It follows that all identifiers declared in a block must be unique. The same identifiers may however be re-used within other blocks. Where an identifier is used within an inner block and also an enclosing outer block then the inner declaration takes precedence for the duration of the inner block. The outer declaration is effectively "masked out" but its value is preserved and again becomes accessible on leaving the inner block. Consider the following:

```
BEGIN REAL x,y,p; INTEGER i,p;
        s1; s2;
        BEGIN REAL x,z; s3;
        END
END
```

Statements s1 and s2 cannot refer to z because it is out of scope; s3 cannot refer to the first x because the inner declaration will take precedence. When s3 refers to x it will always be the one declared with z, but s3 may refer to y and to i. The identifier p is declared twice within the same block and will generate a compile time error.

PROGRAM LAYOUT AND STYLE

The layout of an Algol program is almost entirely within the hands of the programmer. Such concepts as line number or column number have no meaning in Algol. Layout characters such as new lines, spaces, and tabs are generally ignored except for the following cases.

1. Spaces are significant within strings.
2. Language key words and double character symbols should contain no embedded layout characters e.g.

```
BEGIN
a>=b
a:=b
```

and not:

```
BE GIN
a> =b
a: =b
```

In all other contexts the programmer is free to lay out the program as he wishes. The resulting text may thus range from the very elegant to the totally unintelligible. As intelligent layout requires no additional work and results in programs far easier to follow and modify, users are urged to develop a good layout style which should reflect the block structure inherent in an Algol program. Use tab stops to indent text with BEGIN and END pairs aligned as in the examples given in this manual. Labels should start on a new line to the left of the program statements. Statements are separated by semicolons or language key words and not by new line characters, so that if necessary expressions may extend over several lines of text. Similarly one line may contain several shorter statements.

Identifiers should reflect the nature of the quantities they represent; spaces may be included within identifiers if it makes the resulting text more readable. They will be ignored by the compiler. Frequently used loop variables and array subscripts are often most conveniently represented by a single letter identifier e.g.

```
end of file:=char=&^Z;
total:=total+term;
FOR i:=lower STEP interval UNTIL upper DO ....
volume:=height*length*width;
```

Comments should be included where they make the workings of the program more understandable and are described later.

DATA TYPES

The data in the computer memory which may be manipulated by the program is either numeric or Boolean. Numeric values may be real or integer. The difference is that integers have no fractional part and occupy 2 bytes of memory, while real quantities are held in exponent and mantissa form and occupy 4 bytes. The Algol system converts numbers from one type to the other whenever necessary. Arithmetic expressions may contain a mixture of real and integer quantities.

Numerical and Boolean data may appear within the program as literal values. An integer literal is one which has neither a decimal point nor a decimal exponent part. Examples are 3, -200, +1234. The range of integers is -32768 to 32767. Real numbers contain either a decimal point or a decimal exponent, indicated by E or e (depending upon which convention is in use), or both. Examples are 0.1, -2.345, 1.2E3, 25.7e-7. Real numbers use one byte for the exponent and three for the mantissa giving a range of magnitudes from approximately E-38 to E+38 and between 6 and 7 decimal digits of precision.

Boolean literals are the language key words TRUE and FALSE.

IDENTIFIERS AND SYMBOLS

A RML Algol program consists of a sequence of symbols which are printing characters on a standard teletype, terminal or its equivalent. Editing and layout characters are generally ignored by the compiler except where indicated in the following sections. The symbols are frequently grouped into units which the compiler treats as a single entity. These groups are the numeric constants just described, the language key words such as BEGIN and TRUE (which are strings of letters enclosed in single quotes or in upper case depending upon the convention being used), and identifiers. Only the first two letters of a language key word are significant. An identifier is the name given by the programmer to a variable, label, switch, array or procedure. There is a small group of Algol symbols which are made up of two characters. These are the assignment operator (:=), greater than or equals (>=) and less than or equals (<=). Identifiers consist of any number of alphanumeric characters of which the first must be a letter. Only the first six characters are used, any extra ones being ignored. Letters are in upper or lower case depending upon the convention being used. Examples are x, fred, m1, m2, abc123, abc1234. The last two are identical in RML Algol.

All identifiers referring to variables and switches must be declared before they are used so that the compiler knows to what type of object they refer. Labels and procedures may be used before they are declared because the type of the identifier can be deduced from the context in which it is used.

ARRAYS

Subscripted variables in Algol are known as arrays. Real, integer, and Boolean variables may be subscripted, while byte variables must be subscripted. Like other variables, arrays are declared at the start of a block. REAL ARRAY and ARRAY are equivalent. The declaration of an array with one subscript has the form:

```
ARRAY ident[ae:ae];
```

The ae's represent arithmetic expressions defining the bounds of the subscript. Either or both bounds may be negative or zero, as long as the second one is not lower than the first. Real ae's will be rounded towards the nearest integer. If there is more than one subscript, the bound pairs are separated by commas; there is no limit to the number of subscripts other than the amount of availible memory. If more than one array is to have the same bounds, the bounds need only be specified after the last one. One declaration may contain any number of array names:

```
INTEGER ARRAY ia[1:30,1:5];
BOOLEAN ARRAY ba1, ba2 [1:n,1:3], ba3 [0:20],
              ba4, ba5 [1:2*n];
```

When arrays are declared with variable bounds care should be taken that the variables have defined values. This means that they should be declared in an outer block and have been assigned values. This feature is used to partition the available storage according to the data.

When an array is used an arithmetic expression is put in each subscript position. The following are possible statements, using the examples above. Conditional expressions are described later.

```
ia[1,4]:=7;
ia[n,m]:=ia[ia[n,2],ia[n,m]];
ia[n,3]:=IF ba2[3,1] AND ba3[n] THEN
        ia[ia[3*n,1], IF ba3[0] THEN 2 ELSE n]
        ELSE n;
```

ARRAY MEMORY LAYOUT AND BOUND CHECKING

The elements of an array occupy a contiguous region of memory. In multi-dimensional arrays the last subscript varies most rapidly. For example, consider a two dimensional array declared as

```
a[m:n, p:q]
```

The address of element a[i,j] would be given by an equation of the form

```
base + size * ((i-m)*(q-p+1)+(j-p))
```

where size is 1, 2 or 4 depending on the type of the array (Boolean or

byte, integer, or real). At runtime a check is made that the address computed lies within the limits allocated for the storage of the array. Note that for multi-dimensional arrays this does not necessarily mean that all subscripts lie within the declared bounds. For example, the element a[-1,15] would be acceptable for an array declared as a[0:9,0:9].

BYTE ARRAYS

The introduction of BYTE arrays in RML Algol is an extension to the language as defined in the Algol 60 Report. BYTE arrays allow for the efficient use of memory when string manipulation or small integer values are required. Within expressions BYTE array elements are treated as type INTEGER, and may be used in any context where an integer is allowed.

Within expressions the contents of a byte array element become the integer value, and the 8 most significant bits of the integer are set to zero. Thus

```
i:=b[n]
```

will always yield a positive value for i in the range 0 to +255.

When assigning to a byte array element, the expression on the right hand side is first converted to type integer if necessary, the eight least significant bits of which are then assigned to the byte array element, the 8 most significant bits being discarded without any checking. Thus

```
b[1]:=-1;
i:=b[1]
```

will assign to i the value +255. The only time the compiler distinguishes between an INTEGER ARRAY and a BYTE ARRAY is at the declaration. In all other contexts the two may be used interchangeably. In particular, a formal procedure parameter specified as type INTEGER ARRAY or BYTE ARRAY will accept either as the actual parameter. E.g.,

```
BEGIN INTEGER ARRAY i[0:100];
      BYTE ARRAY    b[0:100];

PROCEDURE xx(a); INTEGER ARRAY a;
BEGIN
      ........
END;
     xx(i); xx(b); ...
```

will be accepted.

SIMPLE EXPRESSIONS

An expression is a section of program which delivers a result. The result may be of type REAL, INTEGER, BOOLEAN, or LABEL.

The result of a numerical expression is real unless all the variables and literals within it are integer and it contains no real operators. The real operators are exponentiation (denoted by ^) and real division (/). Arithmetic expressions are evaluated with due regard to operator priority and from left to right where these are equal. Parentheses may be used to change the order of evaluation. The following is a list of the arithmetic and logical operators together with their priorities.

operator	priority	meaning
		(highest priority)
^	3	exponentiation
*	2	multiplication
/	2	real division
%	2	integer division
MOD	2	integer modulus
+	1	addition
-	1	subtraction
MASK	1	logical AND
DIFFER	1	logical EXCLUSIVE OR
!	1	logical OR
		(lowest priority)

The operators MOD, MASK, DIFFER, and ! are additional to those defined in the Algol 60 Report.

The operators %, MOD, MASK, DIFFER, and ! take two integer operands and deliver an integer result. The result of integer division (%) is truncated towards zero. The result of integer modulus (MOD) is the remainder lost by integer division. Note that

```
i MOD 0
```

will always return the value zero while

```
i%0   or   x/0
```

will give a division by zero runtime error.

The logical operators MASK, DIFFER and ! consider each of the two integer arguments as a pattern of 16 bits thus:

```
3 ! 5       =7
3 MASK 5    =1
3 DIFFER 5  =6
```

Apart from the cases just discussed, expressions may contain any mixture of real and integer quantities, and conversion between types

will occur automatically as context dictates. For example

```
BEGIN  REAL x,y;
       INTEGER i,j;
       ......
       i:=x*y; x:=x*i; i:=x^y;
       i:=x^i; x:=i%j;
       ......
```

are all valid operations.

In conversion from real to integer the result is rounded towards the nearest integer value.

Boolean expressions are made up of Boolean variables, the literals TRUE and FALSE, arithmetic relations, Boolean procedures, and Boolean operators. The Boolean operators are in order of precedence from highest to lowest priorities.

NOT	NOT b: is FALSE if b is true and TRUE if b is FALSE.
AND	b AND c: is TRUE if both b and c are TRUE otherwise it is FALSE.
OR	b OR c: is TRUE if either b or c is TRUE (regardless of the other) otherwise it is FALSE.
IMPLIES	b IMPLIES c: is FALSE only if b is TRUE and c is FALSE, otherwise it is TRUE.
EQUIVALENT	b EQUIVALENT c: is TRUE if b and c have the same truth value and FALSE otherwise.

The relational operators are

=	equals
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
#	not equals

Relational operators are dyadic. Care should be taken with the use of = and # where either argument is of type real. In this case an exact equals only occurs if both arguments have exactly the same bit pattern, which in the context of real quantities involving rounding may not be very meaningful. A more sensible test to make may be to check if the absolute difference is less than (or greater than) some small quantity e.g.

replace

```
IF x=y THEN ....
```

with

```
IF abs(x-y)<0.0001 THEN ....
```

As with arithmetic expressions, the order of evaluation may be changed by the use of parentheses to group terms.

Examples of Boolean expressions are:

```
NOT x<5 OR bv1 AND (y=0 OR bv2)
x # (y-5)
+(x-5)^2 <= 20
```

where x and y are numeric variables or procedures and bv1 and bv2 are Boolean variables or procedures.

Note the plus sign before the parentheses in the last example. If this had been absent, the compiler would have assumed that the bracket enclosed a Boolean expression and would have indicated an error on finding the closing bracket instead of the expected relational operator. In the first example the brackets do enclose a Boolean expression. (This way of forcing the compiler to recognise a bracketed arithmetic expression within a Boolean expression is not necessary in full Algol 60). The RML Algol compiler can deal with expressions such as x-5>=20 correctly because x must be numeric. Similarly, in the second example, the expression after the # must be numeric, so the compiler does not need a plus sign. In this case the brackets are not essential either.

STRINGS AND CHARACTER LITERALS

A string in Algol consists of a sequence of characters enclosed within double quotes e.g.

```
text(1,"Hello Dolly");
```

All characters within the quotes with an ASCII value less than 32 are ignored. This includes carriage return, tab, and control characters but space is permitted. In order that these and other characters may be included within strings the following convention is used. The characters * and ^ have special significance and are always considered in conjunction with the character following.

CONTROL CHARACTERS

```
^X
```

will insert CONTROL-X into the string. In general ^ has the effect of stripping off all but the 5 least significant bits of the following character.

LAYOUT CHARACTERS

*N	Inserts carriage return/linefeed into string.
*C	Inserts carriage return.
*L	Inserts linefeed.
*S	Inserts space (a literal space is also allowed).
*T	Inserts tab.
*P	Inserts new page (formfeed).

All other characters following * are taken literally; in particular,

```
**      Inserts *
*^      Inserts ^
*"      Inserts "
```

For example

```
text(1,"An example *"STRING*"
*Smight look lik
e this*NX*^2+Y*^2=");
```

will print on the console as:

```
An example "STRING" might look like this
X^2+Y^2=
```

The internal representation of a string is a series of characters stored in sequential bytes terminated by a zero value.

CHARACTER LITERALS

The literal value of any character may be found using an ampersand '&' character followed immediately by the required character. For example to convert a character digit to a number between 0 and 9 we have

```
i:=chin(1)-&0;
```

or to output an X then

```
chout(1,&X);
```

The convention described above involving * and ^ also applies, so to check for end of file (CONTROL-Z),

```
i:=chin(dev);
IF i=&^Z THEN ....
```

or to check for a carriage return character.

```
IF i=&*C THEN ....
```

The two exceptions are &*N and &*" which will lead to the wrong result. The first generates 2 characters and the second is written &".

Character literals are of type integer.

ASSIGNMENT STATEMENTS

The general form is:

variable:=expression

The variable on the left hand side is assigned the value of the expression on the right. Note that in Algol the assignment operator is the double character symbol (:=) and not the equal sign (=) which is a relational operator. Multiple assignments are not implemented in this version of Algol. If the variable is Boolean the expression must also be Boolean. If the variable is numeric the expression may deliver an integer or a real result; it will be converted to the type of the variable if necessary, real numbers being rounded towards the nearest integer. Examples:

```
i := 3+j;
x := IF be THEN j%5 ELSE 36.75;
bv := i#7
```

CONDITIONAL EXPRESSIONS

A conditional expression is one that takes one of several values depending on the result of one or more Boolean expressions. The general form is:

IF be THEN se ELSE e

be stands for Boolean expression (it may itself be conditional), se stands for simple expression, which must not be conditional, and e for any expression. Because e may also be conditional the form can be extended:

IF be THEN se ELSE IF be THEN se ELSE e

The expressions must all be numeric, all Boolean or all designational (these are described later). A conditional expression is made unconditional by enclosing it in round brackets. The following is legal Algol:

```
IF IF bv1 THEN x#3 ELSE y=0 THEN
(IF bv2 THEN 25 ELSE 30) ELSE x+y
```

The first be is conditional. The ae (arithmetic expression) within brackets is conditional and would not be allowed in that context without its brackets. It is a general rule of Algol that IF must not follow immediately after THEN. This is because it can result in ambiguous code. Examples of conditional expressions are:

```
a:=IF a>0 THEN a*a ELSE 0;
large:=IF a>b THEN a ELSE b;
max:=IF a>=b AND a>=c THEN a
     ELSE IF b>=a AND b>= THEN b ELSE c;
a:=IF IF x>0 THEN y>0 ELSE y<50 THEN 3*x ELSE 0;
```

CONDITIONAL STATEMENTS

These have the same form as conditional expressions, except that it is not necessary for there to be an ELSE part. There are consequently two forms of conditional statement:

```
IF be THEN s1
IF be THEN s1 ELSE s2
```

s1 must not be conditional but s2 may be. In the first case no statements are obeyed if the be delivers a FALSE result. In the second case s1 is obeyed if the result is TRUE otherwise s2 is obeyed. As s2 may be conditional this form can be extended indefinitely:

```
IF be1 THEN s1 ELSE IF be2 THEN s2 ELSE s3
```

Just as expressions are made unconditional by enclosing them in round brackets, so statements are unconditional if they are enclosed in the brackets BEGIN and END. Examples of conditional statements are:

```
IF a>0 THEN sum:=sum+a;

IF char=&^Z THEN close(dev);

IF samp>max THEN max:=samp
ELSE IF samp<min THEN min:=samp;

IF a>b AND c>d THEN
BEGIN ......
END ELSE
BEGIN ......
END;
```

FOR STATEMENTS

A FOR statement allows the repeated execution of a statement with different values of a variable known as the controlled variable. The general form is:

```
FOR variable:=fle,fle,.....fle DO s1
```

s1 may be conditional. The controlled variable must be real or integer and not subscripted. (In full Algol 60 subscripted variables are allowed.) fle stands for "for list element". Lists containing only one element are allowed. There are three types of for list element (1) an

arithmetic expression (2) a STEP element and (3) a WHILE element.

A STEP element has the form "ae STEP ae2 UNTIL ae3". After each execution of the controlled statement the value ae2 is added to the variable. Before each execution of s1, including the first, the variable is tested against ae3. If it is greater than ae3 and ae2 is positive, or less than ae3 when ae2 is negative, then the element is said to be exhausted. It should be noted that ae2 and ae3 are evaluated each time they are used, so that the value may be changed by the execution of the controlled statement. A STEP element may result in the statement not being executed at all, for example if ae>ae3 and ae2>0.

A WHILE element has the form "ae WHILE be". On each iteration the arithmetic expression is evaluated and assigned to the variable. Then the Boolean expression is evaluated and if the result is TRUE the statement is executed, otherwise the element is exhausted. Examples of FOR statements are:

```
FOR i:=min STEP 1 UNTIL max DO sum:=sum+s[i];
FOR i:=1 STEP i UNTIL 1024 DO ....
FOR i:=1, 3, 99, j, -6, 11 DO ....
FOR x:=1, x*2 WHILF x<.025 DO ....
FOR i:=100 STEP -1 UNTIL -100 DO ....
FOR x:=0.1, 1, x*5 WHILE x<1000, 20 STEP -5 UNTIL 0 DO..
```

FOR loops can be nested as deeply as desired. For example,

```
FOR i:=1 STEP 1 UNTIL max DO
FOR j:=1 STEP 1 UNTIL i DO a:=a+b[i,j]^2;
```

Matrix multiplication might look like:

```
FOR i:=1 STEP 1 UNTIL m DO
FOR j:=1 STEP 1 UNTIL n DO
BEGIN  x:=0;
       FOR k:=1 STEP 1 UNTIL p DO x:=x+a[i,k]*b[k,j];
       c[i,j]:=x;
END;
```

The body of the FOR loop may be a dummy statement, for example to skip to the start of a new line.

```
FOR i:=chin(dev) WHILE i#*C DO ;
```

The loop variable may also be a dummy:

```
FOR i:=0 WHILE test DO body;
```

In this case body may be a procedure or block which sets a Boolean variable test, or test could itself be a Boolean procedure.

```
FOR i:=body1 WHILE test DO body2 ;
```

In this example body1 could be an integer or real procedure, test a Boolean procedure, and body2 a procedure or block, giving several

possibilities for loop construction.

A FOR loop is not a simple statement and cannot be called following the THEN part of a conditional statement unless enclosed within a BEGIN and END. It may however follow the ELSE clause without needing an enclosing BEGIN and END.

DUMMY STATEMENTS

A dummy statement is one in which there is nothing before the terminating END, ELSE or semicolon. Examples are:

```
BEGIN END
IF be THEN ;
;;
BEGIN s1; END
IF be THEN ELSE ;
PROCEDURE dummy; ;
```

COMMENTS

RML Algol allows three types of comment. Any symbols appearing after an END until the first occurrence of semicolon, FINISH, END, or ELSE are ignored. These are known as END comments. E.g.

```
END this is ignored;
END so is this ELSE
END and this also FINISH
```

The other form is

```
COMMENT any sequence not containing semicolon;
```

This form is allowed after a semicolon or after a BEGIN. Within comments single quotes must be matched. An alternative to using the key word COMMENT is to enclose text within braces. This is an extension to the Algol 60 Report. Such comments may contain embedded matching braces, or embedded unmatched single quotes. This form of comment may be used anywhere that COMMENT may be used. E.g.

```
{this {comment} is ignored}
```

Full Algol 60 also allows an additional type of comment within procedure calls and declarations.

LABELS, SWITCHES AND GOTO STATEMENTS

Any statement may be labelled by preceding it with an identifier and a colon. The scope of the label is the block in which it occurs. Program control is transferred to a labelled statement by a GOTO statement.

```
BEGIN REAL x; s1; s2; GOTO lab; s3;
lab: s4
END
```

The following is not allowed because the label is not in scope:

```
GOTO lab;
BEGIN REAL x; s1; s2;
lab: s3
END;
```

Labels in an outer block may however be accessed from within an inner block e.g.

```
BEGIN REAL q;
lab:    s1; s2;
        BEGIN REAL y;
                s3; GOTO lab;
        END
END
```

It should be noted that a compound statement does not become a block because there is a labelled statement within it. The second example would have been allowed but for the declaration of x. For the same reason labels in different compound statements but within the same block must have different names.

A switch is a list of labels declared at the start of a block. All the labels must be within scope at the declaration.

```
SWITCH s:=lab1,lab2,lab3;
```

The simplest use of a switch is in a GOTO statement, GOTO s[ae]. The ae is evaluated and is used as an index to the list of labels in the declaration. If, for example, the ae has the value 2 the effect of the statement is the same as GOTO lab2. If the value of ae is less than 1 or greater than the number of labels in the declaration then the effect is that the statement is treated as a dummy statement. Example:

```
BEGIN SWITCH sw:=case1,case2,case3;
try:    text(1,"casenumber=");
        GOTO sw[chin(1)-&0];
        text(1,"*Nillegal value"); GOTO try;
case1: ....
case2: ....
case3: ....
```

DESIGNATIONAL EXPRESSIONS

Designational expressions are like arithmetic or Boolean expressions. In a designational expression the elements may be labels or switch elements. The full definition of a GOTO statement is:

GOTO de

where de stands for designational expression. An example:

GOTO IF x=0 THEN lab1 ELSE IF b THEN s[i+3] ELSE lab2

Designational expressions may also result in the address of a procedure, as will be described in the section on procedure parameters. This is an extension of Algol 60, in which designational expressions are only allowed for labels.

PROCEDURES

A procedure is a statement which is declared at the start of a block, but it is not executed when the block is entered. It is given an identifier and the appearance of its name causes the statement to be executed. The simplest type of procedure has no parameters and does not deliver a result.

```
BEGIN
PROCEDURE dothis; s;
    s1; s2; dothis; s3; dothis
END
```

s is the statement which is executed when dothis appears in the program. It is known as the body of the procedure. s is treated as a block even if it has the form of a compound statement or a single statement. This is to prevent GOTO statements from leading into a procedure which is not active.

A procedure may deliver a result of type REAL, INTEGER, or BOOLEAN. Such a procedure is known as a "type procedure" or function. Its name can then be used in expressions, which will cause the procedure to be executed and the result to be used in evaluating the expression. Within the procedure body the value which will be returned is set by assigning it to the name of the procedure. Such an assignment statement may occur anywhere within the procedure body and there can be any number of them. Execution of the procedure continues until either the end is reached or a GOTO leads out of it. If the name of the procedure occurs within the procedure body itself, except on the left of an assignment as just explained, then the procedure will call itself and is said to be recursive. The following example illustrates several points:

```
BEGIN INTEGER i;
        INTEGER PROCEDURE j;
        IF i<0 THEN GOTO nogood
        ELSE IF i=1 THEN j:=0 ELSE
        BEGIN i:=i-1;
              j:=j+i
        END procedure j;
        i:=10; i:=j;
nogood:
END
FINISH
```

The procedure refers to variable i which is declared in the main program in the same block as the procedure. The declaration of i must come first or the compiler would have assumed that it refered to an as yet undeclared procedure of type Boolean (lines 3 and 4) or integer (line 6). Line 5 would have failed because the identifier to the left of := must be already declared. There must always be some condition which causes a recursive procedure to deliver a result or exit without recursing, as on lines 3 and 4. If this had not been done, or if i had not been decremented on line 5, the procedure would have called itself until the available storage was used up. The label nogood is attached to a dummy statement. Note also that the body of the procedure does not have to be enclosed by BEGIN and END; in this case it is a conditional statement.

PROCEDURES WITH PARAMETERS

The action of a procedure can be made to depend upon data supplied to it through a list of parameters at the time it is called. The procedure declaration contains a list of formal parameters. These are the names which are used within the body of the procedure. The type of each formal parameter is given in a specification, which looks rather like a set of unsubscripted variable declarations. The list of formal parameters is enclosed in round brackets and is placed immediately after the name of the procedure. The identifiers are separated by commas. Only the names are given, not subscripts or procedure parameters. For example,

```
REAL PROCEDURE p(x,y,a,r,lab);
VALUE y; REAL x,y;
REAL ARRAY a;
REAL PROCEDURE r;
LABEL lab;
```

In full Algol 60 a more complicated type of parameter separator (the "fat comma") is also allowed.

NUMERIC AND BOOLEAN PARAMETERS BY VALUE

This is the simplest type of parameter. When the procedure is called the actual parameters are evaluated and the value is passed to the procedure. Within the procedure body a parameter called by value acts in every way like a variable declared within the procedure, except that it is assigned an initial value when the procedure is entered. The value may be changed within the procedure but the new value is not accessible once the procedure has finished.

```
PROCEDURE p(i,x,b); VALUE x,b,i;
REAL x; BOOLEAN b; INTEGER i;
BEGIN IF b THEN a:=x+1 ELSE a:=x-i;
      x:=2*i; a:=x+i;
END
```

"a" is a variable previously declared outside the procedure. Note that the VALUE specification must come before the part specifying the types of the parameters. The RML Algol system converts between INTEGER and REAL if the types of the actual and formal parameters are not the same. No other type conversions are allowed. A possible call of this procedure is:

```
p(5.32, 2.5*y, TRUE AND z>0)
```

where x and z are numeric variables.

Note the TRUE before the relation z>0. If this had not been present the compiler would not have known that it should be compiling a Boolean expression. All actual parameters which are Boolean values must start with a Boolean identifier, or one of the symbols NOT, TRUE or FALSE. For the same reason any identifier which is the first item of an actual parameter must have been declared, so that the compiler knows what type of expression to evaluate. It is never necessary to enclose an actual parameter in brackets. These rules are needed because the RML Algol compiler makes only one pass through the source program. They are not necessary in full Algol 60.

VARIABLES CALLED BY NAME

Any formal parameter which is not specified to be VALUE is said to be called by NAME. Instead of a value being passed to the procecedure, the address of the variable is transmitted. It follows that the actual parameter must be the name of a variable of the correct type. (In full Algol 60 an expression is allowed and the address of a routine to evaluate it is transmitted). When a variable called by name is assigned to within the procedure body the variable specified in the call is changed. Thus, variables called by name do not act like locally declared variables. The formal name stands for the actual name used in the call. The actual parameter is brought within the scope of the procedure body.

In RML Algol, array parameters must be called by name. (The full Algol 60 call of array names by value involves making a local copy of the whole array.) The actual parameter is the name of the array, without subscripts. Within the procedure body the formal array name is used with subscripts. There must be the same number of subscripts as in the original declaration of the array whose name was used as a parameter. Array elements (an array name followed by subscripts) may be used within or as an actual parameter, but only when the formal parameter is by value. This compiler accepts the use of BYTE ARRAY and INTEGER ARRAY interchangeably in procedure calls. (See the section on byte arrays).

Objects called by name may be passed on from procedure to procedure through the parameter list. Unsubscripted variables called by name may be used as the controlled variable in a FOR statement.

```
BEGIN ARRAY ar[1:20];
REAL x; INTEGER i;

REAL PROCEDURE rp; rp:=i^2+2;

PROCEDURE p(a,k,z); VALUE a;
REAL a; INTEGER k; ARRAY z;
FOR k:=1,2 DO z[k]:=rp*a;

      ar[1]:=10;
      p(ar[1],i,ar);
END
FINISH
```

When procedure p is called, the value of the parameter a is initialised to be 10.0 and k within the procedure becomes equivalent to i in the main program. When rp is called for the first time i has the value 1 so the result of the expression which is assigned to rp is 3.0. z[1] is assigned the value 3.0*10.0, which means that ar[1] becomes 30.0. Next time, when i=2, the value of rp is 6.0 but a still has the value 10.0 so the effect of the statement is ar[2]:=60.0.

STRING AND SWITCH PROCEDURE PARAMETERS

When the formal parameter is a string, the actual parameter may be either a string of characters enclosed in double quote marks or, if the call is within a procedure having a string parameter, the name of such a string. String parameters can only be used as actual parameters in further procedure calls, so it follows that the information in the string (as opposed to the address of the string, which is the information transmitted to the called procedure) can be used only by machine code called from within the procedure. This may be done by using the pre-declared procedure text(device,string), string handling procedures in ALIB.ALG or some machine code written by the user and activated by an ioc call. E.g.

```
PROCEDURE moan(message,num);
VALUE num; INTEGER num; STRING message;
BEGIN   text(1,"*NError at line ");
        write(1,num); text(1,message);
END;
```

The use of switch parameters is straightforward. The actual parameter is the name of a switch. When this name is used within the procedure body (with a subscript) the effect is as if it had been used in the block in which the switch was declared. However in RML Algol (but not in full Algol 60), if the execution of a parametric switch leads to within a procedure which has been called recursively, then the return is to the most recent call of the recursive procedure. If this consideration is important the RML Algol programmer should use several label parameters instead of a switch. RML Algol finds the correct incarnation of a recursive procedure if it is jumped into through a label parameter. A procedure has been called recursively if there is more than one call in force. If A calls B, B calls C, and C calls A then A has been used recursively. In practice this restriction is unlikely to prove to be a limitation.

LABELS AND PROCEDURES AS PARAMETERS

The treatment of parameters of type LABEL, PROCEDURE, REAL PROCEDURE, INTEGER PROCEDURE and BOOLEAN PROCEDURE is similar. The actual parameter is a designational expression which in RML Algol (but not in full Algol 60) must be preceded by a type specification. This requirement is included so that the one-pass RML Algol compiler may allow as yet undeclared procedures and labels to be used in procedure actual parameters. As with variables called by name, arrays, and switches, it is the address of the label or procedure which is passed to the procedure being called. The actual parameter must be in scope at the point of call but need not be within the scope of the called procedure; its use as a parameter effectively brings it within scope. If a procedure body contains a label declared with the same identifier as a formal parameter of the procedure, then the formal parameter will take precedence until after the declaration of the label. (In full Algol 60 the block structure always determines the order of precedence).

The pre-declared functions and input/output names cannot be used as procedure parameters, where the formal parameter is a procedure. A dummy procedure which calls the pre-declared one must be used. This is because the pre-declared procedures are not treated in the same way as those declared by the user, in order to shorten compiled programs and to increase the speed of execution. (In full Algol 60 the pre-declared procedure names can be used in this way). The pre-declared procedures can of course be used in expressions where the formal parameter is a value parameter. sin(cos(3)), for example, is allowed.

The RML Algol compiler does not differentiate between name and value calls of parameters which are switches, strings, labels, and procedures. Where the

(only allowed for labels and procedure types) the value is calculated on procedure entry only, and not each time the parameter is used within the procedure body. The calls of all these parameters are therefore by value, although the compiler does not force the user to specify this.

To illustrate these points, suppose that two procedures have the following headings:-

```
PROCEDURE p(s,lab,rp,st);
SWITCH s; STRING st; LABEL lab;
REAL PROCEDURE rp;

REAL PROCEDURE x(y,st); VALUE y;
REAL y; STRING st;
```

A possible call is:

```
p(sw, LABEL IF be THEN lab1 ELSE lab2,
REAL PROCEDURE x ,"abc")
```

A designational expression has been used as an actual parameter of type LABEL. As with arrays and switches only the name of the real procedure is used as a parameter. The parameters of the parametric procedure are included when the procedure is actually called, and not otherwise. A possible call of x within the body if p is:

```
rp(rp(3,"DEF"),st)
```

SUMMARY OF POINTS ON PROCEDURES

On entering a procedure the memory required is allocated dynamically according to the declarations. It follows that procedures are intrinsically recursive in nature, the limit on the depth of recursion being set by the available memory.

The body of a procedure is a "statement"; this may range from a simple (even dummy) statement to a compound statement or block. Within such a block there may of course be further procedure declarations, so that the following is a valid structure.

```
PROCEDURE tom;
BEGIN
      PROCEDURE dick;
      BEGIN
            PROCEDURE harry;
            BEGIN
                  s1; ....
            END;
            s2; ....
      END;
      s3; ....
END;
```

The scope of these procedures follows the normal rules of scoping, so that statements s1 and s2 may refer to tom, dick or harry; statement s3 may refer to tom and dick but not harry.

Statements within a procedure may make reference to any variable that is "in scope", not just those passed through the parameter list. In Fortran a COMMON statement would be necessary. It is also possible to jump out of a procedure by means of a GOTO statement to any label that is within scope.

There is a problem regarding the scoping of procedures in the case where the user declares a procedure of the same name as one of the pre-declared ones. This results from the fact that these pre-declared functions are compiled differently from those declared by the user, to make them faster and to economise on memory.

```
BEGIN INTEGER i;

      PROCEDURE abc;
      BEGIN ...
            z:=sin(y);
            ....
      END;

      REAL PROCEDURE sin(x);
      VALUE x; REAL x;
      BEGIN
            ....
      END;

      ....
END
FINISH
```

According to the Algol 60 Report the scope of the two procedures "abc" and "sin" is the block in which they are declared. The statement "z:=sin(y)" in the first procedure is referring to the second procedure in the block. In this compiler however the statement will generate code corresponding to a built-in procedure identifier that it already knows about. No error message is given. The problem could be avoided in this case by simply reversing the order of the two procedures, or better still changing the name so that no ambiguity can exist. There is no problem with procedures having names different from the built-in ones. As a general rule all procedure names are best kept unique amongst themselves and also from the variables.

INPUT/OUTPUT PROCEDURES

In Algol the exact form of Input/Output is not strictly defined but left up to the implementer to make best use of whatever facilities are available. Input/Output takes place through a series of procedures built into the runtime system. Input/Output is device independent. All Input/Output is associated with a stream or device number. In the case of non file-structured devices such as the console and printer Input/Output can be performed by simply choosing the appropriate stream number. These are summarised in appendix 2. In the case of disk files however some pre-dialogue is necessary to open or create a specified named file. In this case the corresponding stream numbers are allocated dynamically by the system. The procedures to perform this dialogue are described in a following section. In the case of disk files we also have the choice of serial or random access. The latter will be dealt with later.

SERIAL INPUT/OUTPUT

In all of the pre-declared I/O procedures the first parameter is the stream number denoted by dev. The name val indicates a REAL variable and ival an INTEGER variable. As the formal parameters are called by VALUE the actual parameters may contain expressions; the system will convert between integer and real values if necessary.

PROCEDURE skip(dev);

Outputs a carriage return/linefeed to dev.

INTEGER PROCEDURE chin(dev);

Read the next character from dev. The result of the procedure is the value of the character. In the case of disk input the character CONTROL-Z is returned at the end of file.

REAL PROCEDURE read(dev);
or
REAL PROCEDURE read(dev,label);

Read a floating point number or integer number from dev. The number is in free format, and is terminated by any character which cannot be part of a number. Decimal exponentiation is indicated by E. Spaces, tabs and blank lines preceding the number are ignored but other characters will give an error. A space will terminate the number except between the E and the exponent field. Integers may be read without rounding errors provided they appear as valid integers in the input i.e without decimal point or exponent parts. To allow the possibility of reading a file of unknown length, the second form given above may be used. In the event of passing the end of file control is passed to the label. The name is not preceded by the LABEL

indication as the compiler knows that the second parameter must be a label or a designational expression. End of file is a legal terminator; the jump will not happen unless another read is done. If the optional label is not given a runtime error occurs if end of file is passed. Examples of valid number formats are:

```
0.123  +1.23E -3    -123
```

The read routine will also accept the following although the output routines never generate such formats:

```
E-3   .123    -123.
```

It may be desirable to read a data source containing text comments. The read routine can be instructed to ignore any character preceding the number which cannot be part of the number by the call:

```
ioc(18);
```

A consequence of the use of this mode of reading data is that numbers of the form

```
E-6  or  .45
```

are no longer valid. The leading "E" or "." is regarded as comment; the actual numbers read in this case would be -6 and 45. To return to the default mode where comments are not permitted call:

```
ioc(19);
```

PROCEDURE text(dev,"string");

Output a string to dev. See the section on strings regarding interpretation of format and control characters. The string may also be a string parameter of the procedure in which text is called, in which case the actual parameter is the string identifier e.g.

```
PROCEDURE message(s); STRING s;
BEGIN text(1,s); .......
```

PROCEDURE chout(dev,ival);

Outputs a single byte to dev. If a character is to be output, its ASCII value must be used. This can be found by using the character literal facility. For example.

```
chout(1,&X);
```

will print X on the terminal.

```
PROCEDURE write(dev,ival);
or
PROCEDURE write(dev,ival,radix);
```

Prints ival as an integer on dev. The default radix is decimal. Non-significant characters are not printed. If formatted print is required use rwrite. Output in octal or hexadecimal is possible by including the optional third parameter.

```
radix=0    for decimal
radix=1    for octal
radix=2    for hexadecimal
```

Any other value for radix will lead to a runtime error.

```
PROCEDURE rwrite(dev,val,a,b);
or
PROCEDURE rwrite(dev,val);
```

Floating point output to dev. val is the value to output, a and b define the format such that:
a= total number of characters including sign and decimal point.
b= number of digits after the decimal point.
If b is zero then we have formatted integer output. If the value of a is inconsistent with that of b some large value will be substituted.
If a=0 then exponent format is used with b decimal digits.
If both a and b are zero or if they are omitted altogether as in the above example then the program defaults to exponent format with 6 decimal digits.

Various aspects of the output formatting can be controlled by calls to the predeclared procedure ioc. These calls have the effect of setting flags within the runtime system which remain in effect until some further call is made to change then. These calls to ioc can be considered in 3 groups. The first of each group is the default state in effect when the program starts. The various calls within each group are mutually exclusive.

The first group is concerned with what action is to be taken if the value to be output is too large to be accomodated by the specified format.

ioc(6)

The routine first attempts to accomodate the number by moving along the decimal point while maintaining the total field width constant. If this fails the routine will use exponent format provided the field width can be maintained else a row of asterisks "****" is printed indicating an out of range number.

ioc(7)

No format changes whatsoever are allowed. If the number cannot be accomodated then a row of asterisks is printed.

ioc(8)

No error print allowed. When this ioc call is in effect the error print indicated by a row of asterisks is never used. Format changes are allowed; if necessary exponent print will be used regardless of the field width specified.

The second group is concerned with the representation of spaces within the output format.

ioc(9)

Set the "default space character" to space (ASCII 32). Leading zeros are printed as spaces.

ioc(10)

Set the "default space character" to null (ASCII 0). Leading zeros will be suppressed. The number is left justified. (The null character is trapped by the routine and not actually sent to the output stream).

The third group is concerned with the representation of positive numbers.

ioc(11)

Use the current default space character (see group 2 above) where a positive sign is expected. Initially the default space character is space. If ioc(11) is called after ioc(10) the result is to suppress the character slot reserved to indicate a positive result.

ioc(12)

Print "+" to indicate a positive number.

NOTE: calls to rwrite and write are terminated by printing the "default space character" (see group 2 above). This is initially set to space which serves as a terminator to separate output such that it can be reread by the read routine.

INPUT OUTPUT SELECTION

The RML Algol system allows the user the ability to select input/output files or devices from within the program or from the console keyboard. For this purpose there exists a buffer into which I/O selections are placed through I/O stream number 7, or through a number of calls to ioc. The basic sequence of events consists of:

1. Place an I/O selection string into the buffer.
2. Call a command string interpreter to read the contents of the buffer and copy the string into an "input list" and/or "output list" as appropriate.
3. A call of predeclared procedures "input" or "output" reads the next entry in the "input list" or "output list" and returns to the program the appropriate stream number, having opened or created any necessary files.

Input from stream 7 is buffered and only made available to the program when a carriage return character is entered. Incorrect characters can be removed using the rubout key which on the RML 380Z is done rather more elegantly than on the pure CP/M version. There are two pointers associated with stream 7, one with input and the other output. As characters are entered or read from the buffer the appropriate pointer is advanced by 1. These pointers may be reset using the following ioc calls.

ioc(0)

Reset the input pointer. The next call of chin(7) will return the value of the first character in the buffer.

ioc(1)

Reset output pointer and write a string terminator into the first buffer position. The next call of chout(7,char) will place the value of char into the first position of the buffer and advance the position of the string terminator. Note that following the use of ioc(2) through ioc(5) described below, before reading from stream 7 the programmer should issue both an ioc(0) and an ioc(1) to reset the pointer and wipe out the current buffer contents.

The following ioc calls allow input or output lists to be entered.

ioc(2)

This produces a prompt on the console of the form:

OUT=IN?

The user then enters a command string of the general form:

outputlist=inputlist <cr>

When the carriage return <cr> character is given to terminate the command line the command string interpreter is called. Every character up to the separating equal sign (or carriage return if no equal sign present) is copied and stored as the current "output list" and everything after the equal sign is copied and stored as the current "input list". A pointer is associated with each of these lists and if a new input or output list entry is found then the corresponding pointer is reset to the start of that list. The detailed form of these lists is described later.

ioc(3)

This is similar to ioc(2) but the text is taken directly from the contents of the buffer without any user prompt. A typical calling sequence to set up an input/output list might be:

```
ioc(1); text(7,"outputlist=inputlist"); ioc(3);
```

ioc(4);

This produces a prompt on the console of the form:

```
INPUT=
```

The user then enters a command string of the general form:

```
inputlist <cr>
```

This string then becomes the current "input list", the output list remaining unchanged.

ioc(5)

This is similar to ioc(4) but the text is taken directly from the contents of the buffer without any user prompt. A typical calling sequence might be:

```
ioc(1); text(7,"inputlist"); ioc(5);
```

Note: A call of ioc(3) or ioc(5) leaves the contents of the buffer unaffected. The same string may if desired be parsed twice to set up both input and output files of the same names. This is in fact done within the compiler to select its input and output.

The general form of the input and output lists consists of a sequence of one or more device or file specifications separated by commas e.g.

```
CON:,A:OUT1,,LST:=DATA.DAT[B],RDR:
```

In the above example 4 output channels and 2 input channels are specified. A call of the pre-declared procedures input or output (decribed later) will scan the appropriate list from the current position up to the next occurrence of a comma or end of list indicator. A stream number will be returned corresponding to the

A list of the device mnemonics used and their corresponding stream numbers is given in appendix 2.

A CP/M file specification is of the general form:

DRIVE:FILENAME.EXT

The characters recognised within file names are letters, digits, '$' and '?'; the latter should be reserved for specifying ambiguous file names. Lower case letters are converted to upper case as per the normal CP/M convention. All characters less than space (ASCII 0 to 32) are ignored within I/O lists.

The FILENAME consists of from 1 to 8 characters. The file extension ".EXT" if present consists of from 1 to 3 characters. If no extension is given a default value will be assumed; this is initially set to three spaces. The method of changing the default file extension is described under library procedure "swlist" in the section "Input/Output directly to or from memory". It is possible to force the use of the default file extension regardless of what is given by the call:

ioc(20)

In order to return to the default situation where a specified file extension takes precedence call:

ioc(21)

The DRIVE: consists of one of A:, B:, C:, D: or may be omitted. If omitted a default is assumed according to the following rules. At the start of each list the assumed drive is the "logged on drive" when the program is first entered. Any subsequent drive specified within the list then becomes the default for following entries.

Switch options may be added to any input/output device or file specification and consists of a series of up to 12 characters enclosed within square brackets. Lower case letters are converted to upper case. Switch options must not contain a comma or equal sign. Certain switches are recognised by the runtime system and acted upon; in the example given above the input file DATA.DAT[B] the switch [B] causes the file to be opened for "random access" reading. Other switches not used by the system may be used by the program. A facility exists for the program to read the switch list directly.

The occurrence of two adjacent commas within an I/O list is equivalent to specifying the "null" input/output device NL: (stream 0).

The ioc calls described above will have set up input/output lists. These lists may now be used to assign files or devices through the predeclared procedures input and output.

dev:=input

will read the next entry in the "input list". If the entry is found to be a device then dev will be assigned a value corresponding to that device name. If a disk file was specified then that file will be opened. A buffer region will be allocated to contain the file control block and sector buffer (if serial access). The stream number returned will be from 64 upwards, the actual value indicating which buffer is allocated to that file.

A negative value for dev indicates an error e.g. bad syntax, no entry found in input list or no file found of that name.

dev:=output

Similar to input but for output files or devices. A number of options exist regarding what action is to be taken if an output file name specified is found already to exist. These options are selected by calls to ioc which set the appropriate flags within the runtime system. The first is the default case.

ioc(13)

No checks are made. A second file of the same name will be created. A problem may be encountered later on trying to access such files.

ioc(14)

The existing file of the name specified will be deleted before the new file is created.

ioc(15)

If a file name is found to already exist the call of output will return a stream number of -100. No new file is created.

CLOSING AND DELETING FILES

When the use of a file is completed it should be closed by a call of the predeclared procedure:

close(dev)

This will close the file associated with stream dev by a previous input or output call. If dev does not correspond to a disk file nothing happens. NOTE: If an OUTPUT FILE is not closed its contents will be LOST. Input files should also be closed, as this call also serves to release the buffer and file control block associated with that file and makes it available for further use.

delete(dev)

This will delete the file associated with dev by a previous input or output call and release the file control block and buffer for reuse.

INPUT/OUTPUT SUPPORT ROUTINES

The following additional procedures are recognised by the runtime system and are made known to the compiler by including the text of ALIB.ALG with the program source.

rewind(dev);

The serial input or output file associated with dev is (first closed in the case of output files and) rewound for reading from the beginning.

dev:=findinput("string");

This call will open the file or device defined in "string" for input on stream dev. If the first character of "string" is found to be a question mark '?' then the effect is as follows. The remainder of the string is printed on the console as a prompt to the operator who enters the required input file or device name. e.g.

```
dev:=findinput("?Source file=");
```

will prompt the operator:

```
Source file=
```

who then enters the required name.

```
dev:=findinput("DATA.DAT");
```

opens the file DATA.DAT on the logged on drive.

The input specification may in fact consist of an "input list" the first entry of which will be used and assigned to dev. Note that the use of this procedure will wipe out any previous input specifications waiting in the input list.

dev:=findoutput("string")

This is analogous to findinput but for output. The output specification may if desired be generalised to be a complete input/output list as described under ioc(2) and ioc(3) above.

i:=rename;

This procedure renames a file. The old filename and drive information is taken as the next entry in the "inputlist". The new filename is taken from the next entry in the "outputlist", e.g.

```
ioc(1);
text(7,"FRED.ABC=B:JOE.XYZ");
ioc(3);
i:=rename;
```

will rename file JOE.XYZ on drive B: as FRED.ABC. Note that the CP/M rename utility will rename all files that satisfy the input specification. On exit

i=-1	implies a failure e.g. file not found or illegal syntax.
i=255	CP/M reply from rename regardless of success or failure.

The default file extension will be used if none is specified, or if ioc(20) is in effect, will be used regardless. If a file of the same name as the new name given is found already to exist then the result will be the same as described under procedure "output" with regard to calls of ioc(13) to ioc(15), namely:

ioc(13)	No checks are made.
ioc(14)	Erase any pre-existing files of the same name.
ioc(15)	Return the value -100 in i.

i:=newext(j,"XYZ")

The file associated with stream j by a previous call of input or output is closed and its file extension changed to the 3 character string given as the second parameter. This string becomes the default file extension e.g.

```
j:=findinput("FRED.ABC");
i:=newext(j,"XYZ");
```

will rename the file FRED.ABC as FRED.XYZ. No checks are made as to the pre-existence of files of the same name. A negative result in i implies a failure; the expected reply is 255.

a:=bios(n,bc)

This procedure performs a direct call through the BIOS jump vector where

```
n = entry in the jump table (0 to 14)
bc= contents of BC register pair on entry.
a = contents in A register on exit.
```

Refer to the CP/M System Alteration Guide for details.

a:=cpm(c,de)

This procedure performs a direct call to CP/M where

c = contents of C register on entry (0 to 27)
de= contents of DE register pair on entry.
a = result in A register on exit.

Refer to the CP/M Interface Guide for details.

i:=fcblock(dev);

This returns in i the address of the file control block associated with file stream dev. This can be useful only to users who wish to manipulate CP/M facilities directly.

i:=exflt(a,t);

Extend the file control block list. The RML Algol system is initially set up to allow 4 serial files and 2 random access files open at any time. Should users require more than this number of files then this procedure may be used to extend the list of file control blocks available. Each call extends the length of the list by one. On exit a negative value in i indicates an attempt has been made to extend beyond its maximum length of 16 entries. The parameters to exflt are:

a = address of buffer to use
t = file type
If t=0 then serial file else random access

The buffers used are user declared arrays, the address of which is found using procedure location e.g.

```
BEGIN  BYTE ARRAY buf[0:160];
       i:=exflt(location(buf[0]),0);
```

The buffer sizes required are for serial files 161 bytes (33 for file control block + 128 for sector buffer) and for random access files 33 bytes. It is the user's responsibility to ensure that the array is large enough to accommodate the buffer and that such buffers do not overlap or become overwritten.

INPUT/OUTPUT DIRECTLY TO OR FROM MEMORY

As an aid to text processing and related manipulation e.g. setting up file extensions or reading the switch list a facility exists to read or write using the standard input output routines directly to or from anywhere in memory. Such I/O is associated with stream number 10. A number of string handling routines relevant to the following are described in the section on "library procedures". Before I/O can be

performed via memory it is necessary for the user to set up pointers to where input/output is to occur. As each character is read/written the corresponding pointer is advanced by one. The following procedures to manipulate these pointers are in ALIB.ALG.

seti(a)

Set the INPUT pointer to the address a.

seto(a)

Set the OUTPUT pointer to the address a.

In practice a call of location would probably have been used to find the address. In order to find the current values of the input/output pointers:

i:=ipoint

Returns in i the current address of the input pointer.

i:=opoint

Returns in i the current address of the output pointer.

A typical sequence might be:

```
BEGIN  BYTE ARRAY buf[0:1000];
       seto(location(buf[0]);
       seti(location(buf[0]);
       rwrite(10,x,0,6); .......
       i:=opoint; .......
       x:=read(10);
```

It is the user's responsibility to ensure that such I/O stays within the declared bounds of the array buffer used.

i:=swlist

Returns in i the address of the switch list.

The user can check if any switch options have been specified following a call of "input" or "output" by reading the contents of this switch list. These switches (a maximum of 12 characters) can be read using input stream 10. A typical sequence might be:

```
seti(swlist);
i:=chin(10);
```

The first switch is now in i. The list is terminated with a zero value. The switch list always contains information relevant to the most recent call of the procedures "input" or "output".

The default file extension is stored in the 3 bytes [illegible]
list. This can

characters into the appropriate buffer by means of output to stream 10 e.g.

```
seto(swlist+13);
text(10,"XYZ");
```

This sequence will set the default file extension to XYZ. On entry the default extension is set to null, i.e. 3 spaces.

This technique can also be used as a way of reading small quantities of data in a manner similar to the DATA statement of BASIC. E.g.

```
seti(sloc("1.32  99.6 ....."));
FOR i:=1 STEP 1 UNTIL 20 DO x[i]:=read(10);
```

The procedure sloc is described in the section on library procedures. Another example involving text can be found in the program VDU.ALG on the distribution disk.

RANDOM ACCESS FILES

A file may be opened to be read by random access rather than serial access. Such files are opened as "input" files with a switch [B] set to signify block I/O. If the file is to be updated i.e. written to then an additional switch is needed [BM] where the M indicates "modify". These rules imply that only pre-existing files may be opened for random access. As example of an "input" specification.

```
DATA1.DAT[B],DATA2.DAT[BM]
```

The first file is opened for random access reading and the second for reading/writing.

i:=rblock(dev,a,b,n)

will read n blocks from the disk file associated with stream dev, starting at block number b, writing the contents in memory starting at address a. The length of the transfer is 128*n bytes. The first block of the file is block number 0. The address in general will correspond to part of an array set up by means of procedure location (see section on library procedures) e.g.

```
i:=rblock(dev,location(buf[0]),b,10);
```

On exit i will have the following meaning.

- i=0 successful read.
- i=1 read past end of file.
- i=2 reading unwritten data.
- i=3 hard error.

The user should ensure that the declared array is large enough

to accept the transfer. Any part of a selected transfer extending beyond the end of file will be set to zero.

i:=wblock(dev,a,b,n);

Will write n blocks to disk; the parameters are the same as for rblock. On completion i can take the following values.

i=0	successful write.
i=1	error in extending file.
i=2	end of disk file.
i=3	hard error.
i=255	no more directory space available.

LIBRARY PROCEDURES

The following procedures are built into the runtime system and can be made known to the compiler by including the source of file "ALIB.ALG" with the program. Some of the following are machine dependent but all are available for the RML 380Z. See the listing of "ALIB.ALG" in appendix 4 for the formal definitions of the procedure parameters.

i:=location(x)

This returns with i set to the address of variable x; x may be REAL, INTEGER, or an array element of type REAL, INTEGER, or BYTE. In the case of REAL or INTEGER arguments the address returned is that of the slot assigned to that variable (see description of the workings of the runtime system). Each slot occupies 4 bytes and in the case of INTEGERs only the upper half is used so that in this case 2 should be added to get the actual address containing the integer. Array elements as arguments always return the correct address. The procedure works by recalling the most recent variable address computed; as the argument is called by value the compiler will in fact accept any expression as the actual parameter, although the result will correspond to the final variable specified. Users who wish to to find the address of Boolean variables may construct a similar procedure with the same body as location but with a formal parameter of type Boolean by value.

i:=fspace

This returns the number of bytes free (allowing for a safety margin for stack operations). Note that on large systems the result may exceed 32K and thus appear to have a negative value in two's complement representation.

blmove(s,f,len)

Block move of len bytes starting at address s to the block starting at address f. In general the use of procedure location (see above) would be used to set up the addresses e.g.

```
blmove(location(a[0]),location(b[0]),100)
```

It is the user's responsibility to ensure that such block moves stay within the limits of the declared arrays. This procedure will work correctly if the two blocks overlap.

```
i:=peek(a)
```

Returns the byte value contained within the address given by a.

```
poke(a,i)
```

Sets the contents of address given by a to the value of (the 8 least significant bits of) i.

```
a:=in(c)
```

Input from a port. This procedure executes an IN A,(C) instruction.

```
out(c,a)
```

Output to a port. This procedure executes an OUT (C),A instruction.

```
b:=parity(i)
```

This Boolean procedure returns TRUE if the character value of i (8 least significant bits) has EVEN parity else FALSE.

SHIFTS AND ROTATES

In the following procedures v is the value (type INTEGER) and n is the number of places to shift or rotate. Note that only the 4 least significant bits of n are used so its value should be in the range 0 to 15.

```
i:=shl(v,n)     Shift LEFT.

i:=lsr(v,n)     Logical shift RIGHT.

i:=asr(v,n)     Arithmetic shift RIGHT.

i:=rotl(v,n)    Rotate LEFT.

i:=rotr(v,n)    Rotate RIGHT.
```

Arithmetic shift right extends the sign bit whereas logical shift right always places zeros into vacated positions.

```
x:=random
```

Returns a pseudorandom number in the range 0 to 1.

clarr(a,len)

Clear array area of length len bytes starting at address a.

dpb(u,t,s,a)

Set up the disk parameters, u=unit number (0 to 3), t=track, s=sector, a=DMA address.

i:=rdisk

Read the disk directly using information set up in a previous call to dpb. The result from the CP/M call will be in i.

i:=wdisk

Write to disk directly using information set up by a previous call to dpb. The result from the CP/M call will be in i.

i:=sloc("string")

Returns in i the address of the start of the string. The actual parameter may also be a string parameter of a procedure e.g.

```
PROCEDURE x(s); STRING s;
BEGIN INTEGER i;
        i:=sloc(s);
        i:=sloc("XYZ");
```

Strings consist of a series of characters stored in sequential bytes terminated by a zero.

atext(dev,s);

This is similar to the pre-declared procedure "text" but the second parameter is the address of the string. e.g.

text(dev,"XYZ"); is equivalent to atext(dev,sloc("XYZ"));

i:=tlen(s)

Returns the length of the string whose address is at s. E.g.

i:=tlen(sloc("XYZ"));

returns the value 3.

i:=smatch(long,short)

This procedure compares two strings looking for the first match within the long string corresponding to the contents of the short string. The parameters are the addresses of the strings. If a match is found the value of i is set to the address within the long string corresponding

match is found i will be set to zero. Additional matches may be found by giving as the starting address of long the value one greater than the result of the previous match.

The following procedures are only available for the RML 380Z.

i:=emt(n)

The program executes an "EMT n" instruction; on exit i contains the value returned in the accumulator.

wait(n)

Delay for 10*n msecs, with n in the range 1 to 255.

The following procedures are concerned with low resolution graphics on the RML 380Z.

chpos(x,y)

The values of x and y define a coordinate position on the VT screen with the origin set 4 lines up. The valid ranges for x and y corresponding to the screen are thus

0 <= x < 40 , -4 <= y < 20

Output can now be directed to the screen starting at the specified position through device stream 11. As each character is written the x coordinate is advanced one position to the right. Once text extends beyond the limits of the screen it stays off for all subsequent text until reset by a further call of chpos or point (see below). The use of graphics will in general be associated with a call of emt(13) to clear the screen and set the 4 line scroller (see COS manual). Data may also be read back from the screen through device stream 11; again the x coordinate is advances after each character read. Once beyond the screen the value returned will be CONTROL-Z. For example to write "Hello" in the middle of the screen:

```
i:=emt(13);
chpos(17,8);
text(11,"Hello");
```

point(x,y,z)

This procedure plots a point on the screen of intensity z at position (x,y), where:

z=0	dot off
z=1	dim dot
z=2	bright dot
z>2	plot the character value of z

In this case the coordinates (x,y) are in terms of the basic graphical unit, again with the origin 4 lines up, so that:

$$0 <= x < 80 \quad , -12 <= y < 60$$

This routine also set the screen coordinates for I/O through device stream 11 as described under procedure chpos.

line(x1,y1,x2,y2,z)

This procedure draws a line from position (x1,y1) to position (x2,y2) where the values of x, y, and z are as defined for point.

LIBRARY INSERTS

A facility exists which allows the contents of "library" source files to be included with the body of the program at compile time, e.g.

```
LIBRARY "B:ALIB.ALG"
```

or, using the upper case convention,

```
'LIBRARY'  "B:ALIB"
```

The effect at compile time is that on encountering the language key word LIBRARY the compiler looks for an input file specification enclosed within string quotes. This file is opened and its contents included with the program source at the points the call is found. In the above case the file ALIB.ALG on drive B: is read, the default extension being ".ALG". The default drive is the logged on drive. This facility allows the user to construct libraries of frequently used procedures thus avoiding duplication of text and excessive editing.

```
BEGIN INTEGER i,j,k;

LIBRARY "ALIB"
LIBRARY "IOLIB"
LIBRARY "STATLIB"

PROCEDURE abc; ......
```

This example would include the contents of three library files in turn when compiling. These files may if desired themselves contain LIBRARY directives. The limit on the depth of such calls is set by the maximum number of input and output files that may be open at any one time. In the compiler as distributed this limit is set to five.

EXAMPLE PROGRAMS

The following examples illustrate various aspects of the language. The first four are fairly straightforward; the final two examples assume a fairly advanced knowledge of mathematics. Further examples can be found on the distribution disk.

The first example lists a table of the integers up to 20, together with their square roots, on the console.

```
BEGIN INTEGER i;

      FOR i:=0 STEP 1 UNTIL 20 DO
      BEGIN   rwrite(1,i,5,0);
              rwrite(1,sqrt(i),0,6);
              skip(1)
      END
END
FINISH
```

The second example simply lists a file on the console. On detecting the end of file it loops back for further files to list.

```
BEGIN INTEGER i,d;

      {get input file}
loop: ioc(4); d:=input;
      {check if valid file}
      IF d<64 THEN text(1,"*NTry Again")
      ELSE
      BEGIN   {list file on console}
              FOR i:=chin(d) WHILE i#&^Z DO chout(1,i);
              close(d);       {release FCB}
      END;
      GOTO loop;      {go round again}
END
FINISH
```

The next example is a procedure to illustrate string handling. The routine makes use of several procedures from ALIB.ALG. The procedure scans a piece of text starting at address "old" and substitutes every occurrence of a given string "olds" by that given in "news". The source is itself in the form of a string, i.e. terminated with a zero value. The resultant string will start at address "new". The calling sequence

```
        la:=location(a[0]);
        lb:=location(b[0]);
        substitute(lb,la,"Jack","Z1");
        substitute(la,lb,"Jill","Z2");
```

will replace every occurrence of "Z1" by "Jack" and "Z2" by "Jill". Both the initial string text and the resultant string start at location a[0]. The array b is used as working space.

```
    PROCEDURE substitute(new,old,news,olds);
    VALUE new,old; INTEGER new,old;
    STRING news,olds;
    BEGIN INTEGER i,j,ns,os,nl,ol,oldfin;

          ns:=sloc(news);
          os:=sloc(olds);
          nl:=tlen(ns);              {addresses of strings}
          ol:=tlen(os);                        {lengths of strings}
          {address of closing zero of input string}
          oldfin:=tlen(old)+old+1;
          {look for matches}
          FOR i:=smatch(old,os) WHILE i#0 DO
          BEGIN   j:=i-old;                    {length of text to copy}
                  blmove(old,new,j)            {move over portion of text}
                  new:=new+j;                  {update pointers}
                  old:=old+j+ol;               {skip old string}
                  blmove(ns,new,nl);           {copy in new string}
                  new:=new+nl;                 {update pointer}
          END ;
          blmove(old,new,oldfin-old);          {copy remainder}
    END substitute;
```

The fourth example, quicksort, is a sorting algorithm originally developed by C.A.R. Hoare. An array of values is sorted into ascending order. The method involves reordering terms such that it can be partitioned in the form

```
a[low],a[low+1],...a[i-1] < a[i] <= a[i+1],a[i+2],...a[high]
```

The pivot value in this case is arbitarily chosen as the value of the final element on entry. The procedure then calls itself recursively for each side of the above expression until each partition contains only one term. The following coding exploits a feature of this compiler that the value of the loop variable on exit from a loop will be that which led to the loop's termination. This may not be the case on other Algol compilers.

```
PROCEDURE quicksort(a,low,high);
VALUE low,high; INTEGER low,high;
INTEGER ARRAY a;
IF low<high THEN
BEGIN  INTEGER i,j,pivot,x,y;
       i:=low-1; j:=high; pivot:=a[j];
loop:  i:=i+1;
       FOR x:=a[i] WHILE x<pivot DO i:=i+1;
       j:=j-1;
       FOR y:=a[j] WHILE j>i AND y>=pivot DO j:=j-1;
       IF i<j THEN
       BEGIN a[i]:=y; a[j]:=x; GOTO loop;
       END ;
       {move pivot to centre}
       y:=a[high]; a[high]:=x; a[i]:=y;
       {always deal with the smaller partition
       first to minimise depth of recursion}
       IF i-low<high-i+2 THEN
       BEGIN   quicksort(a,low,i-1);
               quicksort(a,i+1,high)
       END ELSE
       BEGIN   quicksort(a,i+1,high);
               quicksort(a,low,i-1)
       END
END quicksort;
```

The next example is a statistical test. Observations are made in pairs, the first of each pair belonging to one group and the second to another. To find out if there is any difference between the two groups, we first find the total difference between them. We calculate the probability of getting this difference by chance, if the pairs of observations had been assigned randomly to each group, rather than always the first of the pair to the first group. If there are j pairs, there are 2^j ways of assigning the pairs into the groups. Each combination must be tested by finding the total difference between the groups and counting the number of occasions on which this difference is greater than or equal to that actually observed. This count divided by 2^j is the probability of observing the difference by chance. In the program, the differences between the observations in each pair are held in the array d. The procedure br adds to the sum the difference indicated by the parameter n with sign indicated by s. Unless n indicates the last difference, it generates two more sums, one with a positive difference and one with a negative difference. Each time n reaches 1 the total sum is checked to see if it is greater than or equal to the observed sum of differences.

To solve this problem without recursion involves a number of nested FOR loops equal to the number of data pairs. Thus, a separate program would have to be kept for each number of data pairs.

```
BEGIN INTEGER i,j,count;
REAL a,b,obs;
ARRAY d[1:100];

PROCEDURE br(n,s,sum);
VALUE n,s,sum;
INTEGER n,s; REAL sum;
BEGIN  sum:=sum+d[n]*s;
       IF n=1 AND abs(sum)>=abs(obs) THEN count:=count+1 ELSE
       IF n#1 THEN
       BEGIN br(n-1,1,sum); br(n-1,-1,sum)
       END
END br;

       text(1,"*Nnumber of pairs?"); j:=read(7);
       obs:=0;
       FOR i:=1 STEP 1 UNTIL j DO
       BEGIN   a:=read(7); b:=read(7);
               obs:=obs+a-b; d[i]:=abs(a-b);
       END ;
       text(1,"*Nsum of differences");
       rwrite(1,obs,8,2); count:=0;
       br(j,-1,0); br(j,1,0);
       text(1,
       "*NProbability of same or greater with random signs ");
       rwrite(1,count/2^j,7,3);
END
FINISH
```

The final example is a procedure for solving simultaneous equations. The left hand side matrix is set up in a two dimensional array a[row,column] and the right hand side in a vector b[row]. The array names (a and b are only examples) are passed to the procedure to correspond to the names lhs and rhs, together with an integer giving the number of equations and a label to exit to if there is no solution. The answers are left in the right hand side vector. The method uses a Gaussian elimination with partial pivoting.

```
    PROCEDURE solve(order,lhs,rhs,fail);
    VALUE order;
    INTEGER order; ARRAY lhs,rhs;
    LABEL fail;
    BEGIN  INTEGER row,col,row1,order1,i,j;
           REAL max;

           FOR order1:=order STEP -1 UNTIL 2 DO
           BEGIN   max:=0;
                   FOR j:=1 STEP 1 UNTIL order1 DO
                   IF abs(lhs[j,order1])>max THEN
                   BEGIN max:=abs(lhs[j,order1]); row:=j;
                   END ;
                   IF row#order1 THEN
                   BEGIN   max:=rhs[order1];
                           rhs[order1]:=rhs[row];
                           rhs[row]:=max;
                           FOR col:=1 STEP 1 UNTIL order1 DO
                           BEGIN   max:=lhs[order1,col];
                                   lhs[order1,col]:=lhs[row,col];
                                   lhs[row,col]:=max
                           END
                   END swop equations;
                   IF lhs[order1,order1]=0 THEN
    nosol:         BEGIN   text(1,"*Nno solution");
                           GOTO fail
                   END ;
                   FOR j:=STEP 1 UNTIL order1-1 DO
                   BEGIN   max:=lhs[j,order1]/lhs[order1,order1];
                           rhs[j]:=rhs[j]-rhs[order1]*max;
                           FOR col:=1 STEP 1 UNTIL order1 DO
                           BEGIN   lhs[j,col]:=
                                   lhs[j,col]-lhs[order1,col]*max
                           END zero one element;
                   END zero one column;
           END triangularise the left hand side;
           IF lhs[1,1]=0 THEN GOTO nosol;
           FOR row:=1 STEP 1 UNTIL order DO
           BEGIN   rhs[row]:=rhs[row]/lhs[row,row];
                   FOR row1:=row+1 STEP 1 UNTIL order DO
                   rhs[row1]:=rhs[row1]-lhs[row1,row]*rhs[row];
           END ;
    END solve simultaneous equations;
```

COMPILING AND RUNNING PROGRAMS UNDER CP/M

Running RML Algol is a two stage process:

1.Compiling: The program source is read by the compiler to produce an output file in a form to be read by the runtime system.

2.Running: This stage loads the file output from the compiler and runs it.

The simplest sequence of commands given a program source in a single file "PROG.ALG" would consist of:

To compile program: ALGOL PROG

To run program: ARUN PROG

We will now consider these activities in more detail. The two basic files involved are:

The compiler	ALGOL.COM
The runtime system	ARUN.COM

The default disk drive for input and output files is the logged on drive. The default file extensions are:

Source files	.ALG
Compiler output	.ASC
Monitor file	.MON

COMPILING

In the simplest case given above the compiler reads the program source from the file specified; if no file extension is given then the default will be used. The output file created is given the same name as the source file but with the extension ".ASC". Any pre-existing file of the same name as the output file will be deleted before the new output file is created. If the compiler detects any errors in the program source the output file is deleted but compilation continues until the end of the source, checking for further errors. Error messages are sent to the console. At the the of compilation the size of the resulting program is printed and control is returned to CP/M.

A more general form of calling the compiler is:

ALGOL outlist=inlist

For example:

ALGOL OUT=IOLIB,B:MATHS,PROG

Using this method

procedures, ending with the file containing the program. It should be remembered that the overall source should correspond to the required Algol block structure, from the first BEGIN to the final corresponding END and FINISH. Files may be taken from several drives; if the drive is not specifically included then the current default will be used. This is discussed in the section on I/O selection. In the example IOLIB.ALG is taken from the logged on drive and MATHS.ALG and PROG.ALG from drive B:. The output OUT.ASC goes to the logged on disk. An alternative (and perhaps better) way of combining source files is by the use of the LIBRARY facility previously discussed. It must be remembered however that the use of such library calls is restricted to the final file specified in the input list otherwise the remaining input file specifiers will be overwritten. This is discussed in the details of library procedure "findinput".

If a second output stream is specified then a listing of the compiler identifier tables will be generated. Compiler error messages will also be sent to this stream along with an indication of the maximum table size the system can support.

ALGOL OUT1,OUT2=PROG

will send the compiler output to OUT1.ASC, and all compiler error messages and identifier tables go to OUT2.MON.

ALGOL OUT1,CON:=PROG

will send errors and identifier tables to the console.

If no input/output is specified in the call, or if an error exists e.g. bad syntax or a non-existent source file is given, then the compiler will prompt for I/O. For example a call of the form:

ALGOL

will result in a prompt of the form:

OUT=IN?

The user may now specify a list of input and output files as for the above case.

The output from the compiler is about the same length as the corresponding source text.

If the I/O files were specified in the initial calling line i.e. "ALGOL PROG" then upon completion the compiler will return to CP/M. If the I/O files were given as the result of a prompt from the compiler, then upon completion the compiler will be restarted, to allow further programs to be compiled. A reply of CONTROL-C in this case will return control to CP/M.

RUNTIME SYSTEM

Given a successfully compiled program, the output file so created may now be run by a calling the runtime system as follows:

ARUN filename

The assumed file extension is ".ASC". The file specified will be loaded and then executed. If no input is specified or if an error is found e.g. bad syntax or non-existent filename then the runtime system will prompt the user for an input file. For example a call of the form:

ARUN

will prompt for input:

INPUT=

to which the user responds with the required filename.

Upon completion of the program the system prints "~" on the console and waits for an operator response. Typing CONTROL-P will rerun the program or CONTROL-C will return control to CP/M.

If a runtime error is detected then suitable diagnostic information is sent to the console (see section on runtime errors). Unless the user is making use of the error handling facility (see procedure "error" in library section) the system will now wait for the operator to investigate the cause of the error. The program may be rerun from the beginning by typing CONTROL-P or control returned to CP/M by typing CONTROL-C.

The return to CP/M upon completion or upon detecting a runtime error can be made automatic by a call of

ioc(22)

within the program.

A call of the form

ioc(60)

causes an immediate restart of the program from the beginning. Any files open at the time will not be closed although all file control blocks are released.

CORE IMAGE FILES

If the user intends to run a program many times it may be saved on disk in a more compact core image format by specifying a switch [S] at runtime. e.g.

ARUN filename[S]

Upon loading the program the system will delete the original ".ASC" file produced by the compiler and create a new file of the same name and extension in core image format. Future calls of the form

ARUN filename

will now automatically load and run from this core image file. As CP/M reserves a minimum of 1K bytes per file the saving may not be significant for small programs (except for some saving in the actual loading time.) The original ".ASC" file was in a system independent format, but the new core image format is not relocatable and so any changes to the runtime system which affect its size will make it necessary to recompile the program.

COMPILER ERROR MESSAGES

FAIL X ON LINE Y IDENT Z SYMBOL S
"LINE UP TO ERROR"

X is the failure number (see below), Y the line on which it occurred, Z the last identifier read, and S the decimal value of the last symbol (see section entitled "compiler representation of basic symbols"). "LINE UP TO ERROR" is a copy of the input line up to and including the symbol at which the error was found. The compiler output is switched off and the file deleted. The compiler however continues to check the syntax of the remainder of the program. In all compilers a tradeoff is made between the amount of error information given and the size and speed of the compiler. In this implementation the emphasis has been to produce a compiler that can be used on a very modest sized computer. There is always a danger, particularly with a one pass compiler, that following the detection of a genuine error the system may fail to synchronize fully and thus produce additional spurious errors.

1 Identifier declared twice in same block.
2 Undeclared identifier.
3 No [after array name, except as a procedure parameter, or ordinary procedure used as a function.
4 No] at end of subscript list.
5 More than 255 variables in the main program or a procedure.
6 No FINISH at end of program. (Too many ENDs).
7 No ELSE part of a conditional arithmetic expression.
8 No ELSE part of a conditional Boolean or conditional designational expression.
9 Relational operator not found where expected. Will occur if the first arithmetic expression of a Boolean relational expression

is totally enclosed in round brackets.

10 Arithmetic primary does not start with +,-,.,(, digit or identifier.

11 %, MOD, !, MASK, or DIFFER does not have two integer operands.

12) missing in arithmetic expression.

13 Controlled variable in FOR is undeclared or subscripted.

14) missing in Boolean or designational expression.

15 More identifiers in scope than the tables can accomodate. The compiler automatically makes the tables as large as possible on a given system.

16 Statement starts incorrectly. If this occurs at the terminating FINISH is means there are not enough ENDs.

17 Undeclared or unsuitable identifier on left of :=

18 Array declaration faulty.

19 Type specification of actual parameter is not LABEL, PROCEDURE, REAL PROCEDURE, BOOLEAN PROCEDURE or INTEGER PROCEDURE.

20 Wrong number of subscripts. In the case of formal arrays, this error cannot be detected until runtime.

21 No) after actual parameter list.

22 FOR statement element not terminated by , or DO.

23 Procedure body not delimited by ;.

24 := not found where expected.

25 No THEN after IF.

26 VALUE specification is not the first specification of procedure formal parameters.

27 FINISH in middle of program. Possibly an unmatched BEGIN, " or '.

28 Procedure formal parameter list not ended by).

29 Parameter specified twice, or is not in formal list, or specification not terminated by ;.

30 Label/procedure list full.

31 UNTIL not found where expected.

32 No (after name of standard procedure (except input or output).

33 THEN followed immediately by IF.

34 Procedure actual parameter starts with an undeclared identifier.

35 Function or variable used as procedure.

36 procedure input or output is followed by a (.

38 Arithmetic expression contains Boolean variable in illegal context.

39 Parameter specified VALUE is not in formal list.

40 Parameter specification not complete.

41 An array has been called by value.

42 Input/output procedure call error.

44 Integer literal not in range.

46 Switch list does not end with ;.

47 Switch has more than one subscript.

48 Word BYTE not followed by ARRAY.

49 Input files exhausted without end of program recognised.

50 A procedure used before its declaration was assumed to be of a type different from the actual type. Try reordering procedures to eliminate the forward reference.

51 Input file specified in a LIBRARY call not found.

COMPILER IDENTIFIER TABLE AND IDENTIFIER TYPES

The compiler may be instructed to print on the console or to the monitor file a list of all the identifiers declared, together with information about their type and the addresses they will occupy in the memory. Variables are placed on a stack and the variable number is the position on the stack relative to a pointer. The pointer is held in location PBASE in the runtime program. The address of the variable is found by multiplying the variable number by 4 and adding this to the contents of PBASE.

Four numbers are printed after each identifier in the compiler identifier table.

The first of these is the stack position except for labels and procedures. For labels and procedures the symbolic label number is printed. This is the digits part of a symbol such as L123 which is output by the compiler.

The second number is the procedure number of the enclosing procedure in which the identifier is declared. The main program is 0, and the procedures are numbered serially as they are encountered, regardless of depth of declaration. As an exception, the actual number of a procedure is printed, instead of the number of the enclosing procedure.

The third number is the line number of the source program.

The fourth number is the current size of the compiled code. This information can be related to the position given when runtime errors are detected.

The type information of the identifier is then listed as follows. The numbers represent the internal representation of the data types.

0	procedure formal parameter (type not yet known)
1	real
2	integer
3	Boolean
5	real array
6	integer array
6	byte array
7	Boolean array
8	switch
10	procedure
11	real procedure
12	integer procedure
13	Boolean procedure
14	label

The compiled code of a procedure contains a list of the types of the parameters. The following types may appear, in addition to those above.

4	string
21	real by name
22	integer by name
23	Boolean by name

COMPILER REPRESENTATION OF BASIC SYMBOLS

These are the decimal values which are printed in a compiler error message. Language key words are represented in the Algol source by the word enclosed in single quotes or in upper case and in the compiler by 40*1st letter+second letter.

If a compiler error message contains a symbol which is not on the list, an illegal compound symbol has been detected. The usual cause of this is an unmatched single quote.

letters A-Z		1-26
[		27
]		29
^	(exponentiation)	30
:=		7000
!		33
"	(string brackets)	34
#	(not equal to)	35
$		36
%	(integer divide)	37
>=	(greater or equal to)	38
(		40
)		41
*	(multiply)	42
+		43
,		44
-		45
.		46
/	(real divide)	47
digits 0-9		48-57
:		58
;		59
<	(less than)	60
=		61
>	(greater than)	62
<=	(less or equal to)	63
BEGIN		85
BOOLEAN		95
BYTE		105
AND		118
ARRAY		122
COMMENT		135
DIFFER		169
DO		175
ELSE		212
END		214
EQUIVALENT		217

FALSE	241
FINISH	249
FOR	255
GOTO	295
IF	366
IMPLIES	373
INTEGER	374
LABEL	481
LIBRARY	489
MASK	521
MOD	535
NOT	575
OR	618
PROCEDURE	658
REAL	725
STRING	780
STEP	780
SWITCH	783
THEN	808
TRUE	818
UNTIL	854
VALUE	881
WHILE	928

RUNTIME ERRORS

In the event of an error condition being detected during program execution, a message is sent to output device 1 (console or video screen generally) of the form:

```
ERROR n
ADD  PBASE PROC LOC
aaaa bbbb   p1   d1
aaaa bbbb   p2   d2
  .    .    .
aaaa bbbb   0    dC
```

where: n error number (see following list)
p1 procedure number where error was detected
aaaa address of program counter
bbbb value of PBASE at error
(see section on runtime system)
d1 location of program counter relative
to the start of the program

Both aaaa and bbbb are printed in hexadecimal.
The procedure number can be found by counting procedures from the beginning of the program starting from 1. The main program is given the number 0. This information can be found in the compiler identifier table output. If p1 is non-zero the calling sequence is then printed on the following lines until the outermost level (p=0) is reached. This traceback information can be used to investigate the nature of failure

in greater detail if required (see section describing working of the runtime system). The information given in dl etc can be related to the corresponding information given in the compiler identifier tables to help locate the position of errors. The program may be restarted from the start by typing CONTROL-P or control returned to CP/M by typing CONTROL-C.

RECOVERY FROM RUNTIME ERRORS

In normal operation a program is terminated by the detection of a runtime error. It is possible however to continue following an error allowing the program to exit in a controlled manner e.g. close output files, give more useful diagnostic information, values of variables and so on. This recovery is achieved by including a call of procedure "error" (in ALIB.ALG) in the program before the failure occurs e.g.

```
error(LABEL crash);
```

On detecting an error the runtime system will produce the error information given above and then transfer control to the label (or designational expression) "crash" in the user's program. It is advisable that the label be located at an outermost pragram level as it may only be reached if it is within scope at the time of the error. The error number responsible for the failure can be found by means of a call to chin(13) e.g.

```
crash: i:=chin(13);
       IF i>30  THEN GOTO cpmbug ELSE .....
```

RUNTIME ERROR NUMBERS

0 Undefined error. This implies that an error has been detected which has no corresponding entry in the error list. This hopefully will only occur where the user has made modifications to the runtime system and failed to update the error list.

1 Variable space used up (stack overflow). Probably the result of excessive recursion or array declarations too large for the available memory. The error traceback may fail under these circumstances (the first line should always be correct). Overflow is checked following block or procedure entry and array declarations.

2 Procedure called with the wrong number of parameters.

3 Procedure called where the actual and the formal parameter types do not match.

4 Array used with the wrong number of subscripts.

5 Array subscript out of range (below base of array).

6 Array subscript out of range (above top of array).

7 Integer division by zero.

8 Real division by zero.

9 Real overflow.

10 Real to integer conversion overflow.

11 Real overflow detected during normalisation after real arithmetic operation.

12 Error in READ. Character read which is not a legitimate part of a number (ASCII value is less than 48 ie <&0).
13 As for 12 but ASCII value is greater than 57 (ie >&9).
14 Error in READ. Number contains two or more decimal points.
15 Error in READ. A character + - . or E found with no associated digits.
16 Square root of a negative number.
17 Exponential argument too large (>87).
18 Exponential argument marginally too large.
19 Logarithm of a negative number.
20 Logarithm of zero.
21 Table item out of range (below). Found in ioc(n), chin(n), chout(n,c) etc where n<0.
22 As for 21 but where n is greater than maximum value specified in list.
23 End of file detected during READ.

ERRORS IN LOADER

24 Loader syntax error. Output from compiler has been corrupted?.
25 End of input is indicated (CONTROL-Z read) but no program has been loaded. Selecting input device 0 will produce this effect.
26 On completion of input there remain unresolved forward references. Input source is corrupt?
27 No program present on a restart.
28 Label tables overlap program. Program is too large for available memory.
29 Forward reference tables full. This error should be rare but can be avoided by reordering procedures so that they generate fewer forward references, i.e try to arrange that procedures are declared before they are called.
30 Non relocatable core image input file is not compatible with this runtime system.

CP/M ERRORS

31 Channel number is out of range.
32 No directory space found during output.
33 Attempt to read from channel not open for input.
34 Attempt to read from a non serial channel.
35 Attempt to read past end of file.
36 Attempt to write to a channel without write access.
37 Attempt to write to a non serial file.
38 Error in extending file.
39 Attempt to output to random access file without write access.
40 Attempt at random access to a serial access channel.
41 Channel not open.
42 Attempt to rewind a random access file.
43 Random access with a negative block number.
44 No slot available for input or output.
45 Attempt to create an output file for random access.
46 Random access transfer attempted with a block count less than zero or greater than 255.

RUNTIME STACK ORGANISATION

Some of this information will be useful when Algol variables are to be accessed from machine code added to the runtime system.

The stack extends from the end of the runtime program to the end of available memory, as found by interrogating the system. The variable stack grows upwards from the end of the program and a working stack, used in evaluating expressions, passing procedure parameters, and CALL instructions, grows downwards from the end of memory. The variable stack consists of a number of frames, one for the main program and one for each procedure call. Within each stack frame is an array stack, which contains an array frame for each depth of array declaration. In the following section "word" refers to a 16 bit (2 byte) quantity.

The following pointers are used. Their addresses can be found from the listing in the section entitled "adding code sections".

PBASE points at the current variable stack frame

MBASE points at the main program stack frame. When the main program is executing locations PBASE and MBASE contain the same value.

WSBAS points at the base of the working stack in the current level. It is used to delete floaters from the working stack at Algol labels.

ABAS points at the current array frame

FSPT points at the next free location in the variable stack.

The following registers are also of significance.

SP points at the top item of the working stack. It must be saved and restored if used by any machine code added. It is also used for CALL instructions.

IX should also be restored if used. It is the Algol interpretive code program pointer.

IY must be restored if used. It points to a series of flags and working space.

Each stack frame is divided into two parts, a variable part and an array part. The variable part is divided into slots which are each two words (four bytes) long. The actual address of a slot is found by multiplying the slot number by 4 and adding this to the base address which is held in PBASE or MBASE. In the main program frame the first declared variable is in slot 2 and the word pointed at by MBASE contains 0, the level number of the main program. In procedure level frames slot 3 is used for the result of a function and is unused in procedures which do not deliver a result. The procedure parameters occupy slots 4 and upwards, followed by variables declared within the procedure. The first word of each procedure in the compiled program contains the number of

variable slots required by the procedure. The first word of the main program points at the last word of the compiled program which contains the number of variable slots required by the main program.

In procedure frames the first three slots are used for linking information. Starting at the word pointed at by PBASE (slot 0) the words contain the following information.

Word 1. The number of the procedure.
Word 2. The return address
Word 3. PBASE of calling level
Word 4. WSBAS of calling level
Word 5. ABAS of calling level

A variable stack slot may contain any of the following types of item:

1. A real number which is held in the standard four byte format.
2. An integer number or Boolean value which is held in the highest addressed word of a slot. Booleans use only the least significant byte of this word.
3. A label or procedure address, always a procedure parameter. The address itself is in the highest addressed word and in the word below it is the value of PBASE at the time the address was evaluated.
4. The address of an array or a switch either as a declared variable or a procedure parameter. The address is in the highest addressed word of the slot, the remaining word being unused.
5. The address of a string or an unsubscripted variable for procedure parameters of type string and variables called by name.

The address in a switch variable points to the switch vector. The word pointed at contains the number of elements in the switch and subsequent words the addresses of the labels in the switch list.

The array part of a stack frame contains a number of array levels, numbered by depth of declaration within a procedure or the main program. Level 0 always exists and is located immediately above the end of the variables. ABAS points at the base of the current level, which contains the depth of that level. The next word (except in level 0) countains a pointer to the level below. Above the level information are the dope vectors and array elements.

An array variable points at the start of its dope vector. This contains 2*(N+1) words, where N is the number of subscripts. The first word of the dope vector contains the number of bytes occupied by each element (1, 2 or 4), the second the number of subscripts and the third the lower bound of the first subscript.

There are two additional words for each additional subscript. The first contains a multiplier for the previously accumulated element number and the second the lower bound of the next subscript. The final word of the dope vector contains the address of the word beyond the end of the array elements. Array elements themselves are stored immediately after the dope vector.

RUNTIME OPERATION CODES

These are the operation codes which are output by the compiler. The list gives their number in decimal.

Expressions are evaluated using a working stack. The top element is referred to as S1, the next one down as S2 and so on. The stack pointer 'SP' is used for this stack (and also for CALL instructions). It grows down from the top of available memory.

Some of the interpretive routines take data from the program. N1 refers to the next byte after the code, N2 to the next, and so on. In the following section "word" refers to a 16 bit (2 byte) quantity.

0 No operation.

1 Declare array. N1=depth of declaration. N2=number of declarations in multiple. N3=variable number of first declaration. N4=number of bytes in each element. N5=number of subscripts S1=upper bound of last subscript. S2= lower bound of last subscript. S3, S4 etc., bounds of other subscripts.

2 Formatted print. S1=b, S2=a, S3=value, S4=device number.

3 Read to S1 from input device in S1.

4 Store local variable from S1. N1=variable number.

5 Print string. Followed by 7 bit ASCII code, terminated by zero. Device number is in S1.

6 Integer print S1=radix, S2=value, S3=device number.

7 Read next character to S1 from device number in S1.

8 Print S1 as character, S2=device number.

9 Jump. Location is in next word.

10 Leave procedure.

11 Enter procedure with no parameters. Address is in next word.

12 Get local variable to S1. N1= variable number.

13 Integer add. S1:=S2+S1

14 Get array element. N1=procedure number, N2=variable number, N3=number of subscripts. The subscripts are on the stack. The main program is procedure number 0.

15 Store array element. S1=value, other information as code 14 except subscripts are in S2 etc.

16 Set 16 bit constant in S1 from N1 and N2.

17 Integer negate. S1:=-S1

18 Real ^ Integer. S1:=S2^S1

19 Integer multiply. S1:=S2*S1

20 Integer divide. S1:=S2/S1

21 Integer subtract. S1:=S2-S1

22 S1:=S1=0

23 S1:=S1>0

24 S1:=S1<0

25 Get any variable to S1. N1= procedure number, N2=variable number.

26 Store to any variable from S1. N1, N2 as for 25.

27 Standard function. Followed by another code.

2 sqrt	3 sin	4 cos
5 arctan	6 exp	7 ln
8 sign	9 entier	10 abs

28 Jump if S1=FALSE. Address in next word.
29 Set zero in S1.
30 S1:=NOT S1
31 S1:=S1 AND S2
32 S1:=S1 OR S2
33 S1:=S1 EQUIV S2
34 For statement calculator. S1= address of controlled variable. S2= final value. S3=increment. S4=0 for no increment at the first test, else -1. N1=type of control variable (0=REAL else INTEGER). The following word contains the exit address for loop completion.
35 "ioc". Parameter in S1
36 Enter procedure. N1=number of parameters. S1=type of last parameter. S2=Value of last parameter, and so on, in reverse order. The address of the procedure is in N2 and N3. The first word of a procedure is the fixed space on the variable stack required by the procedure. The following bytes contain the procedure number and the number of parameters expected, followed by the type specification of the parameters, in reverse order.
37 Store outer block variable from S1. N1=variable number.
38 Fetch outer block variable to S1. N1=variable number.
39 Set in S1 the address of the variable whose procedure number is in N1, variable number in N2.
40 Skip, device number in S1.
41 Integer S1:=sign(S2-S1)
42 Set 8 bit constant in S1 from N1.
43 Fix S1.
44 Float S1.
45 Set floating point constant from next 4 program bytes.
46 Floating negate.
47 Set label in S1. Address in next word. Second word of S1 becomes variable stack base pointer.
48 Evaluate switch address. S1=address of element 0, S2=subscript. On exit S1 contains address.
49 Real ^ real. S1:=S2^S1
50 Floating multiply. S1:=S2*S1
51 Float S2.
52 Floating divide. S1:=S2/S1
53 Floating add. S1:=S2+S1
54 Floating subtract. S1:=S2-S1
55 Store parameter called by name. S1=value, S2=address.
56 Floating S1:=sign(S2-S1).
57 Jump to address in S1.
58 Enter procedure without parameters whose address is in S1.
59 As 58 but number of parameters in N1. For 59 the rest of the stack is set up as for code 36.
60 Print string whose address is in S1, S2=device number.
61 Set stack depth. N1=procedure number, N2=array depth required.
62 Fetch parameter called by name. S1=address.
63 Stop, end of program. Prints ^ on console or returns to CP/M.
64 Store an address in the program in a local variable. Followed by the variable number in N1 and the address in the next whole word.
65 Jump to the address in local variable number N1.

66 Set in S1 the address of local variable number N1.
67 As for 66 but main program variable.
68 Get local array element.
69 Get main program array element.
70 Store local array element.
71 Store main program array element. Codes 68-71 are followed by the variable number in N1, not by the level number and then the variable number as for codes 14 and 15.
72 Read a floating point number, check for end of file. S1=address of label to go to on end of file. S2= device number.
73 Logical OR. S1:=S2 OR S1
74 Logical AND. S1:=S2 AND S1
75 Logical EXCLUSIVE OR. S1:=S2 XOR S1.
76 Integer MOD. S1:=S2 MOD S1
77 Close file, stream number in S1.
78 Delete file, stream number in S1.
79 Open INPUT and assign to stream number S1.
80 Create OUTPUT and assign to stream number S1.

ADDING CODE SECTIONS

The most likely additions users may wish to make to the runtime system are:

1. adding input/output device handlers
2. specialised code subroutines
3. linking the above to the runtime error handling system

For each of these applications the additions can be made without getting deeply involved in the inner workings of the system. The following listings from the runtime system are all the user needs to understand. See the section "runtime stack organisation" where the significance of several of the important variables is discussed. Users who wish to bypass CP/M and create a stand alone system should refer to Research Machines for details.

Each of the above applications is associated with a list of items. Pointers to these lists can be found near the start of the runtime system. The general form of a list is

```
NAME:   n          :length
        #entry 0
        #entry 1
        ....
        #entry n
```

The first byte defines the length of the list counting from 0. (Note the RML assembler allows "#NAME" as a shorthand for "DEFW NAME"). The name of the list has a corresponding pointer near the start of the runtime program. Note that the exact location and contents of these lists given here may differ from the actual values in the distributed system but the pointer address will be correct. The entries in the list contain the addresses of the corresponding entry points. Other variables of

importance are:

PROGST — contains the start of free space. The program code will start from here.

ENDLIST — contains the address beyond which the lists must not extend.

The various lists follow each other consecutively in memory. To add I/O handlers or code routines the following modifications must be made.

1. Write the code starting at the address contained in PROGST.
2. Update PROGST to a new free space pointer
3. Make an entry in the appropriate list and update the length of the list if necessary.

Each list contains vacant slots or room left at the end for expansion. If this is insufficient, it will be necessary to relocate the entire list(s) and update the pointer(s) accordingly. Other points to observe are:

IX and IY — must be preserved in value on exit.
SP — is used for subroutine calls and for the working stack. Exit from code routines is via a RET instruction and so any use of SP should leave it uncorrupted and balanced before exit.

Other registers may be used as required.

INPUT DEVICE HANDLERS

INLP — Pointer to input list.

INLIST — Name of input list.

Input device handlers read a byte from the input source to register A then exit via a RET instruction. c:=chin(n) will read a byte from the device associated with entry "n" in the list.

OUTPUT DEVICE HANDLERS

OUTLP — Pointer to output list.

OUTLST — Name of output list.

Output device handlers output the byte in register A then exit via a RET instruction. chout(n,c) will send character "c" to device "n" in the list.

IOC LIST AND CODE SUBROUTINES

IOCLST — Pointer to IOC list.

DKLST — Name of IOC list.

ioc(n) will transfer control to the address associated with entry "n" in the ioc list. Return is via a RET instruction. If the ioc is in a procedure body (as in ALIB examples) then access to the procedure parameters is straightforward. (See section on runtime stack organisation). Procedure parameters will occupy from slot 4 onwards and the result, if it is a TYPE procedure, will go in slot 3. The address of the slot is given by

(contents of PBASE)+4*slot number

Thus, the first procedure parameter will be given by

(contents of PBASE)+16

The various contents of these slots are described in the section on stack organisation. Note that integer values occupy the upper half of the 4 byte slots and so it is necessary to add 2 to the above sum.

EXAMPLE CODE PROCEDURE

Consider adding the "peek" procedure (already in ALIB.ALG) using the information given in the listing from the runtime system that follows. The Algol definition using the next available slot in the "ioc" list would be:

```
INTEGER PROCEDURE peek(a);
VALUE a; INTEGER a; ioc(61);
```

The code routine could be as follows.

```
;EXAMPLE CODE ROUTINE "PEEK"
;POINTERS IN THE ALGOL RUNTIME SYSTEM
PROGST  EQU     115H
PSTART  EQU     24EFH
DKLST   EQU     1702H
IOCLEN  EQU     60D
PBASE   EQU     127H

;UPDATE ORIGIN FOR ALGOL PROGRAM
        ORG     PROGST
        DEFW    ENDPATCH

;UPDATE IOC LIST LENGTH
        ORG     DKLST
        DEFB    IOCLEN+1

;ENTRY IN IOC LIST
        ORG     DKLST+2*IOCLEN+3
        DEFW    PEEK

;CODE ROUTINE
        ORG     PSTART
PEEK:   LD      L,4             ;SLOT NUMBER
        CALL    GPAR            ;GET PARAMETER 1
        EX      DE,HL
```

```
        LD      C,(HL)          ;CONTENTS OF ADDRESS
        LD      B,0
        LD      L,3             ;SLOT NUMBER FOR RESULT
        CALL    DPAR            ;DUMP RESULT
        RET                     ;BACK TO ALGOL
;---------
;GET INTEGER VALUE PARAMETER
;ENTER WITH SLOT NUMBER IN L
;EXIT WITH RESULT IN DE
GPAR:   CALL    SLAD            ;ADDRESS OF SLOT
        LD      E,(HL)
        INC     HL
        LD      D,(HL)          ;CONTENTS IN DE
        RET
;---------
;DUMP INTEGER RESULT
;L=SLOT NUMBER
;BC=RESULT
DPAR:   CALL    SLAD
        LD      (HL),C
        INC     HL
        LC      (HL),B
        RET
;---------
;COMPUTE ADDRESS OF SLOT
SLAD:   LD      H,0
        ADD     HL,HL
        INC     HL
        ADD     HL,HL           ;SLOT*4+2
        LD      DE,(PBASE)
        ADD     HL,DE           ;ADDRESS OF SLOT+2
        RET
;---------
ENDPATCH: END
;-------------------------------
```

This program after assembly would be combined with ARUN.COM using the CP/M utility DDT.

LINKING TO RUNTIME ERROR HANDLING

Another possible application is to link the user written code to the runtime error handling mechanism. The table has a slightly different form this time.

JFAIL	Contains a jump to the fail routine.
ERRORS	Points to the error table.
ERLST	Name of error list.

For example, suppose that an error condition exists if the negative flag is set. A call of the following form would be made

```
ERR50:  CALL    M,JFAIL  ;ERROR IN ...
```

The error list entries consist of the return address i.e. #ERR50+3 in this case. The fail routine compares each item in the list with the value on the top of the stack, and the position of the match gives the "error number".

Other pointers of interest are.

INP1 — contains a JUMP to the input handler associated with stream number 1.

OUTP1 — contains a jump to the output device handler associated with stream number 1.

These are used directly by the system bypassing the select stream mechanism in order to p-in. out titles and error information and for checking for operator responses.

INTERRUPT HANDLING

The RML Algol system itself does not use interrupt in any way. There is no reason however why users should not link in code routines which do use interrupt, provided care is taken to ensure that the contents of all registers and working variables are not corrupted. Algol variables may be modified directly if desired. The addresses of simple variables can be passed to code routines using a call by name procedure parameter. The addresses of arrays (or variables) can be found using the procedure "location". Two ioc calls have been added which allow for interrupt to be controlled while disk I/O is in progress. These are

ioc(16)

This call will cause interrupt to be disabled before entering CP/M. Interrupt will be enabled on return from CP/M.

ioc(17)

This call will restore the default situation. No action is taken with regard to interrupts.

EXTRACT FROM ALGOL RUNTIME SYSTEM

```
                0001 *H (ARUN.ZSM)
                0002 ;ALGOL INTERPRETER FOR Z80
                0003 ;CGM/RHA OCT 78
                0004 ;CP/M VERSION
                0005 ;19/2/79 DISK 25A
                0006
0100            0007         ORG     100H
```

```
                0008            RADO
0100 C35B12     0009            JP      START   ;OCTAL DEFAULT RADIX
0103 C35B12     0010            JP      RESTART ;PROGRAM START ADDRESS
0106 C3031B     0011 INIT:      JP      INITIALISE ;INITIALISATION SEQUENCE
0109 C3D113     0012 WAIT:      JP      DONE    ;WAITING CODE
010C C34413     0013 JFAIL:     JP      FAIL    ;ENTER FAIL ROUTINE
010F C3D51A     0014 INP1:      JP      READ1   ;STREAM 1 INPUT
0112 C3DE1A     0015 OUTP1:     JP      PRNT1   ;" " OUTPUT
0115 EF24       0016 PROGST:    #PSTART         ;START OF PROGRAM CODE
0117 C816       0017 INLP:      #INLIST         ;INPUT DEVICE LIST
0119 E516       0018 OUTLP:     #OUTLST         ;OUTPUT DEVICE LIST
011B 0217       0019 IOCLST:    #DKLST          ;IOC LIST
011D 9D17       0020 ERRORS:    #ERLST          ;ERROR LIST
011F CA18       0021 ENDLIST:   #STOPLIST       ;LISTS MUST STOP HERE
0121 0D20       0022 DEVTAB:    #DVTAB          ;DEVICE MNEMONICS
0123 0000       0023 ABAS:      #0              ;ARRAY FRAME BASE
0125 0000       0024 WSBAS:     #0              ;WORKING STACK BASE
0127 6701       0025 PBASE:     #T9-2           ;PROCEDURE BASE POINTER
0129 6701       0026 MBASE:     #T9-2           ;MAIN BASE POINTER
012B C000       0027 FSPT:      #0              ;FREE SPACE POINTER
```

```
                0573 ;INPUT DEVICES
16C8 CD         0574 INLIST:    13D
16C9 E014       0575            #DUMIN          ;DUMMY, RETURNS ^Z
16CB D51A       0576            #READ1          ;KEYBOARD
16CD F01A       0577            #TTYIN          ;TELETYPE
16CF E014       0578            #DUMIN          ;DUMMY
16D1 FA1D       0579            #CPMKEY         ;CON
16D3 F81D       0580            #CPMRDR         ;RDR
16D5 E014       0581            #DUMIN          ;DUMMY
16D7 B216       0582            #RECIN          ;RECORD INPUT
16D9 E014       0583            #DUMIN
16DB E014       0584            #DUMIN
16DD 3118       0585            #C10I           ;ARRAY BUFFER I/O
16DF E014       0586            #DUMIN
16E1 E014       0587            #DUMIN
16E3 F413       0588            #ERNUMB         ;LATEST ERROR NUMBER
                0589
                0590 ;OUTPUT DEVICES
16E5 CD         0591 OUTLST:    13D
16E6 EF1A       0592            #NULL           ;DUMMY BIT BIN
16E8 DE1A       0593            #PRNT1          ;SCREEN
16EA C01B       0594            #TTOUT          ;TELETYPE
16EC C01B       0595            #LPR            ;LINE PRINTER
16EE F91D       0596            #CPMVT          ;VT
16F0 F71D       0597            #CPMPUN         ;PUN
16F2 F61D       0598            #CPMLST         ;LST
16F4 6416       0599            #RECOUT         ;RECORD OUTPUT BUFFER
16F6 EF1A       0600            #NULL
16F8 EF1A       0601            #NULL
16FA 3A18       0602            #C100           ;ARRAY BUFFER I/O
16FC EF1A       0603            #NULL
16FE EF1A       0604            #NULL
1700 EF1A       0605            #NULL
```

```
                0606 ;-----------------------------------------------
                0607 ;IOC CALLS
1702 3C         0608 DKLST:  60D                ;LENGTH
1703 7816       0609         #RESETI            ;RESET INPUT BUFFER POINTER
1705 7F16       0610         #RESETO            ; " OUTPUT " "
1707 8816       0611         #CDECK             ;CALL I/O COMMAND DECODER
1709 9016       0612         #CDEC              ; " " " " " BUFFER
170B 9416       0613         #CINDK             ;" I " " " KEYBOARD INPUT
170D 9C16       0614         #CIND              ;" " " " " BUFFER
170F 4B12       0615         #DEFFC             ;RWRITE DEFAULT
1711 5012       0616         #NOFC              ; " NO FORMAT MODS
1713 5412       0617         #NOERP             ; " NO ERROR PRINT
1715 3C12       0618         #LEADS             ; " PRINT LEADING SPACES
                0619         ;10
1717 3712       0620         #LEFTJ             ; " LEFT JUSTIFY PRINT
1719 4612       0621         #SPLUS             ; " SPACE FOR + SIGN
```

```
1777 F61E       0672         #RENAME            ;RENAME FILE
1779 731F       0673         #NEWEXT            ;CHANGE FILE EXTENSION
                0674         ;60
177B 5812       0675         #RERUN             ;RERUN PROG FROM START
                0676 ;-----------------------------------------------
177D            0677         DEFS    20H        ;FOR EXPANSION
                0678 ;-----------------------------------------------
                0679 ;LIST OF CALLS TO 'FAIL'
179D 2D         0680 ERLST:  45D
179E 5313       0681         #ERR0+3
17A0 0C14       0682         #ERR1+3
17A2 E007       0683         #ERR2+3
17A4 0A08       0684         #ERR3+3
17A6 F908       0685         #ERR4+3
```

```
17F2 3516       0723         #ERR42+3
17F4 E01D       0724         #ERR43+3
17F6 621E       0725         #ERR44+3
17F8 E515       0726         #ERR45+3
                0727 ;-----------------------------------------------
17FA            0728         DEFS    10H        ;ROOM FOR MORE
                0729 STOPLIST:                  ;NO LISTS BEYOND HERE
                0730 ;-----------------------------------------------
```

APPENDIX 1

SUMMARY OF IOC CALLS

ioc(n)	FUNCTION
0 - 5	input/output selection
6 - 12	rwrite format control
13 - 15	output file options
16 - 17	interrupt option on disk I/O
18 - 19	read options
20 - 21	file extension options
22	reboot CP/M on completion
23 - 59	linked to procedures in ALIB.ALG
60	rerun program from start
61 - 62	RML 380Z graphics

PRE-DECLARED PROCEDURES

NAME	SECTION
ioc	pre declared identifiers
abs	Standard functions
arctan	" "
cos	" "
entier	" "
exp	" "
ln	" "
sign	" "
sin	" "
sqrt	" "
chin	Input/Output
chout	" "
read	" "
rwrite	" "
skip	" "
text	" "
write	" "

APPENDIX 1 PROCEDURES IN "ALIB.ALG"

(to be compiled with the user's program if required).

findinput	Input/Output
findoutput	" "
rblock	" "
wblock	" "
rewind	" "
seti	" "
seto	" "
ipoint	" "
opoint	" "
exflt	" "
fcblock	" "
swlist	" "
bios	" "
cpmd	" "
rename	" "
newext	" "
error	Runtime errors
location	Library procedures
fspace	" "
blmove	" "
peek	" "
poke	" "
in	" "
out	" "
dpb	" "
rdisk	" "
wdisk	" "
parity	" "
shl	" "
lsr	" "
asr	" "
rotl	" "
rotr	" "
random	" "
clarr	" "
sloc	" "
slen	" "
smatch	" "
atext	" "
emt	" "
wait	" "
chpos	" "
point	" "
line	" "

APPENDIX 2

INPUT/OUTPUT STREAMS

The following table defines the stream or device numbers associated with I/O channels. These numbers point to an entry in the input/output device lists which contain the address of the appropriate device handler. Vacant slots are available for users to add their own routines. In the case of disk files the stream numbers are allocated dynamically by the system (from 64 upwards). Figures in brackets indicate equivalent stream numbers in the pure CP/M version.

INPUT STREAMS

STREAM	NAME	DEVICE
0	NL:	Dummy input - always returns an end of file character (CONTROL-Z).
1 (4)	TI:	Single character keyboard input.
2* (5)	TTY:	Teletype (SIO4).
4	CON:	CP/M Console.
5	RDR:	CP/M Reader.
7	TIB:	Buffered keyboard input. On the RML 380Z, the line editing using the rubout key is done more elegantly than for the pure CP/M version. Used for I/O file command lines.
10	---	Input from memory (see seti/seto library procedures).
11* (0)	---	Input from VT screen (graphics mode).
13	---	Returns as the next byte the latest runtime error number.

OUTPUT STREAMS

STREAM	NAME	DEVICE
0	NL:	Dummy output - All output is thrown away.
1 (4)	VT:	VT Screen.
2* (5)	TTY:	Teletype (SIO4).
3* (6)	LP:	Lineprinter (EMT 5).
4	CON:	CP/M Console.
5	PUN:	CP/M punch.
6	LST:	CP/M List device.
7	---	Output to I/O file command buffer.
10	---	Output to memory (see seti/seto library procedures).
11* (0)	---	Output to VT screen (graphics mode).

* I/O only available on RML 380Z.

Disk drives recognised are A:, B:, C:, and D:.

Switch options recognised are:

B	Block I/O (random access).
M	Modify access (random access write).

APPENDIX 3 DIFFERENCES FROM THE ALGOL 60 REPORT

RESTRICTIONS

No OWN variables.

No multiple assignments.

No integer labels.

Variables must be declared before use.

Call by NAME is restricted to the case where the actual parameter is a variable name, i.e. as per call by reference in Fortran or call by location in CORAL 66.

Array parameters most be called by name.

The 'fat comma' is not implemented.

Clues are sometimes required to indicate the "type" in procedure parameters and conditional statements.

Only the first six characters of identifier names are significant.

EXTENSIONS

Data type: BYTE ARRAY.

Operators: MOD, !, DIFFER, MASK.

Comments may be enclosed within matching braces e.g. {like this}.

Procedure names may be the result of designational expressions.

Additional functions include string handling, direct disk I/O, block move, clear array, graphics, etc. See appendix 1 for the full list.

APPENDIX 4 PROGRAMS ON THE DISTRIBUTION DISK

ALGOL.COM and ARUN.COM

The Algol compiler and runtime system.

ALIB.ALG

The standard Algol library routines.

LCASE.ALG and LCASE.ASC

This program will convert Algol source files written using convention 1 (all upper case with key words enclosed in quotes) into convention 2 (upper/lower case). The program prompts for input and output file names. The default file extension is ".ALG".

UCASE.ALG and UCASE.ASC

This in the complement to LCASE.ALG just described. Files are converted from convention 2 into convention 1.

MMIND.ALG and MMIND.ASC

Mastermind. Game 1 allows 6 colours and no blanks. Game 2 allows 6 colours and blanks.

VDU.ALG and VDU.ASC

This program is designed as an editing aid for creating Algol source files. The program prompts for an output file name, the default extension is ".ALG". If the upper case convention with key words enclosed within quotes is required then give a [U] switch option with the file specification. Now start typing in your program. The program detects language key words. As soon as sufficient characters have been entered to uniquely define the key word the program will supply the rest. Corrections can be made to the current line being entered using the rubout key. Other special keys are.

CONTROL-U	Erase the current line.
CONTROL-R	Retype the current line after cleanup.
CONTROL-X	Switch off the auto keyword facility. A second call will switch it on again. This allows strings etc. to be entered without extra characters being added.
CONTROL-Z	End of program. Close file and return to the start.
CONTROL-C	In response to the prompt for a file specification will return control to CP/M.

BIBLIOGRAPHY

Peter Naur (Ed.)

Revised Report on the Algorithmic Language Algol 60.

ICL

Algol: Language (Students Edition)
Technical publication 3340
Obtain from:
Technical Publications Service
International Computers Ltd.
ICL House
Putney
LONDON SW15, ENGLAND

This textbook also contains a copy of the Algol 6
Report.

Ekman Froberg

Introduction to Algol Programming.
Oxford University Press.

Daniel D. McCracken

A Guide to Algol Programming.
John Wiley and Sons Inc., New York.

TABLE OF CONTENTS